Easy

A Spectrum Book

INTERFACING PROJECTS FOR THE COMMODORE 64

Jim Downey, Don Rindsberg, and William Isherwood

JAMES M. DOWNEY, DON RINDSBERG,
AND WILLIAM ISHERWOOD

EASY INTERFACING PROJECTS FOR THE COMMODORE 64

Prentice-Hall, Inc.,
Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Downey, James M.
Easy interfacing projects for the Commodore 64.

"A Spectrum Book."
Includes index.
1. Computer interfaces. 2. Commodore 64 (Computer) I. Rindsberg, Don. II. Isherwood, William. III. Title.
TK7887.5.D68 1985 001.64'4 84-18266
ISBN 0-13-223553-6

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-223553-6

Editorial/production supervision: Joe O'Donnell Jr.
Cover design: Hal Siegel
Manufacturing buyer: Gary Orso

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Prentice-Hall Hispanoamericana, S.A., *Mexico*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

CONTENTS

PREFACE

The Commodore 64 represents a significant landmark in the ever-developing personal computer revolution. It is sophisticated enough to be a true computational system suited to the needs of industry, commerce, the laboratory, and the home. Yet, it has a price tag that puts it in what has traditionally been thought of as the low-end computer market. While the Commodore 64 is, undoubtedly, the prototype for a new generation of powerful yet affordable systems, right now it enjoys a unique place in the marketplace. Millions of them have been purchased at the time of this writing. Interestingly enough, one of the Commodore 64's most outstanding features is also the one that is most often overlooked: the ease with which it can be interfaced to a wide variety of devices.

The Commodore engineers incorporated this interfacing versatility early in the design stage of the 64. No less than five separate ports have been provided to allow the Commodore 64 to communicate with the outside world. This book will explain what these ports are and show you how to interface your Commodore 64 to just about anything. The projects that we describe have been included because they meet several important criteria. First, as the title indicates, they must be easy. We have made an effort to keep the complexity of most of the projects to a bare minimum. Recognizing that building these projects will probably represent your first plunge into that intimidating and mystical world of computer hardware, we have tried to choose projects that require only a handful of components.

This was an easy task, thanks to the sophistication built into the Commodore's input-output ports.

Second, the projects were chosen to be informative. This book is intended to serve as a primer in digital design. The fundamentals of digital logic chips are presented along with a detailed description of the 6510 microprocessor and how it is implemented in the Commodore 64. The theoretical aspects of each project are explained with the hope that they will enable you to become proficient enough to design your own custom computer applications based on the principles outlined in these pages.

Third, we have tried to describe projects that would appeal to most users. The projects include printer interfaces, an EPROM programmer, speech synthesizers, a modem, mechanical actuators, and more. Home-built equipment has always been the hacker's solution to the problem of high-priced or unavailable peripherals. We hope that we have included a project or two that will meet your specific needs.

Although you need not have had any prior experience working with electronic hardware, a working knowledge of BASIC is assumed. After you have implemented an interface, you must provide the computer with a program that operates it. Whenever possible, we have provided examples of such programs in BASIC. Sometimes, however, we have had to resort to machine language routines to fully use the hardware. We have included a chapter that provides a brief overview of machine language in the Commodore 64 to aid you in understanding the examples we present. We do not intend this text to be a treatise on machine language programming, nor is it necessary for you to become proficient in machine language to benefit from this book.

One final word of caution is in order. All the projects presented have been fully tested by use and will work if properly constructed. Although we have tried to anticipate most of the construction problems you are likely to encounter, we obviously cannot think of all the possibilities. Errors in wiring, shorts, or faulty components will obviously prevent normal operation. Thus, you should be aware that you may encounter difficulty in getting some of these projects to work. Be careful, follow the instructions faithfully, and have patience—the rewards are worth it. You should also be aware that certain errors in wiring could conceivably damage your computer as well. Doublecheck all wiring before connecting any device to the Commodore 64. Also, never insert a board or plug into the Commodore 64 while it is operating! Good luck and welcome to the world of computer hardware.

1

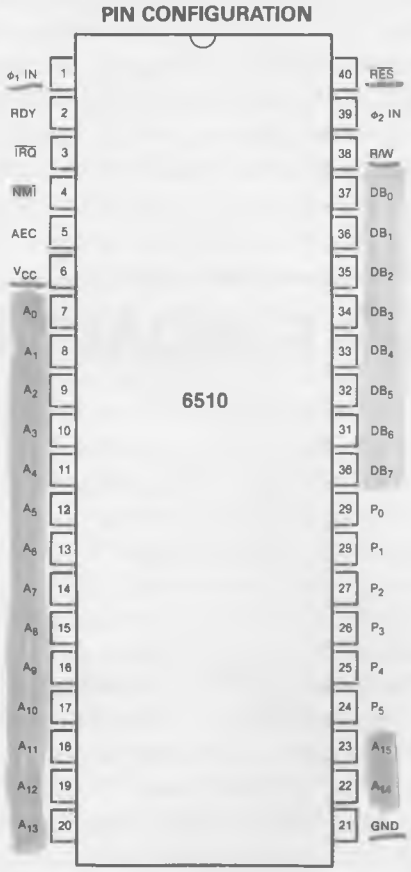
COMPUTER FUNDAMENTALS

The Commodore 64 is both a technological and an economical triumph. Because of the current fierce competition in the personal computer market, Commodore has gone all out with the 64 to provide the most sophisticated computing package yet seen in an 8-bit machine—and all for a cost of around $200 at the time of this writing. The low price is, in large part, a direct result of several very sophisticated, large-scale integrated circuits that were specially designed for the Commodore 64 and that greatly reduce the number of components inside the case. Like most of the bargain-basement computers on the market, the Commodore 64 has BASIC language capabilities and a provision to store programs on cassette tapes. In addition, the Commodore 64 can be interfaced to a wide variety of peripheral devices, such as modems, disk drives, and printers. In fact, there is so much interfacing circuitry inside the Commodore 64 already that most of the projects described in this book will require only a bare minimum of additional components and will rely heavily on optimal use of the components already in the Commodore 64's deceptively small case.

The 6510 Microprocessor

The heart of the Commodore 64 is the MOS Technology 6510 microprocessor (see Figure 1–1). This is a single integrated circuit, about the

FIGURE 1–1
Pin assignments for the 6510 microprocessor.



size of a stick of chewing gum, that contains all the circuitry required for a computer's central processing unit. The 6510 is a very popular microprocessor and is extremely powerful. The 6510 does several important functions in the Commodore 64. First, it can perform arithmetic and logical operations internally. It can also direct data to and from memory that is external to the 6510. Finally, and perhaps most importantly, it can fetch and execute a sequence of commands previously stored in memory. We refer to this sequence as *the program.*

Binary Bytes and Bits

The 6510 microprocessor does all its arithmetic in the binary or base 2 number system. Because the binary system uses only two numbers, 1s and 0s, it is easy to implement in digital electronics. The 1s represent the

high state, +5 volts in the 6510, and 0 is represented by a low state, 0 volts. In order to express numbers greater than 0 or 1, the computer carries the data as a multiple-digit binary number that we refer to as a *word*. Obviously, each digit in the computer must be represented by a discrete electrical circuit. The 6510 has eight such circuits for data manipulation and therefore is said to have an 8-bit word length, each bit standing for a binary digit. If you will recall, your school mathematics course taught that each successive digit in any number system represented the base raised to a successively higher power. The following table shows how bit 0 represents 2 to the 0 power, or 1, and is thus the 1's digit. Bit 1 is 2 to the first power, or 2. Bit 1 thus becomes the 2's digit. Bit 2 becomes 2 to the second power, or the 4's digit, and so on to bit 7. Bit 7 is 2 to the seventh power, or the 128's digit.

Power of 2	7	6	5	4	3	2	1	0
Result	128	64	32	15	8	4	2	1
Bit # in the 6510	7	6	5	4	3	2	1	0

Thus, the 8-bit number 10010110 represents one 128, no 64s, no 32s, one 16, no 8s, one 4, one 2, and no 1s. The sum of the above is 150. Thus 10010110 in binary is equivalent to 150 in decimal (base 10) notation. One obvious problem with an 8-bit data word is that 255 (decimal) is the largest number that can be represented. Larger numbers require more binary bits. Clearly, your Commodore 64 computer can work with numbers larger than 255. It simply does this internally by representing the number with several 8-bit words. Great care was taken in programming the BASIC software so that BASIC could keep track of which word represents which part of the number. Most large computers use a 16-bit rather than an 8-bit word length, which makes such programming much easier. These 16-bit computers still sell for thousands of dollars so that 16-sets of logic circuits do not present much of a problem in the big machines. The original microprocessors, however, were to be sold for under $20, and thus, the compromise was made to limit the number of bits to only 8.

The 8-bit word length is often referred to as a byte. The IBM Corporation introduced the term byte to refer to one-half of its 16-bit word because it had several instructions that would manipulate just 8 bits at a time. Because many of the early microprocessor users had trained on the big 16-bit machines, they referred to the 8-bit word as a byte. The name has stuck, so that even though 8 bits represent a full word in the 6510, those 8 bits are usually called a byte rather than a word in microprocessor jargon.

Computer Anatomy

The Commodore 64 computer, like any computer, consists of three basic parts: the central processor unit, the memory, and the input/output (often abbreviated I/O) devices. The central processor unit is the 6510 chip, which we have already discussed. Now let's examine the other two components.

Memory

Memory in your Commodore 64 computer is important because it is the place where both the program and data are stored. Memory in the Commodore 64 is organized into a series of addresses into which the computer can deposit or from which it can retrieve 8 bits of information at a time. The 6510 selects a specific memory location by placing the binary code of that location on the 16 address lines that emanate from the 6510 chip. Since there are 16 such lines, 2^{16} or 65,536 individual locations can be uniquely addressed.

Memory is divided into two types—read/write (often called RAM, which stands for random access memory) and read-only memory, or ROM. The former can either be written to or read from. RAM is used in your computer to store temporary data. When you enter a BASIC program, it is held in RAM. As long as you do not turn off the power, the computer will be able to remember your program. This brings up an important point about RAM. One of the technological developments that made personal microcomputers, if not possible, at least affordable, was the introduction of semiconductor memory. Originally, computers used expensive core-type memory. Core memory operated on the principle of magnetizing small ferrite rings called cores. Once magnetized, a core would stay magnetized even if the power was turned off. That, however, is not true of modern semiconductor memory. All data is lost when the power is interrupted, even if only briefly. A computer can operate only if a program is resident in memory. In the core-based computers, a startup program called a "bootstrap" was entered through switches on a control panel. Once entered, the bootstrap would then be present in memory every time the computer was turned on. When the first microcomputers, such as the Altair, appeared on the market, they had front panels as well. Because they used semiconductor memory, it was necessary to key in the bootstrap every time the system was fired up. This inconvenience was soon overcome by the rapid development of read-only memories, or ROMs as they are better known.

ROMs are programmed once and retain their data thereafter. Although the computer cannot deposit data in a ROM, it can execute a program that is stored in a ROM. By placing the bootstrap program in an inexpensive

ROM, it was no longer necessary to key it in through the control panel. Not only was this a timesaver, but it proved to be economical as well because the expensive control panel circuitry became unnecessary. Today, the control panel for most microcomputers consists of a single off/on switch. Your Commodore 64 computer comes with 16,384 words of ROM containing a bootstrap operating system called the Kernel, and the BASIC language interpreter. When the 6510 is first started, it automatically starts executing the program beginning at the highest address of memory. This is where the ROM is located. The bootstrap program first does a memory test to find out how much RAM is present in the system. Then it starts the BASIC interpreter.

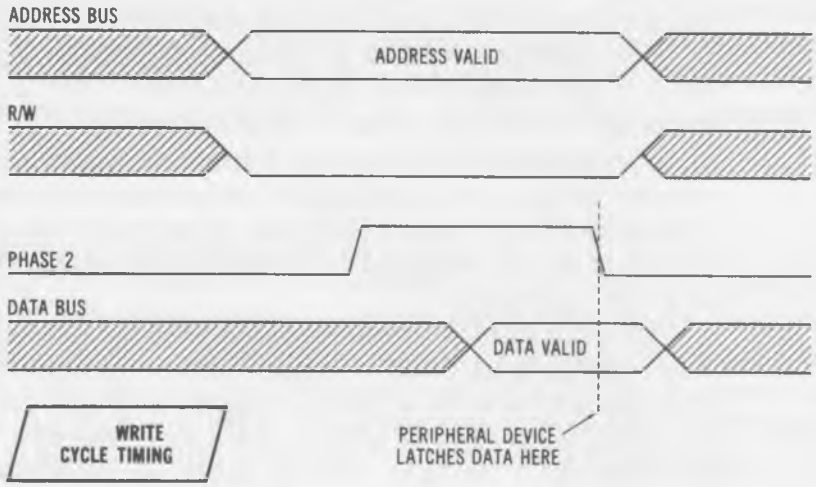
Finally, it should be mentioned that memory size is usually expressed in Ks. One K (for the Greek word *kilo*, meaning one thousand), in computer jargon refers to 1024. This is 2^{10} and a convenient unit of memory size. Thus, when someone refers to the Commodore 64's 64K RAM, they mean that it has 1024×64, or 65536, words of random access memory.

I/O Devices

The third part of the computer that you should be aware of is the I/O (input/output) section. Regardless of how powerful a computer may be, it is useful only if it has some means of communicating with the external world. The Commodore 64 can communicate with us via the TV screen and the keyboard. The TV screen is an output device. Similarly, the keyboard is an input device. Your Commodore 64 is not limited, however, to these forms of communication. For example, it can also communicate with a tape recorder or a disk to save data and programs. Many other forms of communication are possible with your Commodore 64 computer as well. It can turn lights off and on in your house, receive or send Morse code over a ham radio set, measure the temperature in your garden, and a variety of other tasks—if the proper I/O devices are incorporated into the computer. The technique by which such devices are incorporated is called *interfacing*. Interfacing principles for the Commodore 64 are actually quite simple and easily grasped.

The way an I/O device is interfaced to the 6510 is deceptively simple. All I/O devices are interfaced as if they were memory. The actual sequence of events that occurs when the 6510 accesses memory involves four sets of control lines that emanate from the 6510 chip—the read-write line (R/W), the phase-two clock, the 16 address lines (AO-A15), and the 8 data lines (DO-D7). If the 6510 is to write data to memory location 40960, the following sequence of events takes place (see Figure 1-2). The bit

FIGURE 1–2
Write-cycle timing for the 6510.



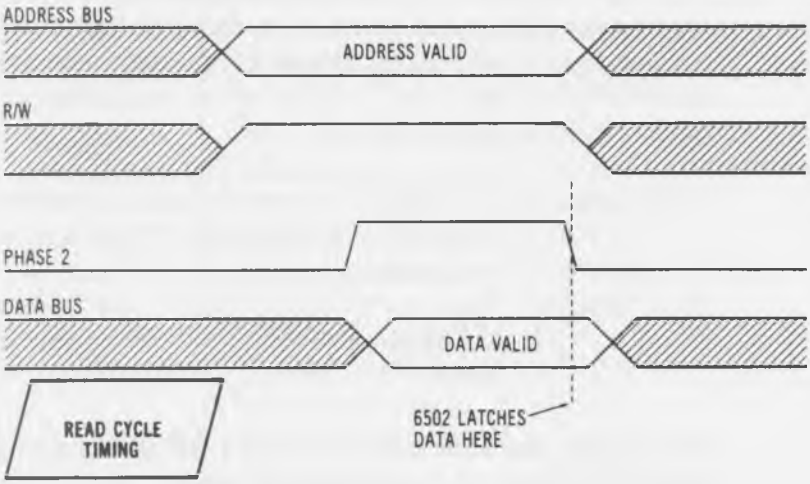
pattern for 40960, 1010000000000000, appears on the 16 address lines. Logic in the memory circuit recognizes that address pattern and alerts that location that it is being queried. Second, the R/W line assumes a low logic state of 0 volts, indicating that a write is about to take place. Finally, the phase-two line goes high for half a clock cycle. During that time the binary representation of the 8-bit word to be transferred is placed on the 8 data lines. As phase two returns to the low state, that transition signals the memory or I/O circuit to take that pattern off the data lines and hold it.

A read cycle is very similar except that the R/W line will be in a high state, indicating that the flow of information will now be from the memory (or an I/O device) to the 6510. When phase two goes high in a read cycle, the memory device must place the 8 bits of data on the data lines and the 6510 must read and hold the data as it briefly appears. See Figure 1–3 for a description of the read cycle.

To interface an I/O device to the 6510, you must find a memory address that is not occupied by RAM or ROM and provide the logic circuitry to recognize when the address appears. Memory addresses 38912 to 40959 are reserved for I/O devices in the Commodore 64. If the device is an output device, such as the printer port, it is designed to capture data on a write cycle. If it is an input device, such as the keyboard, it must be wired to transfer data on a read cycle. The Commodore 64's BASIC has two important commands, PEEK and POKE, that can access memory. Thus, if an I/O device is interfaced into the Commodore 64 as if it were a memory location, you will be able to send data to it with the POKE command and read data from it with the PEEK command.

Although you could add an almost infinite variety of I/O devices to

FIGURE 1–3
Read-cycle timing for the 6510.



the Commodore 64's memory space (over 512 locations are still vacant), two devices already present in the Commodore 64 will meet most of your interfacing needs. These are the two 6526 CIA chips, which are connected to the Commodore 64's user port. This book will show you how to fully use these two devices.

Memory Management in the Commodore 64

Several paragraphs above we stated that the 6510 microprocessor has a 16-line address bus and thus could address 2^{16} or 64K of memory. We also said that (1) all the Commodore 64's I/O devices were interfaced as memory locations, (2) that the system contains 64K of RAM, and (3) that it contains 16K of ROM. Obviously, this adds up to more than 64K of memory. How, then, can the Commodore 64 handle so much memory if its 6510 microprocessor has only 16 address lines? The answer is deceptively simple: several types of memory are installed in the same space, and logic circuits in the Commodore 64 allow the 6510 processor to select which memory type it wants to be active. This technique is called "bank-switching." The 6510 accomplishes the bank-switching through its special onboard 6-bit I/O port. The first two bits, bit 0 and bit 1, of this port contribute to the memory selection scheme. Two other signals, GAME and XROM, are also incorporated into the switching logic. These latter two signals are found on the expansion port connector. These lines are normally pulled to a high state by pull-up resistors but can be brought to a logical 0 by a plug-in cartridge.

The on-board I/O port is located at address 1 and has 6 bits that can

be programmed individually as inputs or outputs by starting a control byte at address 0. System startup puts the required control word in this direction register and it is not necessary to change it. Bits 2, 3, 4, and 5 of the data register (address 1) are used for the video and cassette I/O; we are interested in bits 0 and 1, which can be manipulated to bank-switch the RAM, ROM as follows:

Bit O LORAM	Normally 1. A zero switches out ROM and switches in RAM at $A000-BFFF.* This area is the BASIC interpreter.
Bit 1 HIRAM	Normally 1. A zero switches out ROM and switches in RAM at $E000-FFFF. This area is the kernel ROM.

You must of course be careful to ensure that the microprocessor has a program to run at the instant of bank switch.

Figure 1-4 shows the memory map for the Commodore 64 as it normally runs in BASIC. Bits 1 and 0 of address 1 are set to 11, respectively. A decimal 55 is put there by the initialization procedure if no cartridge is present. If address 1 is changed to have a 01 in those bits, both the BASIC ROM and the Kernel ROM are replaced with RAM. The 4K I/O block stays intact, however. A 10 takes out the BASIC ROM allowing 52K of contiguous RAM but preserving the I/O and the Commodore 64's operating system in the Kernel ROM. Finally, a 00 switches out all I/O and ROMs providing 64K of contiguous RAM. Pulling Ex ROM and/or GAME low can activate the ROMs in the plug-in cartridge at various locations in memory as well (see Chapter 8 for more details).

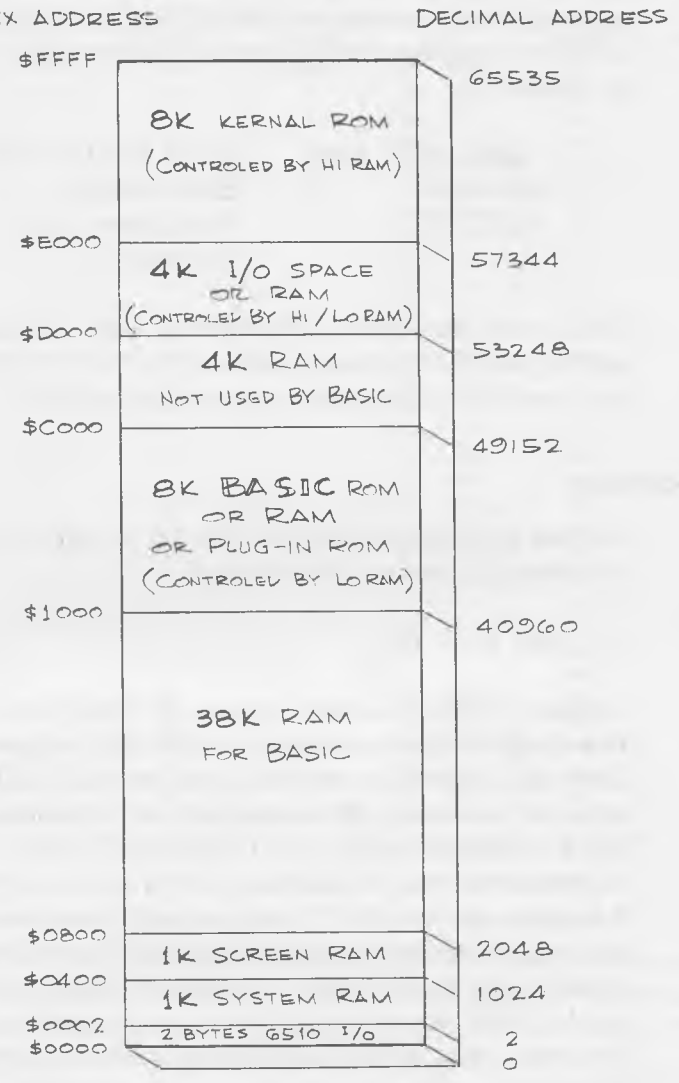
An interesting (and handy) feature of the Commodore 64 is that the write signal to the RAM that resides under the ROM is not disabled when the ROM is selected. Thus, for example, the BASIC ROM can be copied into RAM by the following simple program.

```
10 FOR I = 40960 to 49151
20 POKE I, PEEK (I)
30 NEXT I
40 POKE 1, 54
```

Lines 10–30 copy the ROM to RAM so that a copy of the BASIC ROM is at RAM addresses $A000-BFFF. The LORAM bit of the control port is reset to zero in line 40 to switch to BASIC in RAM. But nothing

*The $ indicates a hexidecimal number. Chapter 2 explains how to work with these numbers. Figure 1-2 gives memory addresses in both hexidecimal and decimal format.

FIGURE 1–4
Memory map for the Commodore 64.



seems to have changed. "READY" appears on the screen as before. Let's enter the following line in direct mode:

```
POKE 41853, 43
```

You will now see READY+ on the screen. We have actually modified the BASIC interpreter with this trivial demonstration program. The advanced programmer can modify any section of BASIC in a similar manner. Likewise, a custom kernel can be developed for addresses $E000-FFFF by an analogous process.

The movement of ROM contents to RAM by the BASIC program given earlier is very slow as compared with a machine-language transfer routine, which can move an entire ROM in less than a second. If you have a machine-language monitor, you can use the monitor's transfer function as follows:

```
T A000 BFFF A000    (Move ROM to RAM)
: 0001 36           (Bank-switch)
: A37D 2B           (Plus character)
X                   (To Basic)
```

The screen should show "READY+" after this sequence is entered, proving that you are running in RAM. RUN/STOP/RESTORE will return your machine to its normal memory configuration.

Data Formats

All data in the Commodore 64 must be handled as binary. Thus, when the computer receives the command

```
LET X = 10
```

a memory location actually receives the binary number 00001010. But how do you think the computer handles the *text* part of that command? Obviously, there is no way to express the word LET or an X in binary terms. It does this by representing each of the printable characters (referred to as alphanumerics) by a 1-byte code. Table 1–1 reveals that an A is represented by a 65 (decimal), a B by a 66, a C by a 67, and so on. It becomes obvious that I/O devices might want data in binary form, or they might want these alphanumeric codes, depending on the device. For example, you would want the interface for a game paddle to read a binary number that is proportional to the angle of the paddle. On the other hand, you would want to send alphanumeric character codes to a line printer. The two data types are handled separately in BASIC as numeric and string variables. Clearly, you must be careful when designing an interface to determine which type of data format you will be working with.

Virtually all computers now use a standard code for representing the alphanumeric characters. This code is referred to as the ASCII Code (American Standard Code for Information Interchange). There are thousands of peripheral devices available that communicate through this code. Because the Commodore 64 uses this code as well, all those devices are, at least in theory, compatible with your machine.

TABLE 1-1: The American Standard Code for Information Interchange (ASCII)

Binary	Decimal	Character	Binary	Decimal	Character
010 0000	32	SPACE	100 0000	64	@
010 0001	33	!	100 0001	65	A
010 0010	34	"	100 0010	66	B
010 0011	35	#	100 0011	67	C
010 0100	36	$	100 0100	68	D
010 0101	37	%	100 0101	69	E
010 0110	38	&	100 0110	70	F
010 0111	39	'	100 0111	71	G
010 1000	40	(	100 1000	72	H
010 1001	41	)	100 1001	73	I
010 1010	42	*	100 1010	74	J
010 1011	43	+	100 1011	75	K
010 1100	44	,	100 1100	76	L
010 1101	45	-	100 1101	77	M
010 1110	46	.	100 1110	78	N
010 1111	47	/	100 1111	79	O
011 0000	48	0	101 0000	80	P
011 0001	49	1	101 0001	81	Q
011 0010	50	2	101 0010	82	R
011 0011	51	3	101 0011	83	S
011 0100	52	4	101 0100	84	T
011 0101	53	5	101 0101	85	U
011 0110	54	6	101 0110	86	V
011 0111	55	7	101 0111	87	W
011 1000	56	8	101 1000	88	X
011 1001	57	9	101 1001	89	Y
011 1010	58	:	101 1010	90	Z
011 1011	59	;	101 1011	91	[
011 1100	60	<	101 1100	92	\
011 1101	61	=	101 1101	93	]
011 1110	62	>	101 1110	94	↑
011 1111	63	?	101 1111	95	—

Logic Chips

Today's computers are built with integrated circuits. An integrated circuit consists of many transistors, diodes, and resistors all together on one small chip of silicon, making up a complete electrical circuit. Today's technology permits thousands of individual components to be put on a single chip the size of the head of a pin. What is really amazing is that most of these integrated circuits, or ICs as they are more commonly called, range in cost from pennies to just a few dollars. Over the years, there has been a steady evolution of logic families and packaging in the IC industry. Logic families such as DTL (Diode Transistor Logic) and RTL (Resistor Transistor Logic) have come and gone, but in the past decade one basic family has emerged and dominated the industry. These are the TTL (Transistor-Transistor Logic) series of integrated circuits. The industry has also adopted a standard nomenclature and packaging system

in the 7400 series of logic chips. The numbering system and electrical properties in this family are the same for chips made by all of the various manufacturers. These ICs make perfect building blocks for the experimenter because most of the logic functions you will require can be found in this 7400 series. Furthermore, they are readily available in retail electronic outlets, which is a big help. Whenever possible, we will use 7400 series chips for the projects in this book.

Logic Ins and Outs

The Commodore 64 and the 7400 series of ICs both use the same logic system of a 1 represented by 2.6 to 5 volts and a 0 by 0 to .6 volts. All of these chips have two basic types of connections: inputs and outputs. An input has a high impedance and senses the level of the voltage applied to it. By contrast, an output has a very low impedance and can supply current in such a way to keep itself at either a logic 1 or 0 as dictated by its logic state. This arrangement allows the designer to have the output of one IC driving the input of another, so that the ICs can communicate with one another in the circuit. In general, a TTL output can drive five or more inputs before the ability of that output to supply current will be overly taxed. This is called "fan out." Although an output line may be tied to many inputs, it should never be tied to another output because there will be contention as to what the resulting level should be if the two outputs happen to disagree in their respective logic states. Such a test of wills will certainly result in ambiguous logic states and with some chips, such as high-power buffers, can even result in rather spectacular burnouts. There are two special exceptions to this rule. First, the "open-collector" devices may have multiple outputs tied together. These devices can only pull down the voltage to 0 and are incapable of pulling it up to +5. Thus, open-collector outputs must be pulled up to +5 volts by an external "pull-up" resistor to the system power. If any one of the outputs on that line goes to an active low, the entire line will be pulled down. In general, we will not be concerned with open-collector devices in this book, but you should be aware of them in your circuit design so that you do not try to use one of them as if it were a normal logic chip. The second exception is the three-state devices, which can have their outputs tied together as long as only one device is active at a time. We will not be concerned with them either in this book, but again, be aware that they exist.

Logic Functions

Digital logic is not difficult to understand, and you should be familiar with the basic AND and OR functions. The truth table below shows a simple AND function:

IN 1	*IN 2*	*OUT*
0	0	0
0	1	0
1	0	0
1	1	1

Note that a 1 will occur on the output only if both inputs IN 1 and IN 2 are in a high state. Contrast this to the OR function below:

IN 1	*IN 2*	*OUT*
0	0	0
0	1	1
1	0	1
1	1	1

In this case, the output will be a 1 if either IN 1 *OR* IN 2 is high. Note that for the 0 or inverse condition, the OR gate serves an AND function, that is, a 0 output will occur only if IN 1 is 0 *AND* IN 2 is 0. What is the inverse logic function of the AND Gate?

If the output of the gate is complemented, we put an N in front of the name. Thus, the truth table for a NAND gate would be:

IN 1	*IN 2*	*OUT*
0	0	1
0	1	1
1	0	1
1	1	0

Similarly for a NOR gate, the truth table would be as follows:

IN 1	*IN 2*	*OUT*
0	0	1
0	1	0
1	0	0
1	1	0

The AND and the OR functions are represented by special symbols as shown in Figure 1–5. Note that the NOR, or complement form, is indicated by an open circle on the lead that is complemented. The triangular symbols to the right of the figure show buffers and inverters. The buffers are useful in amplifying a signal so that it may fan out to more inputs. The complement form is useful when the signal must be inverted. Figure 1–6 shows the pin configuration of the 7400 quad NAND gate, the 7402 quad

FIGURE 1–5
Common logic symbols.

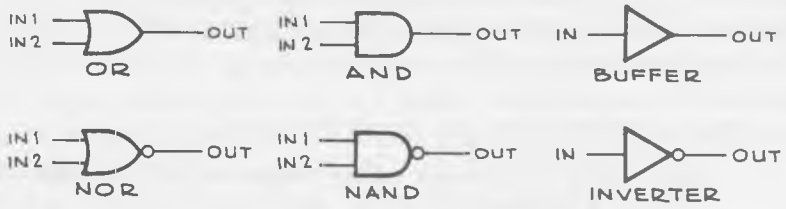
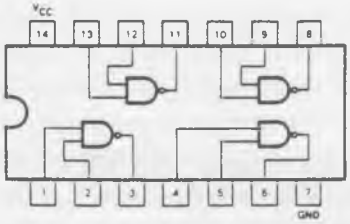
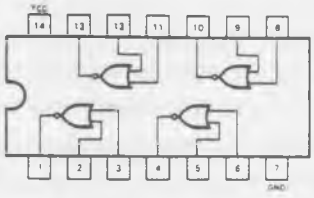


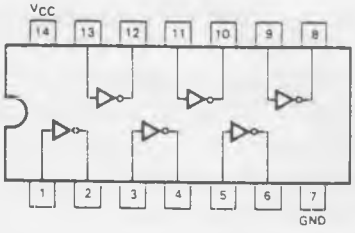
FIGURE 1–6
Pin assignments for 3 common TTL chips.



7400

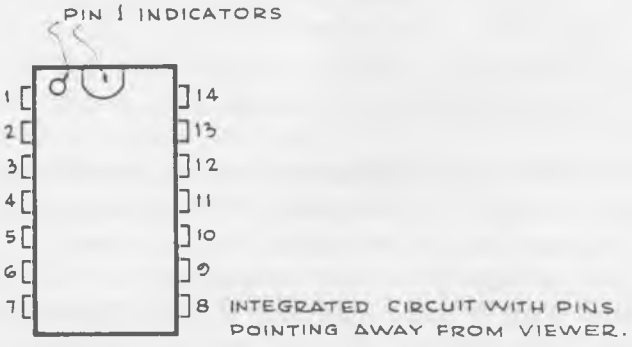


7402



7404

FIGURE 1–7
Pin numbering conventions for integrated circuits.



NOR gate and the 7404 hex buffer. Vcc refers to the +5 volt power connection, and GND is, of course, the system ground.

Finally, all the 7400 series ICs come in the familiar DIP (Dual In-line Package) package. Figure 1–7 shows that when the chip is viewed from the top (with the pins pointing away from you) and the pin 1 indicator up, pin 1 will be to the top left. The pin numbers are then in sequence going counterclockwise around the chip. There may be from 6 to 40 pins on the chip, but they will all use this convention for pin numbering.

2

USING MACHINE LANGUAGE IN THE COMMODORE 64

In Chapter 1 we learned that all I/O Devices in the C64 are interfaced as if they were memory. Furthermore, we found that these devices could be accessed by the commands PEEK and POKE from BASIC. For simplicity's sake, this book will try to use this approach to I/O programming whenever possible. Often, however, we have found that it is necessary to incorporate a machine language subroutine in conjunction with the BASIC program to fully use the capabilities of a particular device. At this point, you are probably asking, "What exactly is machine language?" As you may know, the 6510 (and its predecessor, the 6502) microprocessor is incapable of executing the BASIC language commands directly. These commands are much too complex. Rather, the 6510 is limited to a set of relatively simple commands that we refer to as machine code. We say simple because each machine language command usually results in a relatively simple operation. For example, one machine language command may cause 2 bytes of data to be added together, and another may cause a byte of data to be stored in memory. There are no commands as complicated as multiplication or division in the 6510. How, then, does the C64 perform the intricate commands available in BASIC? The answer is that each BASIC command causes the 6510 to perform a number of simple machine language commands in sequence until the required function is completed. A multiplication actually involves complicated combinations of additions and

shifts to arrive at the answer. This is all accomplished by a large machine language program in the C64 called the BASIC interpreter. As the 6510 executes the interpreter program, it examines each line of your BASIC program and performs all of the machine language steps to satisfy the indicated function. It then proceeds to the next BASIC statement.

The two main advantages of machine language programs are their speed and versatility. The BASIC interpreter is relatively slow and it would be impossible to control a device that required millisecond or faster response time. In machine language this is easy, however, because each 6510 machine-language instruction executes in seven microseconds or less. Also, machine language is much better suited for manipulating individual bits than is BASIC.

Many of the projects in this book rely on some machine-language software. It is not really necessary to become proficient in 6510 machine code in order to use these projects. All of the programs and explicit instructions on their use are provided in the text. We do feel, however, that it would be very instructive at this point to give a brief overview of 6510 machine code and to demonstrate how it can be implemented in the C64.

Registers within the 6510 (or 6502)

The 6510 microprocessor contains six important registers. A register is like a memory location in that a binary number can be moved in and out of it under program control. The accumulator is probably the most important register in the 6510. It is 8 bits wide, the same as a memory location, and that allows you to move data from the accumulator to memory and vice-versa. Also, the accumulator is the only register in which math can be performed. There are two other working registers in the 6510—the x and the y. Data can be transferred between any of these 8-bit registers and memory. Also, certain instructions exist that use these registers as the address for indirect data transfers.

The status register is quite different from the working registers described above. Most of its 8 bits contain information about what is happening inside the 6510. For example, one bit represents the "carry" from an addition operation.

The program counter is a special 16-bit register pointing to the address in memory where the next machine language instruction is to be found. It is normally incremented to the beginning of the next instruction in memory on completion of each instruction, unless that instruction was a branch or jump. In these cases it is changed to the address indicated by the branch or jump instruction.

The last register is the stack pointer. To implement subroutines and other functions, the 6510 must keep a first-in, last-out file of numbers.

This file is kept in memory, and the stack pointer leads to the next available space in memory for the file.

Fundamentals of Addressing: Hexadecimal Notation

The 6510 microprocessor has a group of 16 output lines, called address lines, numbered A0 through A15. The connections from these pins to all other devices make up the "address bus." The 6510, under instructions from the program it is running, activates memory and peripheral devices by putting a combination of "high" and "low" voltages on the address bus. A unique address, corresponding to one memory location or one device, might be written as:

LHLLLLLLLLLHHHHH

where L represents the low voltage (less than 0.6 volts) and H the high voltage (more than 2.6 volts), and the far left character is the voltage on the highest-order address wire, A15. Such a system for representing a unique address would be very unwieldy and difficult to use in written and oral communication and would be subject to errors. In several steps, we'll show how a "shorthand" system can be developed.

First, convert all the "L" values to zeros, and all the "H" values to ones, as follows:

0100000000011111

Since this is a number consisting of all ones and zeros, we call it a binary number. But we still do not have a suitable "shorthand" system.

Let's develop a table of the first 16 binary numbers, together with their decimal order and a single-digit arbitrary character to represent these values. Since we anticipate running out of single digits at the number 9, we will continue by using letters of the alphabet:

DECIMAL ORDER	*BINARY NUMBER*	*SINGLE-CHARACTER REPRESENTATION*
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8

9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Now, if we separate our long binary number into groups of four digits each, we have

0100 0000 0001 1111.

If we look up each group in the table and find its corresponding single-character representation, we have the number

401F

that can be written quickly and, more importantly, can be communicated with much less chance of error than the L/H or zero/one notation.

Likewise, groups of 8 wires having a combination of high and low voltages, such as the contents of a byte of memory, can be represented by two of our characters; for example,

LLHHHLHL
or 00111010

can be represented as 3A in our new notation.

This single-character system of representing a group of four highs and lows (4 bits of information) is the hexadecimal notation that is commonly used to describe binary numbers. Its name comes from the 16 characters (zero to F) used as its set of digits. In our everyday number system, we use 10 characters (0–9). A decimal number such as 6789 means:

9 ones
plus 8 tens
plus 7 hundreds
plus 6 thousands

Moving left one digit increases its "weight" by 10. Our hexadecimal, or hex, number has a similar weighting of each digit, although the weighting is by a factor of 16 rather than 10. Each digit has a weight 16 times the weight of the digit to its right.

With this information, we can convert our hex number $401F to a

number in more familiar decimal notation. (Note that from now on we will distinguish hex numbers by the dollar sign:)

$F	15 ones =	15
$1	1 16's =	16
$0	0 256's =	0
$4	4 4096's =	16384
	total	16415

You will need to use this conversion of a hex number to decimal in your interfacing projects. While addresses are more easily understood in the hex notation, the C64's BASIC must be given numbers in decimal for the PEEK and POKE commands. If you anticipate any serious machine language programming in the C64, we strongly urge you to get Commodore's monitor (available on either disk or cassette) that accepts addresses in hexadecimal notation. It also contains an assembler and a disassembler that greatly facilitates programming.

The Format of Machine Code

Each instruction for the 6510 is represented by a 1-byte hexadecimal operation code and by a three-letter mnemonic name. Table 2–1 presents a list

TABLE 2-1A: Alphabetic Listing of Opcodes

ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y
DEC	Decrement Memory by One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One

TABLE 2-1A: Alphabetic Listing of Opcodes (Continued)

EOR	"Exclusive-Or" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

TABLE 2-1B: Hex Listing of Opcodes

```
00 - BRK                    58 - CLI                    AD - LDA - Absolute
01 - ORA - (Indirect, X)    59 - EOR - Absolute, Y      AE - LDX - Absolute
05 - ORA - Zero Page        5D - EOR - Absolute, X      B0 - BCS
06 - ALS - Zero Page        5E - LSR - Absolute, X      B1 - LDA - (Indirect, Y)
08 - PHP                    60 - RTS                    B4 - LDY - Zero Page, X
09 - ORA - Immediate        61 - ADC - (Indirect, X)    B5 - LDA - Zero Page, X
0A - ASL - Accumulator      65 - ADC - Zero Page        B6 - LDX - Zero Page, Y
0D - ORA - Absolute         66 - ROR - Zero Page        B8 - CLV
0E - ASL - Absolute         68 - PLA                    B9 - LDA - Absolute, Y
10 - BPL                    69 - ADC - Immediate        BA - TSX
11 - ORA - (Indirect, Y)    6A - ROR - Accumulator      BC - LDY - Absolute, X
15 - ORA - Zero Page, X     6C - JMP - Indirect         BD - LDA - Absolute, X
16 - ASL - Zero Page, X     6D - ADC - Absolute         BE - LDX - Absolute, Y
18 - CLC                    6E - ROR - Absolute         C0 - CPY - Immediate
19 - ORA - Absolute, Y      70 - BVS                    C1 - CMP - (Indirect. X)
```

TABLE 2-1B: Hex Listing of Opcodes (Continued)

```
1D - ORA - Absolute, X
1E - ASL - Absolute, X
20 - JSR
21 - AND - (Indirect, X)
24 - BIT - Zero Page
25 - AND - Zero Page
26 - ROL - Zero Page
28 - PLP
29 - AND - Immediate
2A - ROL - Accumulator
2C - BIT - Absolute
2D - AND - Absolute
2E - ROL - Absolute
30 - BMI
31 - AND - (Indirect, Y)
35 - AND - Zero Page, X
36 - ROL - Zero Page, X
38 - SEC
39 - AND - Absolute, X
3D - AND - Absolute, X
3E - ROL - Absolute, X
40 - RTI
41 - EOR - (Indirect, X)
45 - EOR - Zero Page
46 - LSR - Zero Page
48 - PHA
49 - EOR - Immediate
4A - LSR - Accumulator
4C - JMP - Absolute
4D - EOR - Immediate
4E - LSR - Absolute
50 - BVC
51 - EOR (Indirect, Y)
55 - EOR - Zero Page, X
56 - LSR - Zero Page, X
71 - ADC - (Indirect, Y)
75 - ADC - Zero Page, X
76 - ROR - Zero Page, X
78 - SEI
79 - ADC - Absolute, X NOP
7D - ADC - Absolute, X NOP
7E - ROR - Absolute, X NOP
81 - STA - (Indirect, X)
84 - STY - Zero Page
85 - STA - Zero Page
86 - STX - Zero Page
88 - DEY
8A - TXA
8C - STY - Absolute
8D - STA - Absolute
8E - STX - Absolute
90 - BCC
91 - STA - (Indirect, Y)
94 - STY - Zero Page, X
95 - STA - Zero Page, X
96 - STX - Zero Page, Y
98 - TYA
99 - STA - Absolute, Y
9A - TXS
9D - STA - Absolute, X
A0 - LDY - Immediate
A1 - LDA - (Indirect, X)
A2 - LDX - Immediate
A4 - LDY - Zero Page
A5 - LDA - Zero Page
A6 - LDX - Zero Page
A8 - Tay
A9 - LDA - Immediate
AA - TAX
AC - LDY - Absolute
C4 - CPY - Zero Page
C5 - CMP - Zero Page
C6 - DEC - Zero Page
C8 - INY
C9 - CMP - Immediate
CA - DEX
CC - CPY - Absolute
CD - CMP - Absolute
CE - DEC - Absolute
D0 - BNE
D1 - CMP - (Indirect, Y)
D5 - CMP - Zero Page, X
D6 - DEC - Zero Page, X
D8 - CLD
D9 - CMP - Absolute, Y
DD - CMP - Absolute, X
DE - DEC - Absolute, X
E0 - CPX - Immediate
E1 - SBC - (Incirect, X)
E4 - CPX - Zero Page
E5 - SBC - Zero Page
E6 - INC - Zero Page
E8 - INX
E9 - SBC - Immediate
EA - NOP
EC - CPX - Absolute
ED - SBC - Absolute
EE - INC - Absolute
F0 - BEQ
F1 - SBC - (Indirect, Y)
F5 - SBC - Zero Page, X
F6 - INC - Zero Page, X
F8 - SED
F9 - SBC - Absolute, Y
FD - SBC - Absolute, X
FE - INC - Absolute, X
```

of the 6510 instruction set. Let's examine the first of these, ADC. ADC stands for add memory to accumulator with carry. This means that a byte from memory will be added to the contents of the accumulator and the carry register. If there is an overflow, that is, if the result exceeds $FF (255 decimal), then a special bit, the carry, will be set to a 1. The carry prevents any loss of data. Note that many 6510 instructions have eight different addressing modes. This refers to the location in memory where the number to be added to the accumulator is to be found. For example, the operation code $69 is an ADC instruction in the immediate mode and refers to the next byte in memory following the ADC instruction. This is a 2-byte instruction because 1-byte is needed for the operation code and a second (called the operand) for the byte to be added to the accumulator. When the 6510 sees the $69 instruction, it knows that this is a 2-byte instruction and increments the program counter register by 2 before it executes the next instruction. On the other hand, a $6D indicates that an absolute address mode is to be used, and 3 bytes will be required for the instruction. The first byte will be the $6D operation code followed by the

operand, with the address of the memory location to be added. Since 16 bits are needed to define an address, 2 bytes are required for the operand. The first byte is the low order 8 bits of the address, and the second byte represents the high order 8 bits of the address. The same structure applies to the other 6510 commands. A detailed description of all of the 6510 commands and addressing modes is obviously beyond the scope of this book. However, if you wish to learn more about 6510 and 6502 machine language, we recommend *Apple II-6502 Assembly Language Tutor* by Richard Haskell (Prentice-Hall, 1983, $34.95).

A machine language program is usually presented in the following format, called an assembled listing:

```
1C00    18        CLC
1C01    AD101C    LDA $1C10
1C04    6D111C    ADC $1C11
1C07    AA        TAX
1C08    A9 00     LDA #$00
1C0A    00        BRK
```

The left-hand column gives the address where the instruction is stored, the middle column gives the contents of those addresses, and the right-hand column lists the mnemonic. Step 1 clears the carry flag while the next instruction causes the 6510 to load the contents of memory location $1C10 into the accumulator. Next, it adds the contents of location $1C11 to the accumulator. The third instruction moves that sum to the X register. Note that TAX is a 1-byte instruction. The accumulator is then cleared by loading $00 into it (the # means an immediate mode instruction). Finally, a software interrupt is forced with the BRK command to end the program. All of the machine language routines in this book will be presented in this simple format.

The BASIC SYS command

Once a machine language program resides in memory, it can be accessed from BASIC by the command SYS. A SYS command causes the 6510 to start executing machine code at the decimal address following the SYS command. The 6510 will continue to execute machine code until a RTS (return from subroutine) instruction is encountered. At that point, the 6510 returns to BASIC at the instruction following the SYS command.

Consider the following program:

```
1C00    18        CLC
1C01    AD101C    LDA $1C10
```

```
1C04    6D111C    ADC $1C11
1C07    8D121C    STA $1C12
1CA0    60        RTS
```

This program adds the two numbers stored at $1C10 and $1C11. The result is stored at $1C12. To run this example in your Commodore 64, you must first convert the hexadecimal codes to decimal, as shown in the table below.

HEXADECIMAL	*DECIMAL*
18	24
AD	173
10	16
1C	28
6D	109
11	17
1C	28
8D	141
12	18
1C	28
60	96
1C00	7168 (starting address)
1C10	7184 (number 1)
1C11	7185 (number 2)
1C12	7186 (sum)

This program can now be POKEd into memory by the following BASIC program:

```
10   DATA 24, 173, 16, 28, 109, 17, 28, 141, 18, 28, 96
20   FOR I = 0 to 10
30   READ X
40   POKE 7168 + I, X
50   NEXT I
60   REM PROGRAM IS NOW IN MEMORY
70   PRINT "NUMBER 1, NUMBER 2":
80   INPUT X1, X2
90   POKE 7184, X1
100  POKE 7185, X2
110  SYS 7168
120  PRINT "THEIR SUM IS", PEEK (7287)
```

Enter this program and run it. Be sure that you enter integer numbers, each less than 255. Note, that if their sum exceeds 255, the result will

be 256 less than the correct answer because no provision for testing the carry bit was incorporated into this program.

Locating Free Space for Machine Code

There are several places in memory where machine language code can be located. In the previous example, the code was inserted at $1C00. Since the BASIC program was very small, we knew that it would not extend into this area. If the BASIC program were much larger, however, it could compete for this space. Also, any use of string variables could cause high memory to be overwritten with string information. Addresses 828 to 1024 (decimal) contain a free space reserved for temporary storage of data from the cassette recorder. These 196 bytes can be used to hold a machine language program as long as tape operations are avoided while the machine language routine is operative. Any tape read or write will overwrite this area and destroy the program. The best place to put machine language programs is the 4K block of RAM from $C000 to $CFFF. This block of memory is unused by BASIC or by the Commodore 64's operating system.

Using the 60 Hz Interrupt Vector

One particularly useful feature of the Commodore 64 is the way it senses key presses. Every 60th of a second the 6510 is interrupted. On an interrupt, BASIC calculations are stopped, and the 6510 jumps to a program that scans the keyboard to see if a key has been depressed. This routine is executed in a very short period of time, after that it returns to BASIC. The address of the interrupt service routine is held in memory locations $0314 and $0315. The interrupt address can be changed to point to a user-written subroutine that after completion can pass control to the keyboard scan routine at $EA31. This will give a program the appearance of running continuously in the background as BASIC continues to execute normally. There are several rules that must be observed for this mode of operation:

1. The routine must be short. Only a millisecond or two can be spent in the program because control must return to BASIC well before the next interrupt occurs.
2. The program must jump to the Commodore 64's keyboard scan routine with all registers exactly as they were when your program was entered. This is best done by pushing the accumulator, the x, the y, and the status registers on the stack at the start of the program

and pulling them off the stack in reverse order just before jumping to $EA31, the keyboard scan routine.

3. You must change both addresses $314 and $315 simultaneously from a machine language routine. BASIC is too slow to accomplish this with two POKES. An interrupt is almost certain to occur between the POKES so that only half of the address will have been changed. Control will thus be transferred to a nonvalid address, and a system crash will result.

An example of this approach appears in Listing 2–1. This simple

LISTING 2–1
Interrupt vector routes

```
BGRND......PAGE 0001

LINE# LOC   CODE        LINE


00001  0000             ; DEMONSTRATE 60HZ INTERUPT
00002  0000             *=$C000
00003  C000             ;-------------------------
00004  C000             ; EQUATES
00005  C000             IRQVEC=$0314            ;IRQ VECTOR
00006  C000             OLDIRQ=$EA31            ;OLD IRQ ADDRESS
00007  C000             ;-------------------------
00008  C000             ; THIS PART CHANGES THE INTERUPT
00009  C000             ; VECTOR FROM $EA31 TO $C014
00010  C000  08         SINT    PHP             ;SAVE STATUS
00011  C001  78                 SEI             ;INHIBIT IRQ
00012  C002  A9 13              LDA #$13        ;LOW BYTE OF START
00013  C004  8D 14 03           STA IRQVEC
00014  C007  A9 C0              LDA #$C0        ;HI BYTE OF START
00015  C009  8D 15 03           STA IRQVEC+1
00016  C00C  A9 20              LDA #$20
00017  C00E  85 FF              STA $FF         ;INNITIALIZE COUNTER
00018  C010  28                 PLP
00019  C011  58                 CLI             ;ALLOW INTERUPTS
00020  C012  60                 RTS
00021  C013             ;-------------------------
00022  C013             ; THIS PART IS EXECUTED EACH INTERUPT
00023  C013  48         START   PHA             ;SAVE AC
00024  C014  C6 FF              DEC $FF         ;$FF IS JIFFIE COUNTER
00025  C016  D0 06              BNE NDONE       ;33 JIFFIES YET?
00026  C018  A9 22              LDA #$22        ;YES RESET $FF
00027  C01A  85 FF              STA $FF
00028  C01C  C6 FE              DEC $FE         ;CHANGE CHAR EACH 1/2 SECOND
00029  C01E  A5 FE        NDONE  LDA $FE         ;STORE $FE ON THE
00030  C020  8D 27 04           STA $0427       ;CORNER OF SCREEN
00031  C023  68                 PLA             ;RESTORE AC
00032  C024  4C 31 EA           JMP OLDIRQ
00033  C027             .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 IRQVEC   0314    NDONE    C01E    OLDIRQ   EA31    SINT     C000
 START    C013

END OF ASSEMBLY
```

program puts a character in the upper right corner of the screen and twice a second it changes it so that it will step through the entire Commodore 64's character set. The program, of course, does nothing useful except demonstrate that the Commodore 64 can easily do two jobs at once. Lines 10 through 20 change the IRQ Vector. The interrupt service program is in lines 23 through 33. Since interrupts occur 60 times a second, 34 interrupts are counted in $FF before the character code in $FE is changed. The BNE instruction in line 25 determines whether or not $FF has reached zero, indicating that the 34 interrupts have occurred.

Listing 2–2 is a BASIC program that has the decimal equivalent of the program stored in data statements. Enter and run this program to start the new interrupt service routine. Notice that our background task continues regardless of whether we list the program, edit the program, re-RUN the program, or even erase it and enter a new program. Only when the Commodore 64 is reset with a RUN/STOP/RESET will it stop incrementing the display.

In the following chapters, we will illustrate some useful programs that take advantage of the Commodore 64's unique interrupt structure. For example, Chapter 5 will show you how to generate a refresh pulse for a mechanical servo. The generation of these pulses is completely transparent to BASIC, and again it will appear that your Commodore 64 is doing two jobs at once.

LISTING 2–2
Basic version of 2–1

```
0 REM ****** 60HZ INTERUPT DEMO *******
1 DATA 8 , 120 , 169 , 19 , 141
5 DATA 20 , 3 , 169 , 192 , 141
10 DATA 21 , 3 , 169 , 32 , 133
15 DATA 255 , 40 , 88 , 96 , 72
20 DATA 198 , 255 , 208 , 6 , 169
25 DATA 34 , 133 , 255 , 198 , 254
30 DATA 165 , 254 , 141 , 39 , 4
35 DATA 104 , 76 , 49 , 234
100 FOR I=0TO38
110 READ X
120 POKE 49152+I,X:REM POKE INTO  RAM
130 NEXT I
140 SYS 49152 : REM CHANGE VECTOR
```

3

THE 6526 CIA

The 6526 Complex Interface Adapter is a very flexible chip. Within the 6526 IC there are two parallel, 8-bit, latching input/output ports, two 16-bit interval timers, a real-time clock, and a serial shift register. The Commodore 64 has two 6526's and uses them to control and/or communicate with the keyboard, cassette, joy stick, printer, and diskette. The 6526 comes as a 40-pin integrated circuit which is primarily intended as a multifunction companion chip for the 6510 microprocessor. The CIAs greatly reduce the parts count and cost of the Commodore 64 computer. It is an inexpensive building block and, as it turns out, some of its functions can even operate simultaneously.

Many lines from these 6526s are connected to the user port and it is this facility we wish to present in detail in this chapter. We will concentrate mainly on digital I/O and timer applications.

As shown in Figure 3–1, all of port B and bits 2 and 3 of port A are brought out through the general purpose user port. Also, the FLAG and PC lines from one 6526 are brought out, as well as the SP and CNT lines from both 6526's.

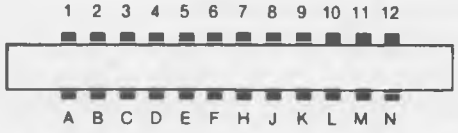
Each 6526 occupies 16 adjacent address locations in the Commodore 64's memory (see Table 3-1). These registers allow the programmer to input and output data in either parallel or serialized form, set and read the timers, read various 6526 status flags, and control various MPU in-

FIGURE 3–1
Signals on the 6526 CIA chip and their pin assignments on the Commodor 64's user port.

User I/O

Pin	Type	Note
1	GND	
2	+5V	MAX. 100 mA
3	RESET	
4	CNT1	
5	SP1	
6	CNT2	
7	SP2	
8	PC2	
9	SER. ATN IN	
10	9 VAC	MAX. 100 mA
11	9 VAC	MAX. 100 mA
12	GND	

Pin	Type	Note
A	GND	
B	FLAG2	
C	PB0	
D	PB1	
E	PB2	
F	PB3	
H	PB4	
J	PB5	
K	PB6	
L	PB7	
M	PA2	
N	GND	



terrupting features. The remainder of this chapter will explain and present examples, with experiments, of the various 6526 functions. The interface projects in the following chapters utilize these functions in specific applications.

The Logic Probe

The logic probe is an invaluable tool for troubleshooting digital circuits. If you are going to build any of the projects in this book, we urge you to either buy a logic probe or build the one described in Figure 3–2. A logic probe has three indicator lamps. When the probe is placed on a line, the

TABLE 3.1: Memory Addresses for CIA #2's Registers

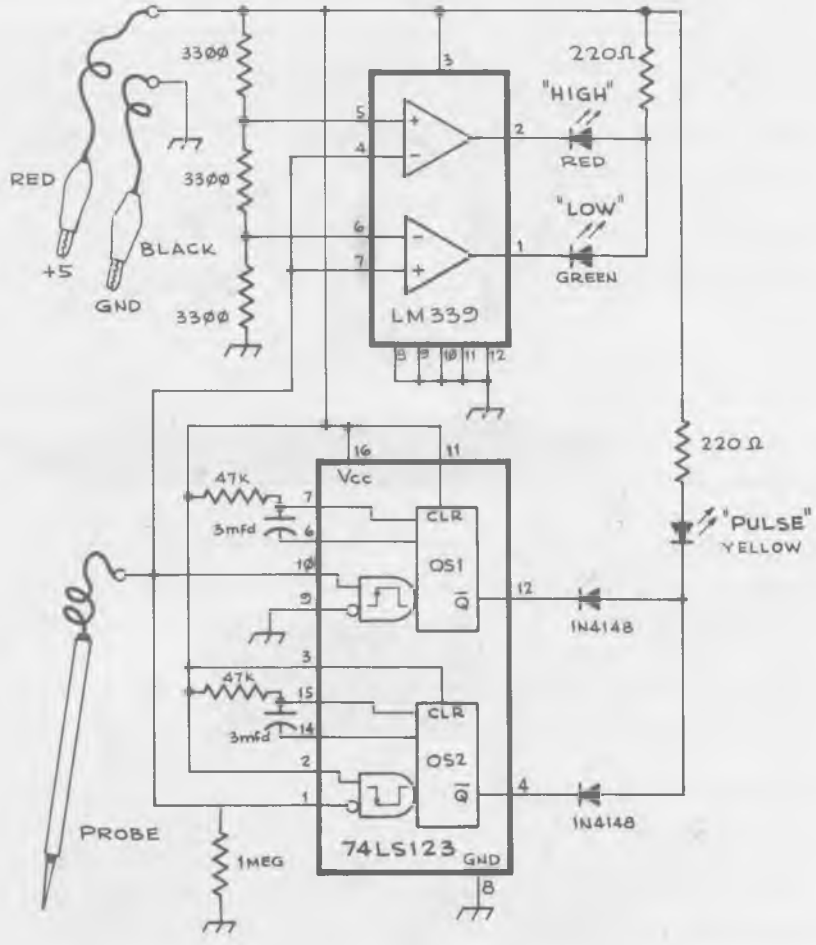
DECIMAL	*HEX*	
56576	$DD00	Peripheral data register A
56577	DD01	Peripheral data register B
56578	DD02	Data direction A
56579	DD03	Data direction B
56580	DD04	Timer A low byte
56581	DD05	Timer A high byte
56582	DD06	Timer B low byte
56583	DD07	Timer B high byte
56584	DD08	10ths of a second register
56585	DD09	Seconds register
56586	DD0A	Minutes register
56587	DD0B	Hours + AM/PM register
56588	DD0C	Serial data register
56589	DD0D	Interrupt control register
56590	DD0E	Control register A
56591	DD0F	Control register B

CIA #1 has the same sequence but starts at 56320 ($DC00).

indicators HIGH, LOW, or PULSE light up respectively when the line is in a logic high, a logic low, or is in a transition from one state to another. A fourth state is indicated when none of the indicators are lit. This state results when the probe is at an intermediate voltage and either indicates an unconnected wire or the real logic state known as Tri-State. In general, the logic probe has replaced the multimeter as the basic electronics tool for the digital troubleshooter.

Figure 3–2 shows the schematic of a simple logic probe. It can be easily built on a 2-inch by 4-inch piece of perfboard. Wire-wrap construction as shown in Figure 3–3 should be used. Attach a 12-inch piece of 22-gauge insulated wire with a small alligator clip on the Vcc and GND pins. Use a red wire for Vcc and a black one for GND. These wires are used to pick up power from the system under test. A third 12-inch wire should be used for the test lead and should terminate in a pointed probe like the one used on a voltmeter. After the circuit is completed, attach the red lead to a +5 volt source and the black one to ground. None of the lights should be on. Touch the probe to the red clip (Vcc); you should see the HIGH lamp go on. Now touch the probe to the black clip (GND). The LOW lamp should light, and the PULSE lamp should flash momentarily, indicating that a state change has occurred. The PULSE lamp should flash on either a high-to-low or a low-to-high transition. If no LEDs light, check to see if the LEDs have been inserted backward; also check the polarity of the electrolytic capacitors and look for missing wires in your wire-wrap connections. Figure 3–3 shows the finished logic probe.

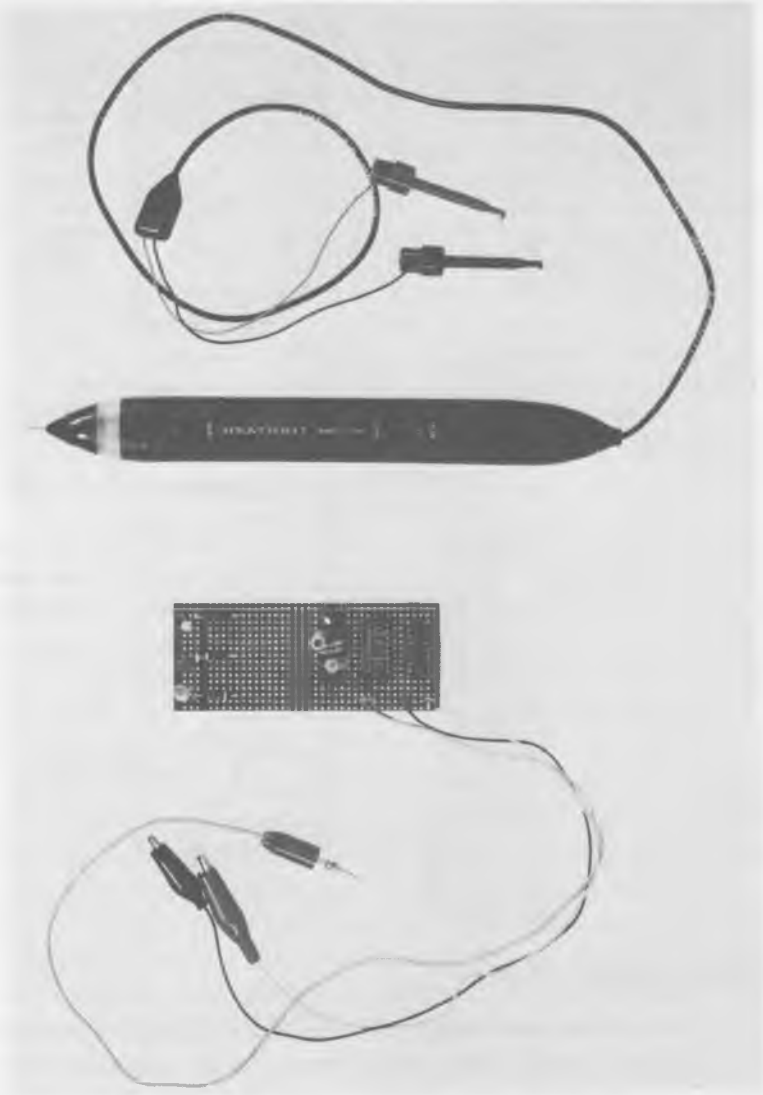
FIGURE 3–2
Schematic of a simple logic probe. It is an invaluable tool for testing digital circuits.



The 8-Bit-wide I/O Ports

One of the most useful features of the 6526 is its 8-bit-wide input/output ports. The 6526 has two of these ports, designated port A and port B. Any bit of each port may be used as either an input or an output line. Each of these 16 I/O lines is thus a bidirectional data path to and from the MPU data bus. In many applications, all 8 bits of the port operate in unison as inputs or outputs. This port configuration is generally termed parallel because the data path conveys all 8 bits of a byte side by side. Each of the port's 8 lines, designated P0–P7, corresponds to a bit in that port's register. Port B's register, one of the 16 address locations occupied by the 6526, is found at 56577 decimal. When all 8 bits of the port are

FIGURE 3–3
Photo of the home-built logic probe and a commercial unit. Both work well. Top photo reprinted by permission of Heath Company.



in the output mode, binary numbers stored at this address control the lines PB0–PB7. For example, consider the binary number:

10010010

This is equal to 146 decimal since the 128, the 16 and the 2 weight bits are set. Writing that number to the port B register would cause PB7, PB4, and PB1 to assume +5V while the rest of the lines would output 0 volts.

When all 8 bits of the port are configured as inputs, the reverse occurs. If PB7, PB4, and PB1 were connected to a +5V source while the rest of the lines were connected to 0 volts, and you read the contents of the port B address, you would find a 146 decimal had been placed there by the 6526's circuitry. Another concept, equally useful in interfacing and nicely provided for by the 6526, is the ability to control each port's bits independently. You could have 3 bits for output while 5 bits are inputs, or any other combination. In yet other applications, it may be necessary to switch all or a portion of the I/O lines back and forth, first inputs, then outputs. The analog to digital converter in the next chapter utilizes this technique. An interface may require only 1 or 2 of the 6526's I/O lines. Any unused I/O line can simply be ignored.

The Port Direction Register

The direction (input or output) of a port's bits is determined by 1s or 0s stored in the port's direction register. To utilize a port direction register, simply put a 1 into any bit that you want to be an output and a 0 for any bit you want to be an input. Table 3–1 gives the memory-mapped address locations for accessing the 6526. Port B's direction register is at 56579 decimal. Table 3–2 illustrates the concept of bit weights. These decimal values can be assigned to each bit of the 8-bit word. As we noted in Chapter 1, the bit weight represents the bit as the power of 2. If we wanted to configure port B as an 8-bit-wide parallel output port, then we would set all bits by storing a decimal 255 at address 56579. Likewise, if we wanted to configure port B as an 8-bit-wide parallel input port, we would store a 0 at 56579. Figure 3–4 gives an example of a bidirectional configuration for the port B direction register that we will use in our experiments later on. This pattern will allow combined input and output via port B. The decimal value of 240 was POKEd to location 56579 to produce the resulting I/O paths for port B register at location 56577.

Header

If you plan to perform the following experiments in this chapter, you will need to construct the I/O header as illustrated in Figure 3–5. The I/O header has 4 LEDs for indicating output status and 4 pairs of input lines connected to switches for input control. This is a simple interface yet very

TABLE 3-2: Bit Weights

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

FIGURE 3–4
Data-direction register configuration for the experiments presented here. Notice how a 1 causes the corresponding bit in port B to be an output and a 0 causes the bit to be an input.

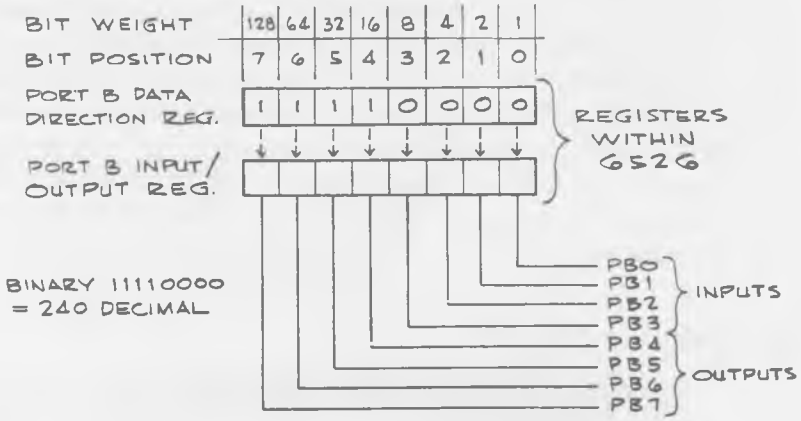


FIGURE 3–5
Photo of the header used in these experiments.

useful in learning to use the C64's USER PORT and some basic 6526 programming. The I/O header will be connected through the USER PORT by mounting the header on a 4¼-inch by 5½-inch perforated experimenter card. The header is plugged into a wire-wrap socket that is glued to the card and wired to the edge connector as described below.

Header Construction

Use only the microminiature TLR-121 LEDs as shown. They require a low current to glow and can be safely connected directly to the 6526's I/O lines. Solder the 4 LEDs onto a 16-pin dip header, making sure the anodes and cathodes are oriented as shown in Figure 3-6. The LEDs will indicate the output status of the port bits to which they are connected. Next, solder a 4-position single-pole single-throw (SPST) DIP switch to the header as shown in Figure 3-6.

Mount a 16-pin wire-wrap socket to a 14½-inch by 5½-inch perforated experimenter's card with a 22/44 card edge connector. Wire-wrap leads to the socket and solder them to the card connector as shown in Figure 3-7. You will also need to construct an adaptor connector for utilizing 22/44 edge cards with the USER PORT's 12/24-pin edge connection. Solder two female edge connectors together as shown in Figure 3-8. Be careful to avoid shorts in all of your soldering.

Digital Output Exercises

Turn off the computer and plug the experimenter board into the USER PORT using the adapter connector described above. Next, install the header into its socket and turn on the computer. As soon as the computer is turned on the 4 LEDS should dimly light. Although the 6526 is initialized by the Commodore 64 in the input mode, the 6526 outputs a small current

FIGURE 3-6
Wiring diagram of the header.

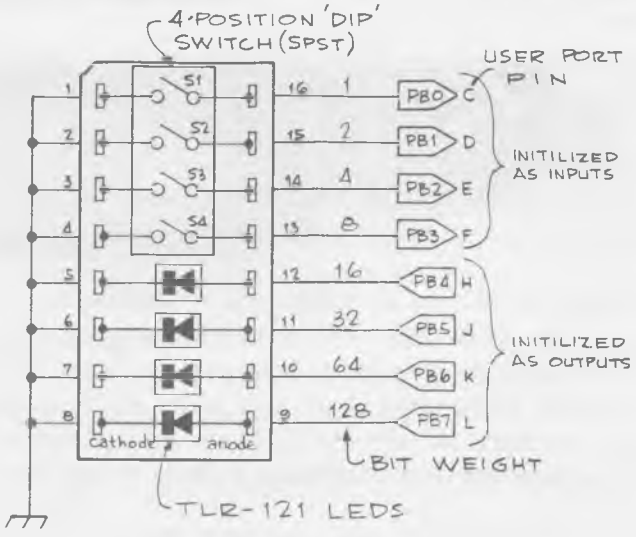


FIGURE 3–7
Details of the header card.

WIRING TABLE

Wire Wrap Socket	User Port
1	1 - GND
2	1 - GND
3	1 - GND
4	1 - GND
5	1 - GND
6	1 - GND
7	1 - GND
8	1 - GND
9	L - PB7
10	K - PB6
11	J - PB5
12	H - PB4
13	F - PB3
14	E - PB2
15	D - PB1
16	C - PB0

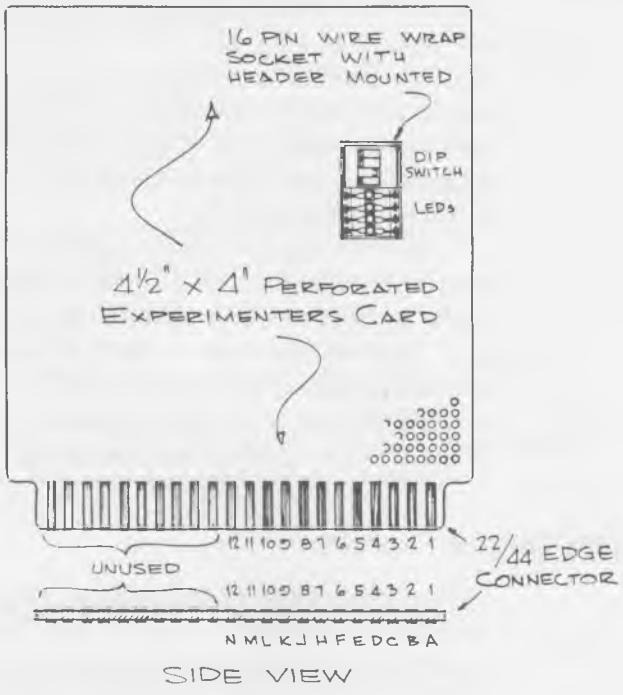
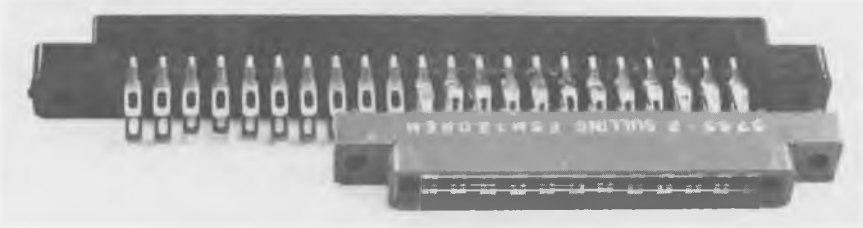


FIGURE 3–8
An adapter that allows the standard 22/44 experimenter's board to be plugged into the C64's user port. This adapter can be used with all of the projects in this book that are interfaced at the user port.



because the I/O lines are pulled up to +5 volts through 10,000 ohm resistors inside the 6526. This small current is enough to dimly light the LEDs. If the LEDs do not light as indicated, turn the Commodore 64 off and check for wiring errors. Your logic probe may be useful in tracking down any problems. Be sure the LEDs are not installed backwards.

Once you have everything working, enter:

POKE 56579, 240 (and RETURN).
POKE 56577, 0

All 4 LEDs should turn completely off. They turn off because you set the 4 high-order bits of port B to be outputs; 240 being the sum of 128 + 64 + 32 + 16 (and you POKEd all zeroes into port B).

After setting the upper 4 bits for output, POKEing numbers to the port B register will now control the LEDs. Execute the following:

```
POKE 56577, 16
```

The LED connected to PB4 will light. Now enter:

```
POKE 56577, (16 + 32)
```

Two LEDs associated with the weights 16 and 32 will lite.

```
POKE 56577, 32
```

Only the LED connected to PB5 will be on. Now, here's a new trick. Execute this command:

```
POKE 56577, (PEEK(56577) OR 128)
```

The LEDs connected to PB5 and PB7 should now be lit. This is because we are also able to read the values being output. If the original 32 (Bit 5 set) is logically ORed with 128 (Bit 7 set) then 160 will result and both LEDs will be turned on. Look and see what is in the port B register. Enter:

```
PRINT PEEK (56577)
```

and, of course, you get 160. ORing as shown is useful for setting output bits while maintaining the status of the other bits. Thus, it is possible for different programs to control only certain bits of a port without interfering with each other. To clear a bit when an unknown condition exists on the adjacent bits, you will want to use the logical operation AND. To use the AND you must set only the bits you want cleared of the zero state. For example, enter:

```
POKE 56577, 240
```

and all four LEDs will turn on. Now enter:

```
POKE 56577, (PEEK(56577) AND 176) (Note: only bit 6 is low
in 176).
```

This will cause only the LED connected to PB6 to go off. We encourage

the reader to experiment with the LEDs using the AND and OR operators. In several of the following projects, ORing and ANDing are used for controlling a device through the CIA. You will need to understand the intent of these two logical operations to follow the programs associated with the projects.

We just learned some output control concepts using BASIC in the immediate mode. In other words, things happened as we entered a command followed with a RETURN. Now let's try some programmed output control. Enter and run the short program that follows:

```
10 POKE 56579, 240                  (Bits 0–3 input;
                                     bits 4–7 output)
20 INPUT"ENTER BIT WEIGHT";W        (Get output value)
30 POKE 56577, W                    (Output to Port B)
40 GOTO 20                          (Go do it all again)
```

When first run, the LEDs should be completely off. Enter 240 followed by a RETURN and all 4 LEDs should light. Now enter 16 and only the first LED will light. Any single LED may be controlled by entering its associated bit weight and any combination may be controlled by adding bit weights. You can change line 30 to use AND or OR as discussed above.

The following short programs demonstrate two ways to compute bit weights for output control. To stop the programs, hold down RUN/STOP and press RESTORE, which will reset the Commodore 64 and the 6526.

```
 1 REM BINARY COUNTER
 2 REM LEDs REPEAT COUNT 0
   TO 15
10 PB = 56577 : POKE PB +2, 240   (Initialize port B Direction
                                   Register)
20 FOR A = 0 TO 15                (REM Count module 16)
30 POKE PB,A * 16                 (Shift 4 bits left and out-
                                   put to port B)
40 FOR T = 1 TO 1000 : NEXT T     (Pause a while)
50 NEXT A
60 GOTO 20                        (Go do next count)

 1 REM EVENT SEQUENCER
 2 REM SEQUENTIAL CONTROL
10 PB = 56577 : POKE PB + 2,
   240                            (Initialize port direction)
20 FOR A = 4 TO 7                 (Do 4 to 7)
30 POKE PB, 2 ↑ A                 (Output to port B)
```

```
40 FOR T = 1 TO 300: NEXT T     (Pause)
50 NEXT A                       (Do next bit weight
                                 higher)
60 GOTO 20                      (Repeat entire sequence)
```

The first program demonstrates computing and outputting 4 bits with a binary progression. This application could be useful in a multiplexing and demultiplexing application. Notice how variable A was shifted left 4 bits simply by multiplying by 16. The second program demonstrates a sequencing arrangement. Sequential control is commonly used where one event follows the next, such as with a traffic light. In both programs, line 40 is a delay to slow the program down. Change the FOR/NEXT value for more or less delay. In critical real time applications, you may want to synchronize with the system clock using the TI function, or perhaps use one of the 6526's timers.

Many other variations can be devised to control the 4 LEDs on our demonstration header. Try some of your own. But remember, always set up the data direction register, as in line 10, before sending out information.

Digital Input Exercises

The computer's ability to sense input is equally important. The 6526 in your Commodore 64's user port can be initialized for up to 9 input bits, using both ports A and B. The 4 switches connected to the header serve as input signals connected to the 4 least significant bits (LSBs) of I/O port B. We could have chosen any of the bits for input, but the 4 we chose will be adequate to demonstrate how inputs may be received by the 6526.

Let's first experiment with some simple input control in the immediate mode. Enter the following command and hit RETURN:

```
POKE 56579, 240
```

This, of course, initializes the port B direction register with bits 0–3 as input and bits 4–7 as output. Now use the logic probe to check header pins 13, 14, 15, and 16. Note that the pins will be high if the associated switch is off and low if the switch is on. Set all 4 switches to the on position. Now enter:

```
PRINT PEEK (56577) AND 15
```

A zero should be displayed on the screen. Turn off the switch connected to PB2 and check pin 14 with the logic probe. It should be in a high condition; the other 3 inputs should still be low. Enter:

```
PRINT PEEK (56577) AND 15
```

This should now cause a 4 to be printed on the screen.

You might wonder why we AND the inputs from port B with 15. This is required because only the 4 LSBs are actually inputs. However, any output data on the 4 most significant bits (MSBs), bits 4–7, will also be read when we PEEK port B. We can cause the 4 MSBs to become 0s by masking them off using the AND 15. Remember anything ANDed against a 0 yields a 0. Thus the high-order bits will be ignored. Experiment with different combinations of switch settings. Verify that any input line which is put in a high state by turning off that line's switch will cause the weight of that bit to be added to the contents of the port B register. Note that the switches seem to work backwards. That is, turning a switch to off turns on that bit because the pull-up resistors in the 6526 pull all of the input lines up to +5 volts (a logic 1) when the switches are off. When a switch is on, the line is shorted to ground and assumes a logic 0.

In the following short program we will initialize port B, bits 0–3 for input and bits 4–7 for output in a combined input/output operation. The switches should all be in the ON position when you begin (header pins 14–16 should be low). Enter the following program and run it.

```
10 PB = 56577 : POKE PB + 2, 240   (Half input, half output)
20 IN = PEEK(PB) AND 15            (Input & strip 4 MSBs)
30 POKE PB, IN *16                 (Shift to MSBs & output)
40 GOTO 20                         (Go get next input)
```

With the program running, all 4 LEDs should be off. Turn OFF any 1 or combination of the switches and the associated LED will light. Without a doubt, the program demonstrates a most complicated way of turning on and off a LED!

Notice how the *16 in line 30 causes a 4-bit shift to the left before output. For example, if PB2 input is high, then it is equal to weight value 2. 2 * 16 will result in the bit-weight 32 to be output on PB5. In this way, input PB0 countrols output PB4, input PB1 controls output PB5, and so on. Additional lines of code may be inserted into the above program to allow screen acknowledgment of the input. For example, add:

```
35 PRINT "BIT"; IN          (Print bit-weight on monitor)
```

This will scroll a particular bit-weight or sum of weights on the monitor as long as the associated input(s) are high.

The 6526 Timers

In the 6526 there are two 16-bit interval timers, designated Timer A and Timer B. Each has its own particular characteristics and functions. One obvious use of the 6526 in the Commodore 64 is, of course, to generate the time of day clock. Whenever the Commodore 64 is powered up or reset, 6526 #2 is initialized not only to control the keyboard, etc., but one of the timers is configured to issue an MPU (microprocessor unit) interrupt 60 times each second. The interrupt, in turn, forces the Commodore 64 executive program to execute the interrupt service routine which checks for any keyboard input, increments the system's time-of-day clock, and does any other background task.

Controlling the Timers

The two timers in each 6526 occupy registers 4, 5, 6, and 7 and are controlled by bits in registers 14 and 15. These numbered registers are at the base address of the 6526 plus the register number, i.e., 56320 plus register for one 6526 and 56576 plus register for the other, the latter being more useful for interfacing since it is not used by the Commodore 64 operating system. The registers pertaining to the timers are:

REGISTER	*NAME*	*DESCRIPTION*
4	TA LO	Timer A low-order register
5	TA HI	Timer A high-order register
6	TB LO	Timer B low-order register
7	TB HI	Timer B high-order register
13	ICR	Interrupt control register
14	CRA	Control register A—controls Timer A
15	CRB	Control register B—controls Timer B

The timers are actually 16-bit timers, and can thus count as many as 65535 pulses from various sources. Since the registers are only 8 bits, reading or writing requires a low-order and a high-order access (PEEK or POKE). To set the timer for 60,000 pulses, for example, you would POKE the high-order byte with

```
60000/256 = 234 (fraction ignored).
```

The low-order byte is POKEd with the remainder:

```
60000 - 256*234 = 96.
```

The two timers A and B are similar and are controlled respectively by CRA and CRB but are controlled in slightly different fashions. You can start, stop, and change modes of a timer by POKEing a suitable control word into its CR. (Table 3–3 lists useful CR codes.) Each bit has a function; you may remember that bit 0 has a weight of 1 and bit 6 has a weight of 64. The first five bits for both timers operate identically:

Bit 0 A 1 starts the timer. A 0 stops it.

Bit 1 A 1 enables the timer output to appear on PB6 for timer A and on PB7 for timer B.

Bit 2 A 1 causes the PB6 or PB7 output to toggle each time-out, while a 0 causes a brief negative-going pulse to appear each time-out.

Bit 3 A 1 causes the output pulse to appear only once (one-shot). A 0 causes continuous pulses.

Bit 4 A 1 force-loads the timer. This is a write-only bit and always reads back a 0.

The two timers differ in the functions of bits 5 and 6 of their respective CR's. For CRA,

Bit 5 A 0 causes timer A to count the Commodore 64 main clock pulses (1 + MHz). A 1 allows the count of pulses from outside on the CNT line.

Bit 6 Used by the serial port. Does not affect timer A.

TABLE 3-3: Useful Control Register Codes for Timer A and Timer B

HEX	*DECIMAL*	*RESULT*
00	0	Stop timer
01	1	Start clock counting ϕ2 (1.022 MHz)
07	7	Start timer (ϕ2) with square wave on PB6 (PB7 for timer B)
03	3	Start free run (ϕ2) with 1 us strobe pulses on PB6 (PB7 for timer B)
0C	13	One-shot delay counting ϕ2
0F	15	One-shot pulse on PB6 (PB7 for timer B)
21	33	Start timer counting CNT pulses
41	65	Start timer B counting under flows from timer A (control register B only); using this mode, time delays of up to 73 minutes are possible

For timer B, there are four combinations of bits 5 and 6 of CRB which provide four different modes of inputting pulses which are counted:

BIT 6	*BIT 5*	*TIMER B COUNTS THESE PULSES:*
0	0	Commodore 64 clock (1 MHz)
0	1	Positive transitions on CNT line
1	0	Timer A underflow pulses
1	1	Timer A underflow pulses, providing CNT is high

The Interrupt Control Register (ICR) has one bit for each timer which determines whether an interrupt will be passed on to the microprocessor when time-out occurs (see Table 3–4). By writing a control word to the ICR, the individual interrupts can be enabled or disabled. In binary, the word

```
100000BA
```

will enable the A and/or B interrupts if the value A and/or B is 1. Thus, to enable timer B interrupts, POKE a 128 + 2 into ICR. Likewise, you can disable timer interrupts with the binary word

```
000000BA
```

where a 1 in the A or B position disables the corresponding interrupt. To disable B interrupts, for example, POKE ICR, 2.

The above control of interrupts involved writing to the ICR. Reading the ICR is unrelated to the bits written. Instead, bit 7 is set if any event has occurred in the 6526, whether or not the interrupt has been enabled

TABLE 3-4: The Interrupt Control Register

BIT	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
Read Side	Event Occurred	0	0	Flag	SP	Alarm	TB	TA
Write Side	Set/Clear Indicator	X	X	Flag	SP	Alarm	TB	TA

The CIA chip monitors events from 5 different sources. Any event will set the designated bits in the read side of the interrupt control register. Thus, this register can be PEEKed to see whether an event had occurred. Bit 7 will be set if any of the 5 events had occurred. Bit 4 will be set on a high to low transition on the flag pin. Bit 3 indicates that data has arrived or left the serial port. Bit 2 will be set when the time of day alarm goes off, and bits 1 and 0 will be set when either timer B or A underflows. Whether the event causes a system interrupt in addition to setting the bits depends on what has been POKEd to the write side of the interrupt control register. Any function's interrupt can be enabled by POKEing a number to that port with bit 7 and the bit corresponding to the desired function set. Similarly, any function's interrupt can be prevented by POKEing a number to that port having bit 7 clear and the bit corresponding to the function set.

for passage on to the microprocessor. Also, bit 0 is set if a timer A underflow has occurred and bit 1 is set for timer B. Reading the ICR clears any bits that are set, so you get only one PEEK.

One more note may be helpful to all the "fine print" above. Timer B can be made to count Timer A underflow pulses, so that the two are cascaded together. When this is the case, the combination is a 32-bit counter that can count over 4 billion pulses. Even counting at the fast Commodore 64 clock rate (1 MHz) requires over 4000 seconds (over one hour) for the combined timers to underflow.

Interval Timer Applications

One useful application for the 6526 is to time real-world events. Although these timers are best suited for the relatively short periods applicable to digital circuits and machine language programming, we will present an experiment to illustrate how to initialize and use Timer A as a precision interval timer using BASIC. This experiment will turn the PB6 LED on and off 10 times per second.

The timer decrements a user-set register at the MPU clock rate. When the counter reaches 0, two things occur: a flag bit is set in the interrupt control register (ICR) and the countdown restarts from the original user-set value. Since the MPU clock is quartz-crystal controlled, precision timing is guaranteed.

To demonstrate the use of Timer A, enter and run the following program:

```
 10 CIA = 56576          Base address of CIA
 20 DB = CIA + 3         Data-direction register B
 30 TL = CIA + 4         Timer A—low byte
 40 TH = CIA + 5         Timer A—high byte
 50 ICR = CIA + 13       Interrupt control register
 60 CA = CIA + 14        Control register A
 70 POKE DB, 240         Set direction register
 80 POKE ICR, 127        Disable interrupts
 90 POKE TL, 56          Load Timer A—low byte
100 POKE TH, 199         Load Timer A—high byte
110 POKE CA, 4 + 2 + 1   Set timer mode and start it.
```

You should be able now to see pulses on PB6 at a 10 Hz rate, that is, 10 flashes per second. By varying the values POKEd into the timer registers, the pulse rate should change. The lowest value (1 into TL, 0 into TH) will produce pulses at a 510,000 Hz rate, certainly not visible on a LED but easily seen on an oscilloscope. You can even clear out the program

with a NEW command and the timer will continue to operate once it has been started. There are many other possible ways to use the timers.

The Serial Ports

Two very powerful serial ports are available to the user and are not used at all by the operating system. These ports are labelled SP1 and SP2 and are used in conjunction with their respective clocking lines CNT1 and CNT2. The two ports are derived from the two CIAs and are not to be confused with the IEEE Serial Bus that handles the disk, printer, and other peripherals and that uses entirely separate portions of the CIA chips.

The Commodore 64 supplies only one 8-bit port on the user connector. For many applications, however, you may need more ports. There are several ways to multiplex additional lines into this port to expand its capabilities. For example, Chapter 8 shows one scheme in which a counter is used to direct output data to several different ports. One of the most attractive ways to expand the number of ports is to utilize the capabilities of the 6526's serial port. By cascading shift registers, an almost unlimited number of 8-bit parallel ports can be generated. The use of these shift registers is illustrated in the following two sections of this chapter.

The ports can operate in either the input or the output mode, as selected by appropriate bits in control registers. The data is transferred one byte (8 bits) at a time over the SP line, which goes low if the bit is 0, and high if the bit is 1. The first bit transmitted is bit 7 and the last is bit 0. In the transmit mode, a bit is placed on the SP line, and at the same time the CNT lines goes low. After a delay of one-half of a bit-time, the CNT line returns high; by then the data is settled and valid and it is expected that the receiving device will capture the data and await the next bit. After 8 bits are transmitted, the procedure stops, with the CNT line high and the SP line in its last condition. The entire byte transmission process is started, after a suitable selection of mode and timing, by simply storing the byte in the serial port's data register.

A reversal of the SP and CNT lines is used for inputting data, with the external device providing the data serially and exactly eight CNT pulses per byte of data. The CIA will pick up a bit of data each time the CNT line rises from low to high, exactly the same way the CIA expects data to be received when it is transmitting. When a byte of data has accumulated, a flag in the CIA is raised to advise the Commodore 64. By now you should have gathered that the ideal external device for receiving the transmitted serial data would be another 6526 CIA (or its predecessor, the 6522 VIA used in the VIC 20), but another computer must be part of the external system in order to set up and control the external CIA. We will shortly

demonstrate the use of the Commodore 64's CIAs in input and output operations using less complicated chips.

So far, we have said nothing about the rate of data transmission as bits per second (Baud rate). The serial ports on each CIA operate at a rate determined by the values stored in the respective high-order and low-order registers of the CIA's Timer A. If we are using SP2 and CNT2, we have complete freedom to set its Timer A at any value we like. Start-up of the Commodore 64 stores a $FFFF in the timer registers, and we can operate at a very slow rate by not changing this value. At the other extreme, we can store $0001 into the timer (0 into the high-order register and 1 into the low-order) and run our data transmission at 250,000 Baud, over ten times the highest standard rate for RS-232 communications.

We do not have the same timing flexibility when using SP1 and CNT1, since Timer A of their CIA is occupied with the main system timing in controlling the keyboard, key repeat rate, cursor flashing rate, and so on via interrupts. You may vary this timing within limits, but don't expect the real-time clock to be valid or the cursor to flash at the normal rate. If this timing is set below a limiting value, the computer will go dead while it processes continuous interrupts.

We will now describe some simple circuits and programs to demonstrate use of the serial ports to export and import data to and from the outside world. These circuits can be modified as required to turn external devices on and off and read digital data from outside.

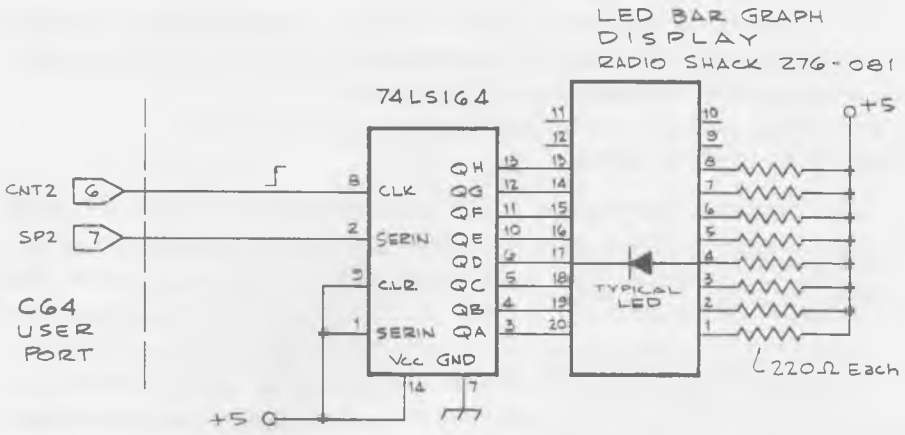
Using a Serial Port to Output Data

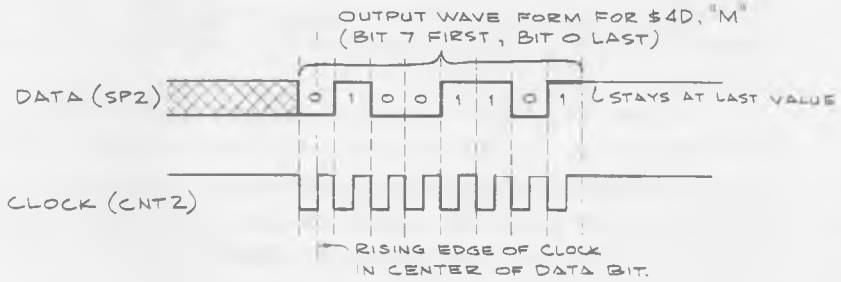
A simple circuit to visually demonstrate use of the CIA as a serial output port is given in Figure 3–9. It consists of eight LEDs to see the data transmitted plus a serial-in parallel-out shift register, 74LS164, to capture the data. By hooking its CLK (clock) line to the user port's CNT2 line, it will shift a bit of data into its first bit and move each bit into the next position each time the CNT2 line goes from low to high (note: the CIA says that its data is valid on the rising edge of CNT2). If one byte of data is transmitted, it will appear as an output of the 74LS164. To make it easier to read, we will invert the data in software before transmission, so that 1s will light the LEDs and 0s will turn them off.

When you have wired the circuit, you may enter this program:

```
40 POKE 56590, 81                (set SR mode, start timer)
50 GET A$:IF A$ = ""THEN 50      (wait for keypress)
60 PRINT A$;                     (put on screen)
70 POKE 56588, 255-ASC (A$)      (send out in inverted form)
80 GOTO 50                       (repeat)
```

FIGURE 3–9
Demonstration circuit for the serial port output.





RUN this program; when you press a key, you will see its ASCII value slowly shifting into the shift register. After shifting stops, the key's ASCII value will be shown by the LEDs. Try pressing the space bar three or more times. You will see the single 1-bit enter the register and move to its correct position.

If you add the following lines to your program, you will see the circuit operate at its fastest speed (250,000 baud).

```
10 POKE 56580, 1          (Timer A—low)
20 POKE 56581, 0          (Timer A—high)
30 POKE 56589, 127        (mask interrupts)
```

When you run the above, try hitting alternate U's and *'s to get a useful test pattern.

If you need more than eight control lines, you can cascade two or more 74LS164s together by connecting the QH output of the first to the

serial input of the second, and so forth. The CLK pins should be connected together.

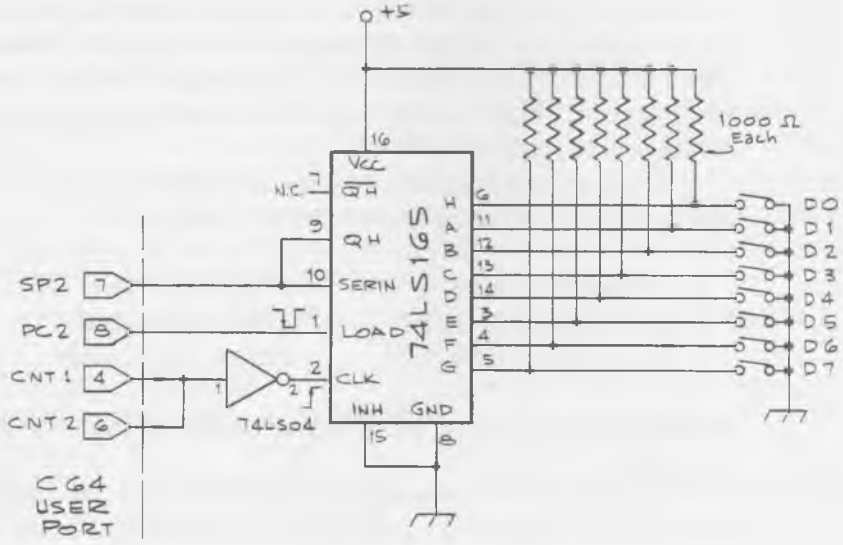
Additional circuitry is required if your application cannot tolerate the switching as the bits go by a particular output. Most applications can tolerate a few microseconds of "glitches."

Using a Serial Port to Input Data

Demonstrating data input is a little more complicated, since the external circuit must provide exactly 8 clock pulses per byte. Rather than recommending a complex circuit, we have chosen to use one serial port for data transfer and the other port's CNT line to provide the 8 clock pulses; we wire this CNT line to the other to make-believe the clock came from outside. The circuit, shown in Figure 3-10 consists of eight switches, one gate of a 74LS04 inverter, and a 74LS165 parallel-in serial-out shift register. We will use CNT1 as the clock, and input clock and data to CNT2 and SP2, respectively. Another useful feature of the CIA is the PC (peripheral control) line which provides a low-going pulse when port B is read or written. The shift register is loaded at the start of the cycle by a pulse on PC2.

We have two edges of the clock pulse to use: the low-going edge, which occurs first, and the rising edge. We know the CIA is going to load data on the latter, so we can use the first edge to shift the shift-register. However, we will lose the data in the first bit unless we can save it somehow. We do this by recycling the output QH back to the serial input

FIGURE 3-10
Demonstration circuits for the serial port input.



connection and retrieve this bit after eight shifts. Since the eighth bit should be bit 0, we connect our DO to the H input and offset other bits by one position.

Since the shift register shifts on the rising edge, we must invert CNT1 in a 74LS04 before applying the signal to the shift-register CLK pin.

After wiring up the demonstration circuit, enter and RUN the following program:

```
10 POKE 56335, 8              (mask CIA1 serial interrupt)
20 POKE 56589, 127            (mask all CIA2 interrupts)
30 POKE 56334,
(PEEK (56334) OR 64)          (make CIA1 serial port an output)
40 X = PEEK (56577)           (load the shift register)
50 POKE 56332, 0              (get 8 clock pulses from CNT1)
60 IF (PEEK (56589) AND 8)
= 0 THEN 60                   (wait until byte is ready)
70 X = PEEK (56588)           (get the data byte)
80 PRINT X;                   (print value)
90 GOTO 40                    (repeat)
```

While the program is running, change the switch settings and watch the numbers change. If all the switches are closed, the printed numbers should be 0; if all are open, the numbers should be 255. You should be able to get any number in this range by a proper binary combination of switch settings.

Note that the shifting and conversion are relatively slow. This is because we are getting our clock pulses from CIA1 which has its Timer A tied up with system functions and we have not dared to change its timing rate. You could easily revise the circuit and the program to input the clock and data via CNT1 and SP1 and get the clock from CNT2; this change would allow freedom to change the shifting rate. Simply connect the shift register output to SP1 instead of SP2 and revise the program to reverse the roles of the two serial ports.

If an application requires the input of more than 8 bits of digital data, two or more 74LS165s can be cascaded together. Connect the QH output of the first to the serial input of the second, etc. The QH of the last should be recycled to the serial input of the first. The LOAD and the CLK pins should be bussed together. The program requires a single load pulse for the multibyte transfer, but requires clocking and data-fetching once for each shift register. For more than two 74LS165s, buffer the LOAD and CLK lines via two gates of the 74LS04 in series for each line.

Counting External Events

One of Timer A's modes is to count clock pulses on the external input, CNT. A useful application for this capability is to have the Commodore 64 function as a frequency counter. Listing 3–1 shows such a program. The program is inserted as an interrupt service routine. On the first interrupt, Timer A is loaded with $FFFF and then enabled to count pulses on CNT (pin 6 of the user port). After a predetermined number of interrupts,

LISTING 3–1
Frequency counter

```
LINE# LOC   CODE        LINE

00001  0000             ; FREQUENCY COUNTER
00002  0000             ; RETURNS THE NUMBER OF COUNTS THAT
00003  0000             ;OCCURED ON USER PORT #6 BETWEEN
00004  0000             ;KEYBOARD INTERUPTS -16.666 MS.
00005  0000             ;LOWBYTE IN 253 HIBYTE IN 252.
00006  0000             ;
00007  0000             *=$C000
00008  C000             ;-------------------------
00009  C000             ; EQUATES
00010  C000             IRQVEC=$0314            ;IRQ VECTOR
00011  C000             TIMLO=$DD04             ;TIMER HIGH BYTE
00012  C000             TIMHI=$DD05             ;TIMER LOW BYTE
00013  C000             CRA=$DD0E               ;CIA CONT. REG.
00014  C000             OLDIRQ=$EA31            ;OLD IRQ ADDRESS
00015  C000             CIAIR=$DD0D             ;INTER REG
00016  C000             ;-------------------------
00017  C000  08        SINT    PHP             ;SAVE STATUS
00018  C001  78                SEI             ;INHIBIT IRQ
00019  C002  A9 14             LDA #$14        ;LOW BYTE OF START
00020  C004  8D 14 03          STA IRQVEC
00021  C007  A9 C0             LDA #$C0        ;HIGH BYTE OF START
00022  C009  8D 15 03          STA IRQVEC+1
00023  C00C  A9 03             LDA #$03        ;MASK OUT TIMERA INTERUPT
00024  C00E  8D 0D DD          STA CIAIR
00025  C011  28                PLP
00026  C012  58                CLI             ;ALLOW INTERUPTS
00027  C013  60                RTS
00028  C014             ;-------------------------
00029  C014  48        START   PHA             ;SAVE AC
00030  C015  C6 FF             DEC $FF         ;READY TO READ?
00031  C017  F0 04             BEQ READ
00032  C019  68                PLA             ;NO RETURN
00033  C01A  4C 31 EA          JMP OLDIRQ
00034  C01D  A5 FB       READ  LDA $FB         ;GET INDEX
00035  C01F  85 FF             STA $FF         ;PUT BACK AT $FF
00036  C021  A9 00             LDA #00         ;STOP TIMER
00037  C023  8D 0E DD          STA CRA
00038  C026  AD 05 DD          LDA TIMHI       ;GET TIMER VALUES
00039  C029  85 FC             STA $FC         ;STORE AT FC & FD
00040  C02B  AD 04 DD          LDA TIMLO
00041  C02E  85 FD             STA $FD
00042  C030  A9 FF             LDA #$FF        ;SET TIMERS TO
00043  C032  8D 05 DD          STA TIMHI       ;$FFFF
00044  C035  8D 04 DD          STA TIMLO
00045  C038  A9 21             LDA #$21        ;START COUNTER
00046  C03A  8D 0E DD          STA CRA
00047  C03D  68                PLA             ;RESTORE AC
00048  C03E  4C 31 EA          JMP OLDIRQ
00049  C041             .END
```

the clock is read and restarted. Since the interrupts occur at precisely 60 per second, the time-base for the count is accurately known. The number of interrupts per sample is determined by the contents of $FB (decimal 251). The latest count is automatically put in addresses $FC (the low order byte) and $FDC (the high order byte). These addresses are found at 252 and 253 decimal respectively.

Listing 3–2 is a BASIC program that POKEs the machine language program into memory and starts it. Lines 145 through 220 read the addresses $FC and $FD and calculate the current frequency. Note that 1/60th of a second is 0.0166666 seconds. A time-base of 10 interrupts or one every .1666 seconds was chosen for this example. The time-base can be changed to count slower or faster frequencies by simply changing the value in line 145. The time-base should be small enough that no more than 65,535 events will be counted between updates. If that occurs, Timer A will overflow and an erroneous frequency will be calculated. Time-bases from 16 ms to 4.24 seconds are possible within this program.

The time-base is very short in this program, and we are not worried about trying to read the answer before it appears in locations 252 and 253. For longer time-bases, this could be a problem. The program can sense the appearance of data in these locations by POKEing a 0 to 253 and then looping with an IF statement until a non-zero number appears in 253.

One note of caution. Don't connect CNT to anything except a TTL-

LISTING 3–2
Basic version of 3–1

```
0 REM FREQUENCY COUNTER PROGRAM
1 DATA 0 , 120 , 169 , 20 , 141
5 DATA 20 , 3 , 169 , 192 , 141
10 DATA 21 , 3 , 169 , 3 , 141
15 DATA 13 , 221 , 40 , 88 , 96
20 DATA 72 , 198 , 255 , 240 , 4
25 DATA 104 , 76 , 49 , 234 , 165
30 DATA 251 , 133 , 255 , 169 , 0
35 DATA 141 , 14 , 221 , 173 , 5
40 DATA 221 , 133 , 252 , 173 , 4
45 DATA 221 , 133 , 253 , 169 , 255
50 DATA 141 , 5 , 221 , 141 , 4
55 DATA 221 , 169 , 33 , 141 , 14
60 DATA 221 , 104 , 76 , 49 , 234
100 FOR I=0TO64
110 READ X
120 POKE (49152+I),X
130 NEXT I
140 REM ****PROGRAM IN PLACE****
145 BASE=10
150 POKE 255,0:POKE 251,BASE:REM SET UP TIMER
160 REM TIME BASE 10 JIFFIES;166.6 MS
170 SYS 49152:REM START TIMER
180 REM READ TIMER
190 COUNT=(PEEK(252)*256+PEEK(253))
200 F=(65535-COUNT)/(.0166666*BASE)
210 PRINT F
220 GOTO 190
```

compatible signal—that is, a signal that oscillates between 0 and +5 volts. If your signal falls outside that range, you will have to use one of the input conditioning circuits described at the end of this chapter. Application of voltages outside of the 0-to-+5 volt range could permanently damage your computer.

I/O Conditioning

Many computer control applications utilizing either input or output may not be appropriate for direct connection to the user ports lines. The loads may be too great, the voltages may be too large or too small, etc. The following paragraphs give useful solutions for some of these common interfacing problems.

Controlling High-Power Devices

The digital output port described above provides outputs that switch between the TTL levels, 0 and +5 volts. The 6526 outputs can only provide a few milliamps of current and thus are very limited in what they can drive. A very common application for a digital output port is for each bit of the port to control a single function; for example, one bit can turn on the coffee pot, while another bit turns off the living room lights. This delegation of duties obviously cannot be done with the direct outputs from the 6526. We have, therefore, provided the following series of simple interface circuits to help solve such problems.

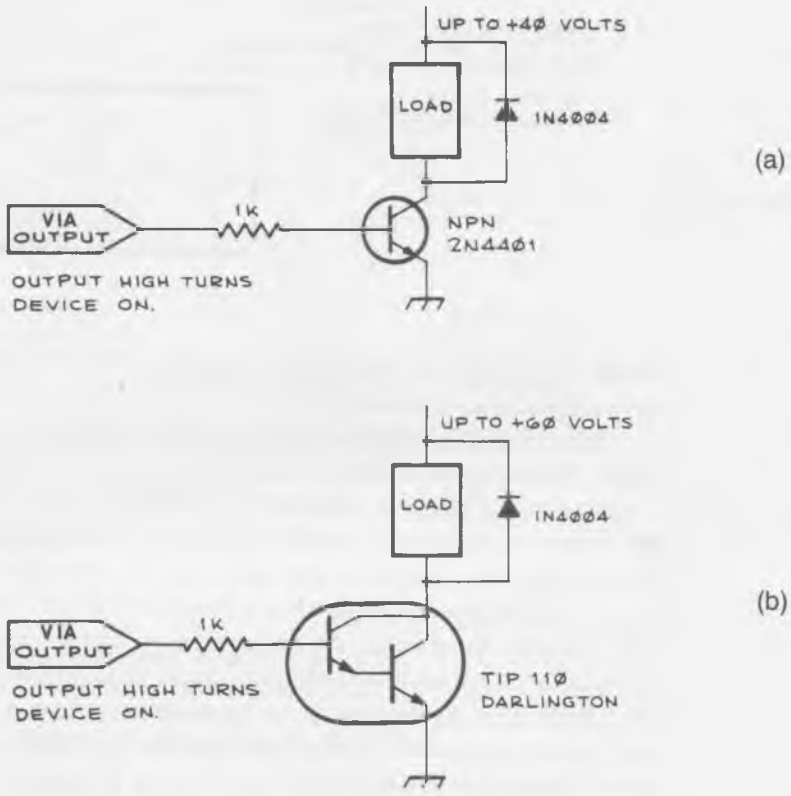
DC Loads

Figure 3-11 shows how a simple NPN transistor can be used to switch DC loads on and off under computer control. When a port output is high, the base of the transistor conducts and allows collector current to pass to the emitter. The 2N4401 will allow up to one-half ampere of current and will tolerate supply voltages of up to 40 volts DC. If the load is inductive—as in a motor, a solenoid, or the coil of a relay—then you must put a diode across the load so the voltage spikes, which are generated when the device is abruptly turned off, do not destroy the transistor.

AC Loads

Many of the devices we want to control operate on 110-volt house current. Because of the high voltages involved and the high current capability of household 110-volt lines, you must be very careful when interfacing such devices to your computer. One inadvertent short and you could reduce

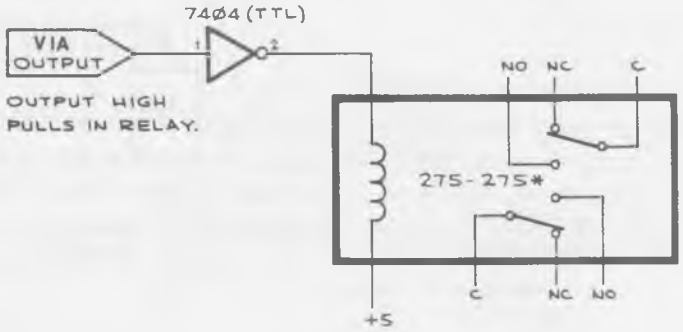
FIGURE 3–11
a. An output from the user port can switch DC loads of up to one-half ampere and 40 volts with this circuit.
b. This circuit will handle loads up to 1 ampere and 60 volts.



your whole computer to a worthless cinder. Therefore, you want to choose devices that are well insulated from the 110-volt (or 220-volt in Europe) line. Traditionally, this precaution has been achieved with relays. There are several relays on the market that can be driven directly by the +5 volts from a latch. Figure 3–12 shows one such relay in use. These relays come in a variety of specifications and can be used for any switching application, including controlling devices of up to 100 to 200 watts.

CAUTION: Unlike the low voltage lines found on the logic boards, 110-volt and 220-volt lines are extremely dangerous. Before plugging these circuits into an outlet, be sure that they are correctly wired, that all connections are properly insulated so that no bare wires exist, and that everything is mechanically secured. DO NOT TOUCH ANY COMPONENTS IN A LIVE AC CIRCUIT. Painful shock or even electrocution could result. Also, do not connect any device exceeding the power

FIGURE 3–12
Sensitive relay controlled by one output from the user port. The number with the asterisk refers to the current Radio Shack part number. Contacts are rated at 1 ampere each for 110VAC.



rating of the relay or triac to the controller. The resulting damage may not be limited to the controller.

More recently, mechanical relays have been eclipsed by solid-state relays. These relays consist of a LED placed next to a light-sensitive triac. Since the only connection between the diode and the triac is a light path, solid-state relays provide excellent isolation. These are much faster than mechanical relays and are much more reliable, since there are no moving parts. The top panel of Figure 3–13 shows a MOC 3010 optocoupled triac used to switch AC devices. We can use it to control a power triac, as shown in the figure. That simple circuit can control up to 600 watts and cost under $3. The bottom figure shows the same circuit adapted for an inductive load, such as a motor. High-powered solid-state relays that can control up to 10 amps are available and cost between $10 and $20. They are sold at most industrial electronics supply houses.

Sensing Low Level and Bipolar Inputs

Many devices we want to inferface do not output TTL logic levels. If we want to sense a signal that is outside the 0–5 volt range of the logic chips or is of such a high impedance that it will not drive the 6526 input, a voltage comparator IC will usually solve the problem. Figure 3–14 shows an LM311 comparator in use as an input conditioner. If the voltage on the noninverting input is greater than that on the inverting input, the output will be logic 0, or 0 volts. If the inverting input has the greatest voltage, then the output will be logic 1 (5 volts). The comparator is very sensitive and as little as a few millivolts difference between the inputs can be detected. Note that the LM311 is an open-collector device and, therefore, requires the 2.2K pull-up resistor between its output and +5 volts. The

FIGURE 3–13
One output bit from the user port can switch up to 6 amperes of 110VAC. The numbers with asterisks refer to Radio Shack part numbers. The upper circuit is for resistive loads while the lower circuit is designed to switch inductive loads, such as transformers and motors.

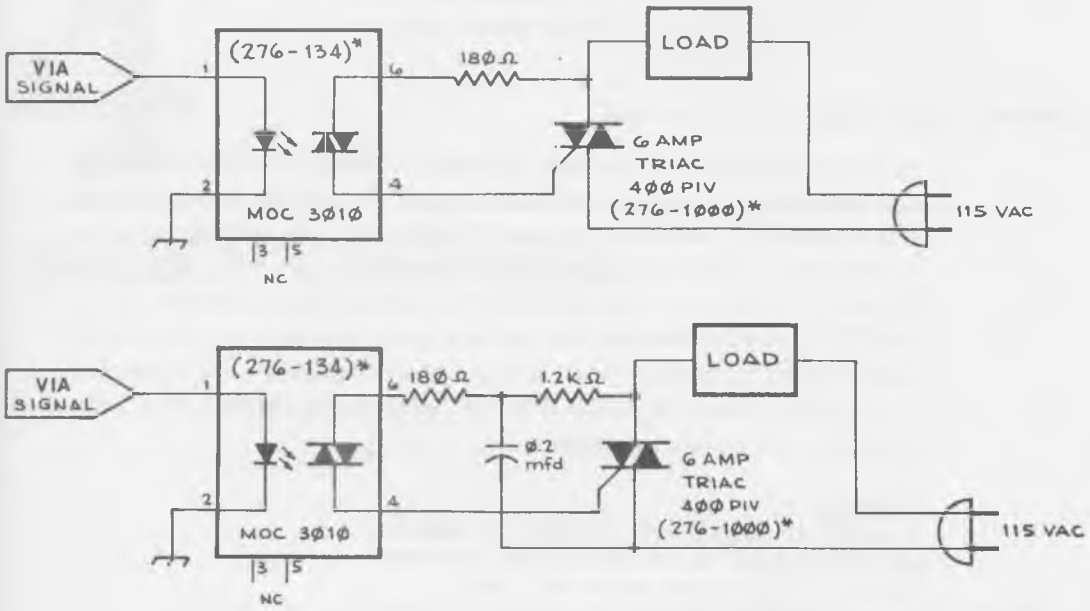
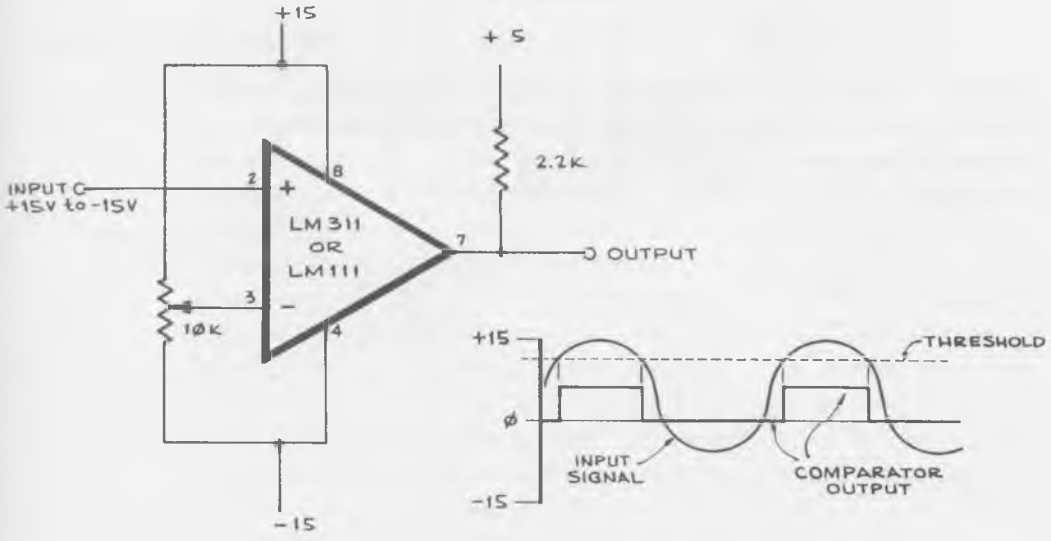


FIGURE 3–14
A comparator as an input conditioner. Note that the output of the comparator will be a logic 1 when the input is above the threshold voltage on pin 3. A simple ± 12VDC supply like the one shown in Figure 11–3 can be used to power the comparator.

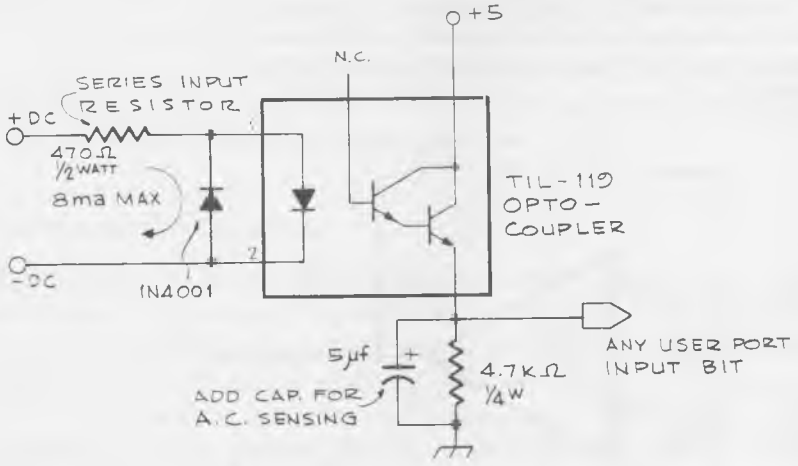


potentiometer must be set so that a threshold voltage is selected that is approximately halfway between the two voltage states expected. This circuit will work with input voltages in the range of −12 to +12 volts and will draw only a few microamperes from the voltage source. Figure 11–3 in Chapter 11 shows how you can generate ±12V from the power connections on the user port to run the comparator.

Sensing High-Voltage Signal Levels

In some applications, it becomes desirable to sense even higher voltages than the levels dealt with in the last paragraph. To provide for high-voltage inputs, the 6526, as well as the entire Commodore 64, must be adequately protected. To accomplish this, we recommend the use of an optocoupler. Figure 3–15 shows a circuit which will handle inputs between +5 to +40VDC. Several hundred volts can be accommodated as long as the series input resistor is chosen to limit the maximum current to 8-ma (resistance = maximum voltage expected/.008). AC signals can be sensed if the 5-mfd capacitor is connected as shown.

FIGURE 3–15
A coupler for sensing high voltages. The series input resistor should be adjusted to limit the current (see text). The 470-ohm resistor will handle voltages from 5 to 40VDC. The opto-isolator gives excellent isolation between the source and the computer.



4

SPEECH SYNTHESIS

The Commodore 64 has built into it one of the most sophisticated sound generators available. Although the Commodore 64's generator is capable of three-part harmony, it does not produce human speech. Speech capability can be added to the Commodore 64, however, with either of the simple projects described below.

Speech Synthesizers

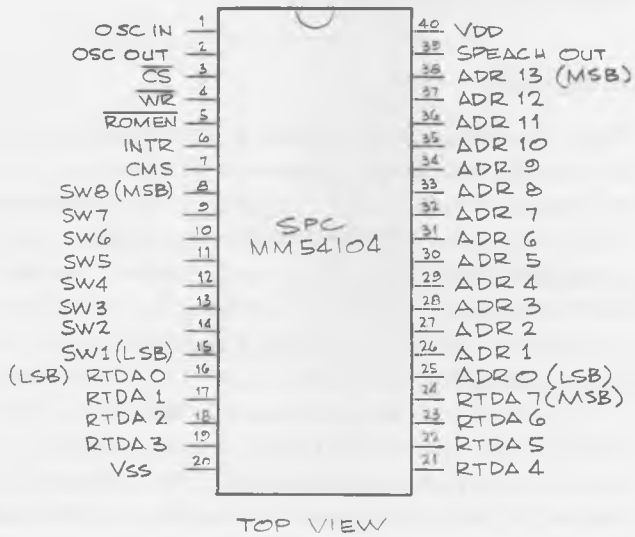
There are several speech synthesizer systems on the market today. Basically, they all take one of two approaches. The first type breaks our speech into its component parts, called "phonemes." These phonemes are the building blocks of the spoken word. By linking together the proper sequence of phonemes, intelligible speech can be created. The other method is to actually record the human voice speaking the various words and to save that information digitally in read-only memories. Each of these systems has both advantages and disadvantages. The first system, the phoneme approach, allows the user an unlimited vocabulary. The shortcoming is that speech quality is poorer and programming is tedious. The second approach allows almost perfect speech quality, but programming the speech information into the read-only memories requires sophisticated equipment. Furthermore, once programmed, the user is limited only to the vocabulary that has been preprogrammed into the read-only memories.

Recently, National Semiconductor has offered a simple and inexpensive speech synthesizer system called the DIGITALKER.* Although the DIGITALKER uses the digitized recording approach, National Semiconductor has solved the problem of programming the speech information by offering a three-chip set: the DT 1050, which includes the speech processor chip the MM54104 and the two 8K read-only memories: the MM52164-SSR1; and the MM52164-SSR2. The speech-processor chip (SPC) and ROMS are currently available from Jameco Electronics, 1355 Shoreway Rd., Belmont, CA 74002 for about $30 for the complete set. In addition, the ROMs can be purchased separately from either Time Domain Systems, Sandy Cove Rd., Rodeo, CA or The Bit Stop, 5958 South Shenandoah Rd., Mobile, AL 36608. Table 4–1 shows the 145 word vocabulary contained in the DIGITALKER ROMs. The numbers refer to the switch code for each of the words. Thus, to say the word "meter" you would have to POKE a 106 to the SPC chip. This will be explained in detail in the programming section below.

How It Works

Figure 4–1 shows the pin assignments for the DIGITALKER SPC. The code for the word to be spoken must be determined from the master word list in Table 4–1. The binary value of that code is asserted on the 8-bit switch bus (SW 1–8). This tells the DIGITALKER where to look in the speech ROMs. When WR goes low, the code on the switch bus is latched

FIGURE 4–1
Pin assignments for the DIGITALKER SPC.



*DIGITALKER is a registered trademark of National Semiconductor.

TABLE 4-1: DT1050 Master Word List—Digitalker

WORD	*CODE*	*WORD*	*CODE*	*WORD*	*CODE*
This is DIGITALKER	0	Q	48	Is	96
One	1	R	49	It	97
Two	2	S	50	Kilo	98
Three	3	T	51	Left	99
Four	4	U	52	Less	100
Five	5	V	53	Lesser	101
Six	6	W	54	Limit	102
Seven	7	X	55	Low	103
Eight	8	Y	56	Lower	104
Nine	9	Z	57	Mark	105
Ten	10	Again	58	Meter	106
Eleven	11	Ampere	59	Mile	107
Twelve	12	And	60	Milli	108
Thirteen	13	At	61	Minus	109
Fourteen	14	Cancel	62	Minute	110
Fifteen	15	Case	63	Near	111
Sixteen	16	Cent	64	Number	112
Seventeen	17	400Hertz Tone	65	Of	113
Eighteen	18	80Hertz Tone	66	Off	114
Nineteen	19	20MS Silence	67	On	115
Twenty	20	40MS Silence	68	Out	116
Thirty	21	80MS Silence	69	Over	117
Forty	22	160MS Silence	70	Parenthesis	118
Fifty	23	320MS Silence	71	Percent	119
Sixty	24	Centi	72	Please	120
Seventy	25	Check	73	Plus	121
Eighty	26	Comma	74	Point	122
Ninety	27	Control	75	Pound	123
Hundred	28	Danger	76	Pulses	124
Thousand	29	Degree	77	Rate	125
Million	30	Dollar	78	Re	126
Zero	31	Down	79	Ready	127
A	32	Equal	80	Right	128
B	33	Error	81	SS*	129
C	34	Feet	82	Second	130
D	35	Flow	83	Set	131
E	36	Fuel	84	Space	132
F	37	Gallon	85	Speed	133
G	38	Go	86	Star	134
H	39	Gram	87	Start	135

TABLE 4-1: DT1050 Master Word List—Digitalker (Continued)

I	40	Great	88	Stop	136
J	41	Greater	89	Than	137
K	42	Have	90	The	138
L	43	High	91	Time	139
M	44	Higher	92	Try	141
N	45	Hour	93	Up	142
O	46	In	94	Volt	143
P	47	Inches	95	Weight	144

*"SS" makes any singular word plural.

into the DIGITALKER. When WR returns to the high state, the speech sequence begins. Data from the ROMs travels over the data bus to the speech processor, which controls both the frequency and gain of the sound generator to produce the speech.

When speech begins, the INTR line goes low. INTR returns to the high state at the completion of the word to signal the host computer that the task is finished. CS is a chip select and must be grounded for the DIGITALKER to operate. CMS is a command sequence pin and must also be grounded for proper operation.

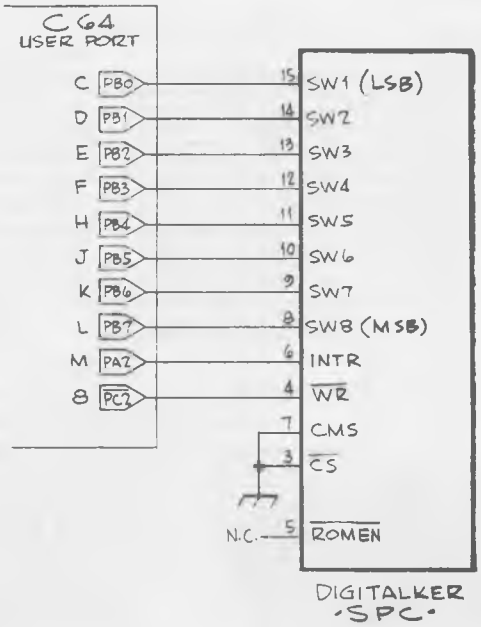
The Interface

Figures 4–2 and 4–3 show the schematic for the DIGITALKER project. We built the circuit on a 4½-inch by 4-inch experimenter card with holes spaced on 0.1-inch centers. Wire-wrap IC sockets were used to hold the ICs, and the discrete components were soldered to Vector T44 wire-wrap pins pushed into the holes of the board. All connections were then made by wire-wrap below the board. You can cut the board as described in Chapter 8 and solder a 12/24-pin edge connector to the card. Alternatively, you can leave the card intact and use the 24 to 44-pin adapter plug shown in Figure 3–8.

One feature of the Commodore 64's sound chip is that it makes provision for an external audio input signal. This signal is mixed with the output of the sound generators and then sent to the TV monitor's speaker. We take advantage of that capability in this project. The external input to the sound chip is found on pin 5 of the audio/video socket next to the serial port on the rear of the Commodore 64. Although the output of the SPC chip can be directly connected to the audio input, the speech quality will not be the best. For that reason, we recommend that you build the amplifier/filter shown in Figure 4–4. To enable the external audio input of the Commodore 64, simply POKE 54296,15.

If you do not wish to use the audio in your monitor, you can substitute

FIGURE 4–2
Connections between the SPC and the user port.



the amplifier/filter in Figure 4–5 for that in 4–4. This amplifier will drive an 8-ohm speaker directly. The completed project shown in Figure 4–6 used this option.

The layout is not critical. Figure 4–6 can be used as a guide for arrangement of the components.

PARTS LIST—DIGITALKER
(Includes parts for amplifier/filter in figure 4–4.)

Quantity

1 4″×4″ experimenters board (Radio Shack #276-159)
1 DT 1050 DIGITALKER chip set
1 LM 741 OPAMP
1 50 PIV-lamp bridge rectifier
1 1000-μF/35V electrolytic
1 9V 1W zenner diode
3 .1-μF capacitors
2 24-pin wire-wrap DIP sockets
1 40-pin wire-wrap DIP sockets
1 8-pin wire-wrap DIP sockets
1 4-mHz crystal
1 50-Pf capacitor

FIGURE 4–3
Connections between the SPC and the two speech ROMS.

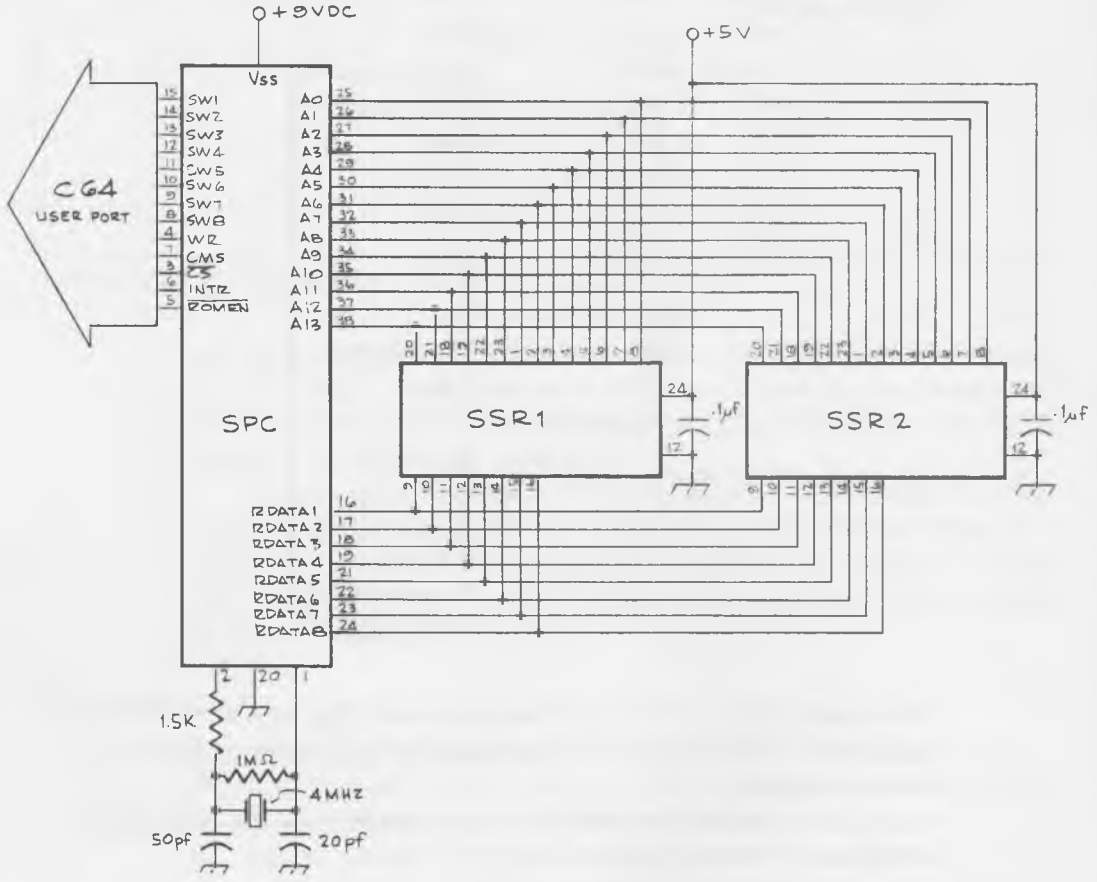


FIGURE 4–4
Amplifier/filter for the speech processor, which connects to the audio/video plug.

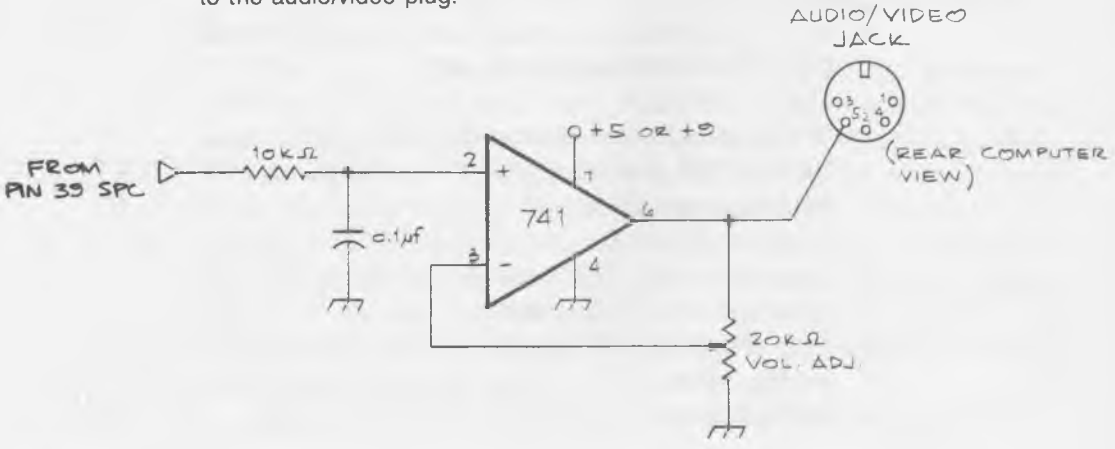


FIGURE 4–5
Amplifier/filter circuit that will drive a speaker directly.

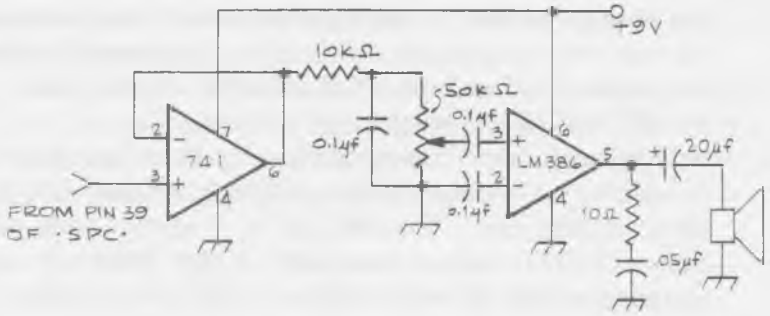
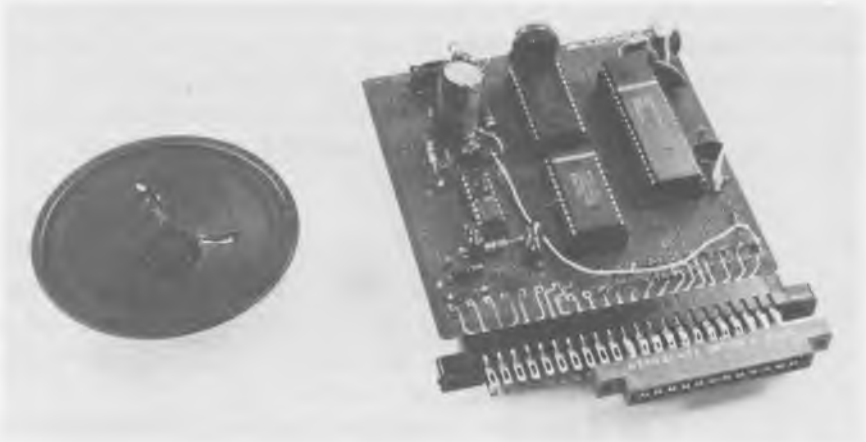


FIGURE 4–6
The completed DIGITALKER project.



1 20-Pf capacitor
1 1-Meg ¼W resistor
1 68-ohm ¼W resistor
2 10K ¼W resistor
1 20K trimpot
1 1.5K ¼W resistor
1 DIN plug for the audio/video socket

Checkout

When all of the connections are completed and your wiring has been doublechecked, plug the board into the user port. *Do not insert any ICs at this time.* Connect the negative lead of a volt-ohm meter to a ground on your board. Set the meter to 25 volts DC full-scale. Turn on the Commodore 64's power and confirm that the Commodore 64 is operating

properly. If not, look for shorts on either the +5 volt or the 9VAC connections. Next, verify that you have +5 volts on pin 24 of both ROM sockets. You should also see about +9 volts on pin 40 of the SPC socket, and on pin 7 of the 741 socket. Now turn off the Commodore 64. Set the volt-ohm meter to ohms × 1. Zero ohms should be observed when the probe is touched to pin 12 of the ROM sockets; pins 3, 7, and 20 of the SPC socket; and pin 4 of the 741 socket.

When all these voltages have been verified, turn off the Commodore 64 and plug the five ICs into their sockets. Note that the two ROM sockets are identically wired. The SSR1 ROM is enabled when pin 20 is low, whereas SSR2 is enabled when pin 20 is high. Thus they can be plugged into either socket. Set the volume control trimpot to a middle position and insert the audio/video plug into its socket on the rear of the Commodore 64. Turn the Commodore 64 back on and turn up the volume on the monitor. Now enter the following program.

```
10 POKE 54296, 15
20 POKE 56579, 255
30 POKE 56577, J
40 FOR I = 1 to 2000: NEXT I
50 GOTO 30
```

Type RUN. You should hear DIGITALKER say "This is DIGITALKER" over and over again. If you do not hear anything, verify that you have pulses on WR with a logic probe. If pulses are present, see if INTR is switching from a high to a low with each cycle. If that is not the case, look for clock pulses on pin 2 of the SPC.

If all of the above checks out, but you still have no response, you may have a wiring error in the audio amplifier. Turn off the Commodore 64 and remove the SPC chip. With power reapplied, touch pin 39 of the SPC socket with a dressmaker's pin held in your fingers. That should cause a 60cycle hum in the television speaker. If that does not occur, look for bad wiring or components in the audio amplifier section. For test purposes, you can jumper the output of the SPC (pin 39) directly to pin 5 of the audio/video socket to confirm if the SPC is producing any speech at all.

Programming the DIGITALKER

Once the DIGITALKER has passed these tests, you are ready to program it. Examine the program that you entered for the checkout. Line 10 enables the sound chip in the Commodore 64. Line 20 sets the data direction register for port B to all outputs. In line 30 we output the value of J through port

B to the SW lines. Since J has a default value of 0 in commodore 64's BASIC, all of the SW lines will be held low. Writing to port B automatically lowers PC2 for one microsecond to latch the SW data, and to start the speech sequence. In line 40 we delay for several seconds, and in 50 the program goes back to repeat the sequence.

It is possible to have the Commodore 64 actually sense when the word is finished rather than to simply delay. This will allow you to send words of varying length and have proper timing between phrases. To accomplish this, add the following commands:

```
15 POKE 56578, 59
40 IF (4 AND PEEK (56576)) = 0 THEN 40
```

Now bit 2 of Part A will be tested to see if PA2 is low. If it is low, the word is not finished and the program will loop back and test it again and again until PA2 is returned to the high state on completion of the phase. Run the program and verify that the words now repeat themselves with no pause between repetitions. Lines 30–40 followed by a return make a useful subroutine for outputting a word (see Listing 4–1).

The value of J determines the word to be spoken from the master

LISTING 4–1
Speak a number

```
1 REM ********************************
2 REM SPEAK A NUMBER FROM 0 TO 9999999
3 REM FOR THE DIGITALKER ON THE C64
4 REM ********************************
5 POKE 54296,15:POKE 56579,255:POKE 56578,59
10 INPUT Y
20 GOSUB 1000
30 GOTO 10
999 REM SUBROUTINE SPEAK NUMBER
1000 IF Y=0 THEN Z7=31:GOSUB 1200:RETURN
1005 Z6=INT(Y/1000000)
1010 IF Z6>999 THEN RETURN
1020 IF Z6<>0THEN Z8=Z6:GOSUB 1100:Z7=30:GOSUB1200
1030 Z5=Y-Z6*1000000:Z5=INT(Z5/1000)
1040 IF Z5<>0 THEN Z8=Z5:GOSUB 1100:Z7=29:GOSUB 1200
1050 Z8=Y-(1000000*Z6+1000*Z5)
1060 GOSUB 1100:RETURN
1099 REM SUBROUTINE NUM
1100 Z3= INT(Z8/100)
1110 Z4=Z8-Z3*100
1120 Z2=INT(Z4/10)
1130 Z1=Z4-Z2*10
1140 IF Z3<>0 THEN Z7=Z3:GOSUB 1200:Z7=28:GOSUB 1200
1150 IF Z2=1 THEN 1190
1160 IF Z2<>0 THENZ7=Z2+18:GOSUB 1200
1170 IF Z1<>0 THEN Z7=Z1:GOSUB1200
1180 RETURN
1190 Z7=Z1+10:GOSUB 1200:RETURN
1200 REM SPEECH SUBROUTINE
1210 POKE 56577,Z7
1220 IF (4 AND PEEK(56576))=0 THEN 1220
1230 RETURN
```

list (Table 4–1). Note that a value of 0 corresponds to the phrase "This is DIGITALKER." Enter the command:

```
25 J = 1
```

Now start the program. DIGITALKER should say "One." You can make DIGITALKER recite its entire vocabulary in sequence by adding the following code to your program and typing RUN.

```
25 FOR J = 0 TO 143
50 NEXT J
```

Making Intelligent Phrases

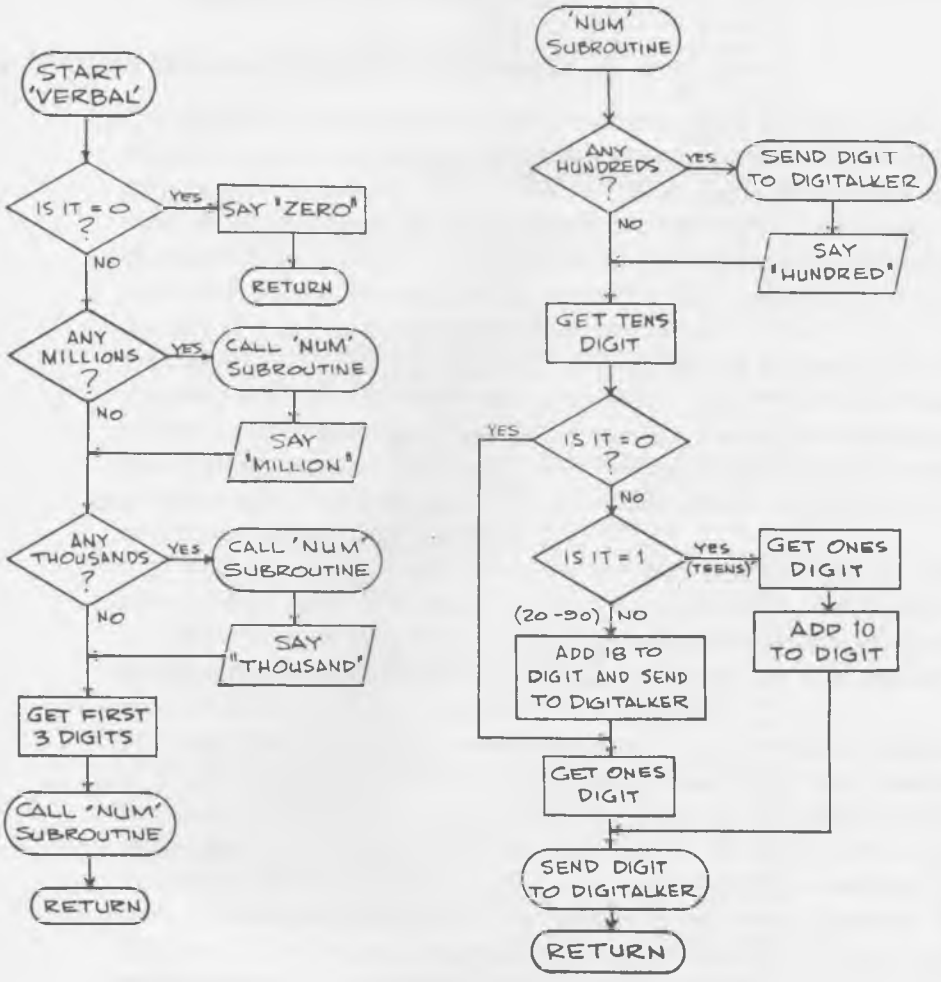
The usefulness of DIGITALKER lies in its programmability: Words can be linked together to make intelligent phrases and sentences. For example, let's have DIGITALKER create the phrase that says the value of a number. The flow chart in Figure 4–7 shows how you arrive at the proper sequence of words. The number 9,715,432 would be stated as nine million, seven hundred fifteen thousand, four hundred thirty two. The program first breaks the number into three-digit groups: The far-right three digits represent the ones and require no suffix, the next three digits represent the thousands and use the suffix "thousand," and the next three digits represent the millions and use the suffix "million." Note that saying a three digit number requires first identifying the far left digit and stating its value, if it is one through nine, followed by the word "hundred." The middle digit is a little more complicated. If it is two through nine, the word "twenty," "thirty," "forty," and so on, is stated followed by the value of the far right digit. If the middle digit is a one, then the number is a teen. In that case, the appropriate teen must be spoken.

Listing 4–2 codes the flow chart in Figure 4–7 into a BASIC subroutine starting at line 1000. The subroutine at 1100 arranges the words for a three-digit number and the subroutine at 1200 sends the codes to the DIGITALKER. The program, when executed, asks for a number with a "?." It will then state, in English, the value of any number you type in between 0 and 999,999,999. Listing 4–2 shows how the subroutine in Listing 4–1 can be incorporated into an applications program—in this case, a talking version of the LUNAR LANDER game.

More Vocabulary

With a little imagination, you can create a whole host of sentences from the standard vocabulary provided in DIGITALKER'S two ROMs.

FIGURE 4–7
Flow chart for a program that can state any number between 0 and 999,999,999. VERBAL is the main entry point. NUM is a subroutine that says a three-digit number and is called by VERBAL.



Remember that words like "to," "a," "for," "be," and so forth can be found in the number and alphabet list as synonyms. Nevertheless, certain applications will require a larger vocabulary. Other ROMs are available from National Semiconductor with specialized vocabularies. Furthermore, you can program your own ROMs using the DTSW500 development kit. This consists of a CP/M-compatible floppy disk containing speech data for over 1000 words and a program for packing user-selected words into a data array for ROM storage. Thus, access to a CP/M-based system having an 8″ disk drive (the Commodore 1541 drive only accepts 5″ disks) and

LISTING 4–2
Lunar Lander

```
1 REM ****************************
2 REM TALKING LUNAR LANDER FOR C64
3 REM ****************************
4 PRINT"■ENTER ROCKET BURNS SO THAT YOU LAND "
5 PRINT"AT 5 F/S OR LESS."
6 PRINT"HIT RETURN TO START":INPUT A$
9 POKE 54296,15:POKE 56579,255:POKE 56578,59
10 H=5000
20 V=300
30 F=300
40 IF F=<0 THEN B=0:F=0:GOTO 60
45 PRINT "■ENTER BURN (0-20)";
50 INPUT B
55 IF B>20 OR B<0 THEN GOTO 45
56 REM CALCULATE V,H,F(VELOCITY,HEIGHT,FUEL)
57 PRINT"■";
60 V=V+(5-B)*3
70 H=H-V
75 F=F-B:IF F<0 THEN F=0
76 IF INT(H) =<0 THEN GOTO 300:REM LANDED?
77 Z7=133:GOSUB 1200:Z7=80:GOSUB 1200
78 IF V<0 THEN Z7=109:GOSUB 1200:REM V NEGATIVE?
80 Y=ABS(INT(V)):GOSUB 1000
82 Z7=82:GOSUB 1200:Z7=32:GOSUB 1200
84 Z7=130:GOSUB 1200
90 Y= INT(H):GOSUB 1000
92 Z7=82:GOSUB 1200:Z7=91:GOSUB 1200:Z7=68:GOSUB 1200
93 IF F<50 THEN Z7=76:GOSUB 1200
97 Z7=84:GOSUB 1200:Z7=80:GOSUB 1200
98 REM READ THE PARAMETERS
100 Y=INT (F):GOSUB 1000
105 Z7=85:GOSUB 1200
110 GOTO 40
300 IF V<10 THEN Z7=65:GOSUB 1200
310 IF V<10 THEN PRINT "■YOU LANDED SAFELY":GOTO 500
320 PRINT"■YOU CRASHED"
330 REM CRASH
400 Z7=75:GOSUB1200:Z7=81:GOSUB1200:Z7=65:GOSUB 1200
410 Z7=79:GOSUB1200:Z7=61:GOSUB1200:Y=V:GOSUB 1000:
420 Z7=82:GOSUB1200:Z7=32:GOSUB1200:Z7=130:GOSUB 1200:END
500 REM SAFE LANDING
505 Z7=66:GOSUB1200:GOSUB1200
510 Z7=79:GOSUB1200:Z7=46:GOSUB1200:Z7=42:GOSUB1200
520 END
999 REM CALCULATE NUMBERS
1000 IF Y=0 THEN Z7=31:GOSUB 1200:RETURN
1005 Z6=INT(Y/1000000)
1010 IF Z6>999 THEN RETURN
1020 IF Z6<>0THEN Z8=Z6:GOSUB 1100:Z7=30:GOSUB1200
1030 Z5=Y-Z6*1000000:Z5=INT(Z5/1000)
1040 IF Z5<>0 THEN Z8=Z5:GOSUB 1100:Z7=29:GOSUB 1200
1050 Z8=Y-(1000000*Z6+1000*Z5)
1060 GOSUB 1100:RETURN
1100 Z3= INT(Z8/100)
1110 Z4=Z8-Z3*100
1120 Z2=INT(Z4/10)
1130 Z1=Z4-Z2*10
1140 IF Z3<>0 THEN Z7=Z3:GOSUB 1200:Z7=28:GOSUB 1200
1150 IF Z2=1 THEN 1190
1160 IF Z2<>0 THENZ7=Z2+16:GOSUB 1200
1170 IF Z1<>0 THEN Z7=Z1:GOSUB1200
1180 RETURN
1190 Z7=Z1+10:GOSUB 1200:RETURN
1200 POKE 56577,Z7
1210 IF(4 ANDPEEK(56576))=0 THEN 1210
1220 RETURN
```

a ROM burner will allow you to create your own custom vocabulary of up to 256 words for a specific DIGITALKER application. The development system currently sells for about $250 and can be purchased from Time Domain Systems, 840 Sandy Cove Rd., Rodeo, CA 94572.

The SP0256 Narrator Speech Processor

The SP0256 NARRATOR is a good example of the phoneme approach to speech processors. This chip contains a preprogrammed set of 64 speech parts called "allophones." Speech can be broken down into about 44 basic parts called phonemes. General Instruments has further subdivided this phoneme set into a larger set called allophones, depending on the phoneme's position in a word. For example, the phoneme DD is actually spoken differently if it is beginning the word, as in "done," than if it is used at the end of the word, as in "tide." The allophone set thus gives two versions—DD1 for the former use and DD2 for the latter. The net result of this is that a much more intelligible speech is created with allophones than with phonemes. The speech quality of the NARRATOR chip may not be as good as that produced by the DIGITALKER, but it has one big advantage—an unlimited vocabulary.

Recently, Radio Shack has begun to market the SP0256–AL2 at a price of well under $15, which makes it a super bargain. Furthermore, the SP0256 is easily interfaced to the Commodore 64. In fact, except for the speech processor chip itself, the only additional parts needed are a 3.12-mHz crystal and a couple of 22-pf capacitors.

The SP0256 refers to a family of integrated circuits consisting of an internal 2K by 8-bit ROM and an amplitude-modulated, variable-frequency generator (see Figure 4–8). Only those SP0256 chips with the suffix AL2 have the allophone set in the ROM. The Radio Shack part number for this version is 276–1784. The reader should be cautioned that other versions are to be found in the marketplace. For example, the SP0256–017 only has the numbers 0 through 11 in the ROM. Before you purchase a chip, be sure it has the AL2 suffix, as Radio Shack sells both versions!

Included in Radio Shack's NARRATOR package is a very informative booklet that explains how to construct speech from allophones. Of particular interest is the dictionary of several hundred words. For example, the word "infinitive" is composed of the allophone sequence: IH, NN1, FF, FF, IH, IH, NN1, IH, PA2, PA3, TT2, IH, VV.

We found the allophone programming very easy to learn, and after about 20 minutes of playing with the NARRATOR speech processor on the Commodore 64 we found that we could code about any word we tried. Table 4–2 gives a complete list of the allophones found in the NAR-

FIGURE 4–8
Chip data for the NARRATOR speech processor.
Courtesy of Radio Shack.

Catalog Number 276-1784

ARCHER®

TECHNICAL DATA

AN EXCLUSIVE RADIO SHACK SERVICE TO THE EXPERIMENTER

SP0256 NARRATOR™ SPEECH PROCESSOR

Features

- Natural Speech
- Stand Alone Operation with Inexpensive Support Components
- Wide Operating Voltage
- Word, Phrase, or Sentence Library, ROM Expandable
- Expandable to 491K of ROM Directly
- Simple Interface to Most Microcomputers or Microprocessors
- Supports L.P.C. Synthesis: Formant Synthesis: Allophone Synthesis

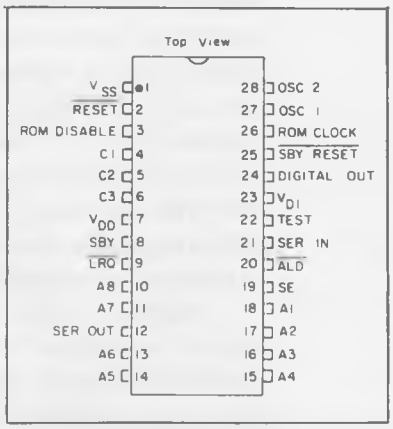
General Description

The SP0256 (Speech Processor) is a single chip N-Channel MOS LSI device that is able, using its stored program, to synthesize speech or complex sounds.

The achievable output is equivalent to a flat frequency response ranging from 0 to 5 kHz, a dynamic range of 42dB, and a signal to noise ratio of approximately 35dB.

The SP0256 incorporates four basic functions:

- A software programable digital filter that can be made to model a VOCAL TRACT.
- A 16K ROM which stores both data and instructions (THE PROGRAM).
- A MICROCONTROLLER which controls the data flow from the ROM to the digital filter, the assembly of the "word strings" necessary for linking speech elements together, and the amplitude and pitch information to excite the digital filter.
- A PULSE WIDTH MODULATOR that creates a digital output which is converted to an analog signal when filtered by an external low pass filter.



PIN CONFIGURATION

Allophone Based Speech Processor – SP0256-AL2

One example of a preprogramed SP0256 is the AL2 pattern.

Allophone Usage with a Microprocessor

The SP0256-AL2 requires the use of a processor to concatenate the speech sounds to form words.

The SP0256 is controlled using the address pins (A1-A8), ALD (Address Load), and SE (Strobe Enable). The object for controlling the chip is to load an address into it which contains the desired allophone. The speech data for the allophone set is contained within the internal 16K ROM of the SP0256-AL2.

CUSTOM PACKAGED IN U.S.A. BY RADIO SHACK A DIVISION OF TANDY CORPORATION

TABLE 4-2: Allophone Codes (in Decimal)

00	PA1	PAUSE	10MS
01	PA2	PAUSE	30MS
02	PA3	PAUSE	50MS
03	PA4	PAUSE	100MS
04	PA5	PAUSE	200MS
05	/OY/	B*oy*	420MS
06	/AY/	Sk*y*	260MS
07	/EH/	*E*nd	70MS
08	/KK3/	*C*omb	120MS
09	/PP/	*P*ow	210MS
10	/JH/	Do*dge*	140MS
11	/NN1/	Thi*n*	140MS
12	/IH/*	S*i*t	70MS
13	TT2/	*To*	140MS
14	/RR1/	*R*ural	170MS
15	/AX/*	S*ucc*eed	70MS
16	/MM/	*M*ilk	180MS
17	/TT1/	Par*t*	100MS
18	/DH1/	*Th*ey	290MS
19	/IY/	S*ee*	250MS
20	/EY/	B*ei*ge	280MS
21	/DD1/	Coul*d*	70MS
22	/UW1/	T*o*	100MS
23	/AO/*	*Au*ght	100MS
24	/AA/*	H*o*t	100MS
25	/YY2/	*Y*es	180MS
26	/AE/*	H*a*t	120MS
27	/HH1/	*He*	130MS
28	/BB1/	Ri*b*	80MS
29	/TH/*	*Th*in	180MS
30	/UH/*	B*oo*k	100MS
31	/UW2/	F*oo*d	260MS
32	/AW/	*Ou*t	370MS
33	/DD2/	*D*o	160MS
34	/GG3/	Wi*g*	140MS
35	/VV/	*V*est	190MS
36	/GG1/	*G*ot	80MS
37	/SH/	*Sh*ip	160MS
38	/ZH/	A*zu*re	190MS
39	/RR2/	B*r*ain	120MS
40	/FF/*	*F*ood	150MS
41	/KK2/	S*k*y	190MS

TABLE 4-2: Allophone Codes (in Decimal) (Continued)

42	/KK1/	*C*ant	160MS
43	/ZZ/	*Z*oo	210MS
44	/NG/	A*nch*or	220MS
45	/LL/	*L*ake	110MS
46	/WW/	*W*ool	180MS
47	/XR/	Hai*r*	360MS
48	/WH/	*Wh*en	200MS
49	/YY1/	*Y*es	130MS
50	/CH/	*Ch*urch	190MS
51	/ER1/	Lett*er*	160MS
52	/ER2/	F*ir*	300MS
53	/OW/	B*eau*	240MS
54	/DH2/	Ba*th*e	240MS
55	/SS/*	Ve*s*t	90MS
56	/NN2/	*N*o	190MS
57	/HH2/	*H*oe	180MS
58	/OR/	St*or*e	330MS
59	/AR/	F*ar*m	290MS
60	/YR/	Cl*ear*	350MS
61	/GG2/	*G*uest	40MS
62	/EL/	Sadd*le*	190MS
63	/BB2/	*B*usiness	50MS

*These can be doubled to prolong the sound.

RATOR'S ROM and their addresses. Figure 4–8 shows the pin assignments for the SP0256–AL2 chip.

Building the Interface

Figure 4–9 shows how we interfaced the NARRATOR chip to the Commodore 64. The six low-order bits of port B on the user port are connected to the six address bits, A1–A6. The binary code on these six lines selects one of the 64 allophones listed in Table 4–2.

The address is loaded and the speech begun whenever the ADL line is brought low. When the speech is begun, LRQ goes low and stays low until the NARRATOR is ready to accept the next address. ADL is connected to PA2 and LRQ is sensed by FLAG2. Thus, the normal sequence of operation is to POKE the address to port B and then pulse PA2 by POKEing it low, then high again. Finally, the program must test the FLAG bit in the 6522's interrupt register to see if the chip is ready for the next address. The output of the NARRATOR is identical to that of the DIGITALKER

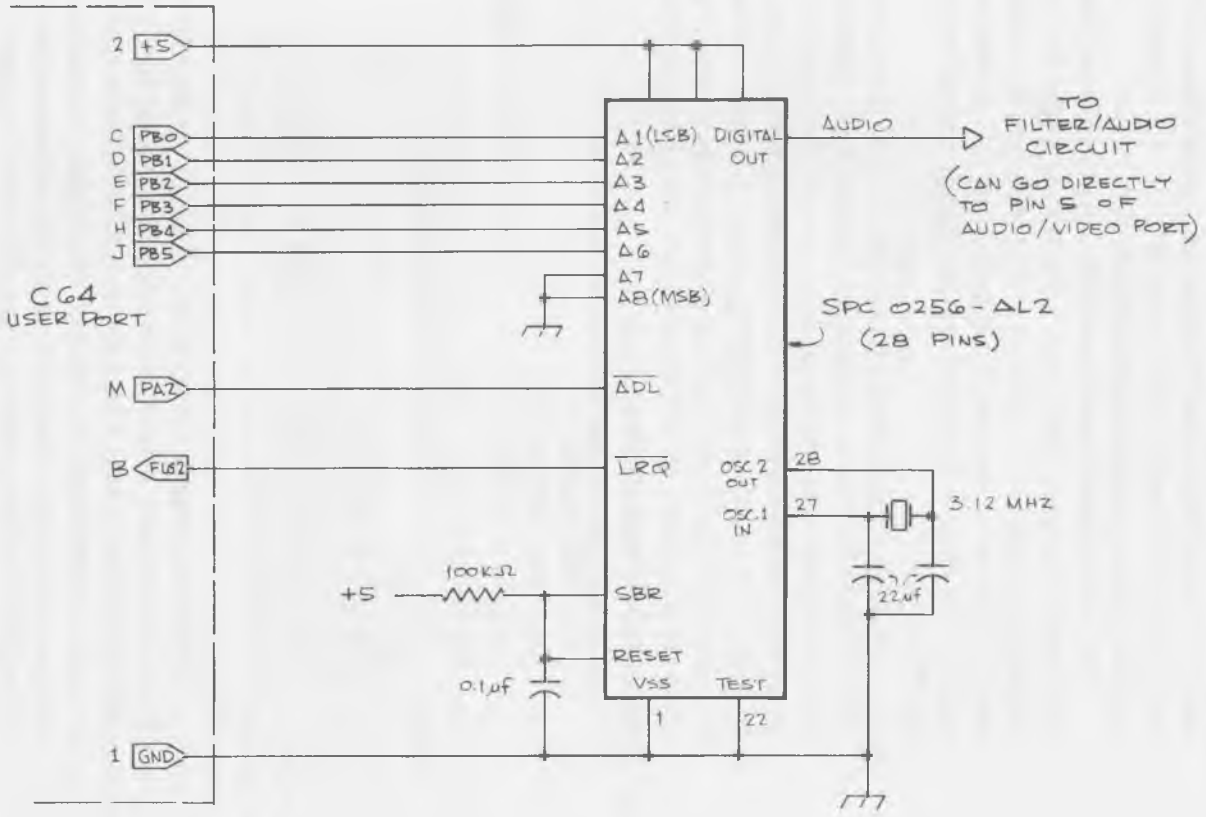


FIGURE 4–9
Connections between the user port and the NARRATOR chip.

and should be filtered for best results. Speech output from pin 24 can either be: 1), input directly to pin 5 of the Commodore 64's audio-video connector; 2), passed through the filtered op-amp circuit shown in Figure 4–4 and then to the audio/video connector; or 3), connected to the amplifier/speaker circuit shown in Figure 4–5.

We built the circuit on a 4-inch by 5-inch experimenter's board using wire-wrap techniques. The layout is not critical. Make the connections as shown in Figure 4–9. Although a 3.12-mHz crystal is specified, we also tried the more common 3.579-mHz color TV crystal and had no trouble. The pitch of the speech will be slightly higher, however, with the TV crystal.

When the board is wired, remove the SP0256 from its socket and plug the board into the Commodore 64. Turn on the computer and verify that the computer is functioning and has a cursor. Next, connect the common pin of a voltmeter to pin 1 of the user port (ground). Test for 0 volts at pins 1, 2, 22, and 25 of the SP0256's socket. Similarly, +5 volts should be seen on pins 7, 23, and 19. If all of these voltages check, turn off the computer and put the SP0256 in its socket. Turn the power back on and enter the following program:

```
 10 POKE 54296, 15
 20 POKE 56579,255
 30 DATA 27,24,6,3, – 1
 40 READ X
 50 IF X>0 THEN GOSUB 200:GOTO40
 60 END
200 POKE 56577,X
210 POKE 56576,147:POKE 56576,151
220 IF PEEK(56589)< >0 THEN RETURN
230 GOTO 220
```

When the program is run the NARRATOR should have said "Hi" to you. If it does not, make the following tests with your logic probe. ADL should be high and briefly show pulses when the RUN command is entered. A similar pattern should be seen on LRQ. Failure to see pulses on ADL means either a wiring error between pin M on the user port and pin 20 of the SP0256, a programming error, or a bad CIA chip in your Commodore 64. If pulses are seen on ADL but not on LRQ, the SP0256 chip is not working. Check pin 28 with the logic probe to verify that clock pulses are present. Miswiring of the crystal circuit, a bad SP0256, or a defective crystal could explain a lack of clock pulses. Finally, check your amplifier-filter circuit. If all the above checks out but no sound is heard,

the filter can be bypassed by connecting pin 24 of the SP0256 directly to pin 5 of the audio/video connector. If speech is now heard, your amplifier is not working.

PARTS LIST FOR THE NARRATOR
(With amplifier shown in Figure 4–4)

Quantity

1 SP0256–AL2
1 3.12-mHz crystal
2 0.1-μF capacitor
1 100-K/¼-watt resistor
1 4½" x 4" experimenter's board, e.g. Radio Shack #267-152
2 22 pf ceramic capacitors
1 10-K/¼ watt resistor
1 7.41 op-amp
1 40-pin wire-wrap socket
1 8-pin wire-wrap socket
1 20-K trim pot
1 DIN plug to fit the audio/video socket (a standard pin plug can be filed down to fit into pin 5 of the socket)

Programming the NARRATOR

Programming the NARRATOR is much more tedious than is programming the DIGITALKER. For example, the word "hundred" in the DIGITALKER requires that only one code, a decimal 28, be sent to it. The NARRATOR, on the other hand, needs a sequence of 11 allophones to say the same word. On the positive side, however, the DIGITALKER cannot say "Commodore 64" but the NARRATOR can. Because the allophones should be spoken in a smooth sequence, one blending into the next, care must be taken to send them in an uninterrupted stream. Furthermore, you must end each sequence with a pause as most of the allophones end with a continuous tone. The program in Listing 4–3 is an example of how this timing can be accomplished. The subroutine at 200 sends the code in the variable x to the NARRATOR and returns to the calling program only when the chip is ready to receive the next code. The timing is accomplished by testing for a 16 in the interrupt register. The allophones are placed in sequence in DATA statements ahead of the main program, which starts at line 500. The DATA in line 500 is a 3 for a pause to terminate the speech and a –1 to signify the end of the sequence. (Note that lines 15 and 20 set up a low-pass filter in the SID chip to improve sound quality.) The sequence in the example says: "Hello—I am the

LISTING 4–3
Simple program for NARRATOR

```
5 REM SET UP SID WITH A LOW-PASS FILTER
10 REM CUT-OFF FREQUENCY AT 6KHZ
15 POKE 54296,31 :POKE 54295,0
20 POKE 54294,50:POKE 54293,0
25 POKE 56579,255
30 REM DATA FOR HELLO, I AM THE NARRATOR
35 DATA 27,7,45,15,53,3,3,3
40 DATA 24,6,3,7,7,16,3
45 DATA 18,15,15,15,3,56,26,51,20
50 DATA 13,58
55 DATA 3,-1
60 READ X
65 IF X>0 THEN GOSUB 200:GOTO55
70 END
200 REM SPEAK AN ALLOPHONE SUBROUTINE
210 POKE 56577,X
220 POKE 56576,147:POKE 56576,151
230 IF PEEK(56589)<>0THEN RETURN
240 GOTO 230
```

NARRATOR.'' By putting your own codes into the data statements you can create your own sentences.

A more elaborate use of the NARRATOR system appears in Listings 4–4 and 4–5. The program in Listing 4–4 is used to create a custom vocabulary file on the disk. When you RUN it, you simply supply it with the sequence of allophone mnemonic codes required for each word. The program looks up the mnemonic in its data array and determines the numerical code for that mnemonic. After the last mnemonic in each word,

LISTING 4–4
Make a file

```
1 REM THIS PROGRAM CREATES A VOCABULARY FILE
2 REM ENTER THE ALLOPHONE NEUMONICS AND IT WILL PUT THEIR
3 REM CODES INTO A DATA FILE
4 REM AFTER EACH WORD TYPE END AN AFTER THE LAST WORD TYPE END, RETURN, QUIT.
5 REM THE FILE IS ORGANIZED WITH ALLOPHONE CODES WITH 64S BETWEEEN WORDS
6 REM THE END OF THE FILE CONTAINS A -1
10 PRINT "FILE NAME";:INPUT B$
12 OPEN 8,8,8,"0:"+B$+",S,W" :REM OPEN FILE
13 DIM A$(66)
15 DATA PA1,PA2,PA3,PA4,PA5,OY,AY,EH,KK3,PP,JH,NN1,IH,TT2,RR1,AX,MM
20 DATA TT1,DH1,IY,EY,DD1,UW1,AO,AA,YY2,AE,HH1,BB1,TH,UH,UW2,AW,DD2,GG3
30 DATA VV,GG1,SH,ZH,RR2,FF,KK2,KK1,ZZ,NG,LL,WW,XR,WH,YY1,CH,ER1,ER2
40 DATA OW,DH2,SS,NN2,HH2,OR,AR,YR,GG2,EL,BB2,END,QUIT
50 FOR I=0 TO 65
60 READ A$(I)
70 NEXT I:REM PUT CODES IN ARRAY
80 INPUT B$
90 I=0
100 IF A$(I)=B$ THEN PRINT#8, I :PRINT I:     GOTO 120
110 IF I<64 THEN I=I+1:GOTO 100
111 IF B$="QUIT" THEN 130
115 PRINT "NOT FOUND"
120 GOTO 80
130 PRINT#8,-1
140 CLOSE 8
150 END
READY.
```

LISTING 4–5
Read the file

```
0 REM READ A VOCABULARY FILE AND SPEAK THE JTH WORD FROM IT
1 POKE 54296,15 :REM TURN ON SID CHIP
2 POKE 56579,255
3 DIM V(300),Z(30)
5 PRINT "FILE NAME";:INPUT B$
10 OPEN 8,8,8,"0:"+B$+",S,R":REM OPEN DATA FILE
13 I=0:J=1
20 INPUT#8,V(I) :IF V(I)=64 THEN Z(J)=I:J=J+1
30 IF V(I)<>-1 THEN I=I+1:GOTO 20
40 CLOSE 8
100 REM *** ARRAY IS BUILT****
110 REM
120 INPUT J :GOSUB 1000:GOTO 120:REM J IS THE WORD # TO BE SPOKEN
130 REM
140 REM *** SUBSTITUTE YOUR INTELLIGENT CODE FOR THAT BETWEEN 40 AND 1000 ***
150 REM
160 REM
1000 REM SPEAK THE JTH WORD IN THE LIST
1020 J=Z(J)
1030 X=V(J):   GOSUB 1230
1040 J=J+1:IF V(J)<>64 THEN 1030
1050 X=3:GOSUB 1230
1060 RETURN
1230 POKE 56577,X
1240 POKE 56576,147:POKE 56576,151
1250 IF PEEK(56589)<>0THEN RETURN
1260 GOTO 1250
READY.
```

enter an END. After the last mnemonic in the last word in the vocabulary list, enter an END and then a QUIT. The program packs the allophone codes into a sequential file with 65s between the individual words and –1 at the very end of the file. By changing the device number in the OPEN statement the file can be stored on tape instead of disk.

Listing 4–5 is a companion to the above program and reads the vocabulary file into a data array. The main body of the program, between lines 40 and 1000, calls the subroutine at 1000 which speaks the Jth word in the vocabulary list created by the program in Listing 4–4. The simple main program here asks for a number and speaks the corresponding word.

If a vocabulary list equivalent to the first 30 words of the DIGITALKER'S word list were created, then the code in lines 10–1180 of Listing 4–1 could replace the main program in 4–5 to make it say, in English, the value of a number. Note that the number of the word to be spoken is carried in variable Z7 in the program in 4–1. With a little imagination, you should be able to modify these programs to satisfy a wide range of specific speech applications.

5

MECHANICAL ACTUATORS

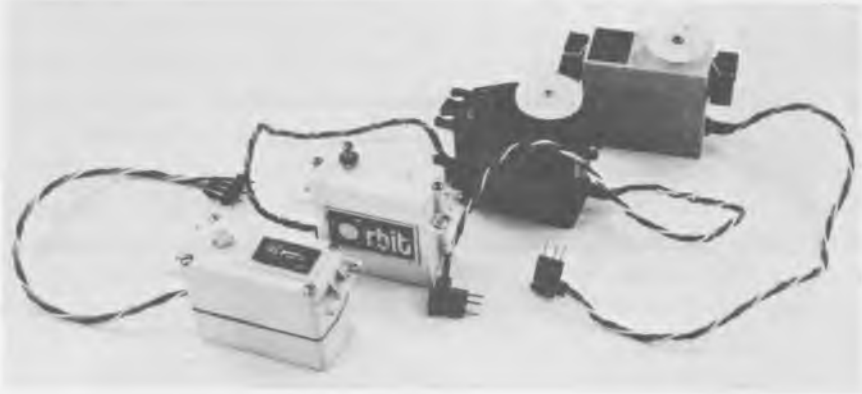
In certain applications, you may want the Commodore 64 to control a mechanical device. Computer-controlled mechanical actuators have been around almost as long as computers themselves. A digital x–y plotter and the head positioner on a disk drive are only two examples of computer-controlled mechanical devices. More recently, the growing field of robotics has sparked a new interest in mechanical actuators among industrial designers and hobbyists. In this chapter, we will show you how the Commodore 64 can be interfaced and programmed to control both the analog servo actuator and a stepping motor.

The Servo Actuator

Figure 5–1 shows a servo actuator of the type used in radio-controlled models. These actuators have been available for over a decade, operate on TTL logic, and are ideally suited for computer interfacing. They offer high torque, fast response time, a high degree of resolution, and excellent repeatability. Furthermore, they are quite inexpensive, ranging from $10 to $30 depending on the quality required.

The servo actuator (or simply servo) has a shaft protruding from the case that can rotate through about a 150° range. The servo is controlled by a pulse-width detector inside the case. The circuit determines the duration

FIGURE 5–1
Model airplane servo of the type used in this project.



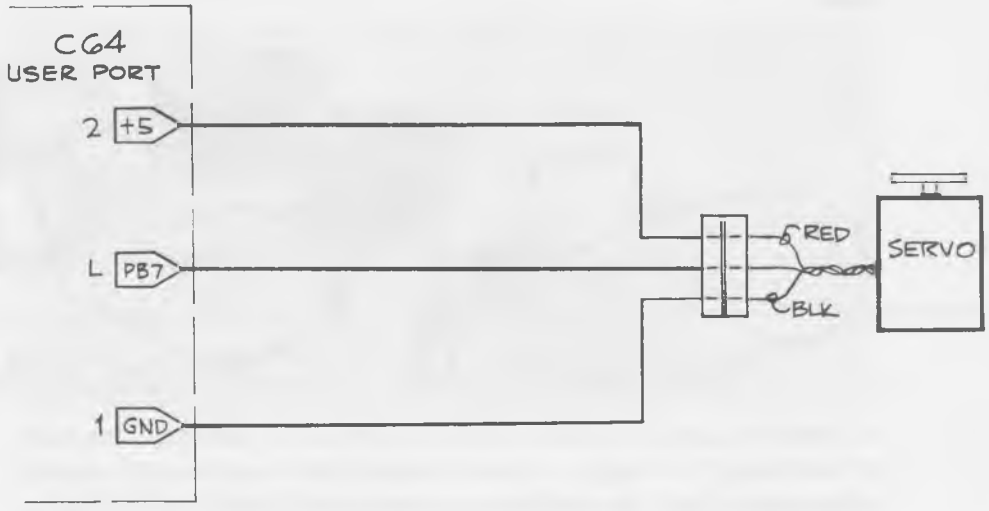
of a positive-going TTL pulse. It then determines the position of the shaft by examining the voltage on a potentiometer that is mechanically coupled to the output shaft. An electric motor moves the output shaft through a gear train until the potentiometer is at a position commensurate with the pulse width. It uses a true negative feedback, or servo-nulling, system, hence the name "servo." Servos come with a selection of output arms so that they can easily be coupled to a wide variety of devices. A servo would be ideal for accurately controlling fluid flow by turning a valve, opening and closing the fingers on a mechanical hand, or, perhaps, just turning a rotary switch.

Servos are available at any hobby store. Any servo compatible with Kraft or Futaba-brand radios (and that includes almost all of them) will work with the interface we describe. We used a Futaba FP–S16 servo to develop our interface.

The Interface

The servo is controlled by pulse-width modulation. About 60 times per second, it must be sent a positive-going TTL pulse. If the pulse stays high for one millisecond, the servo will center. Reducing the width to .5 milliseconds will cause the servo to deflect full scale in one direction, and increasing the duration to 1.5 ms causes the servo to deflect full scale in the opposite direction. The servo can be positioned anywhere in-between these two extremes by providing the appropriate intermediate pulse width. The Commodore 64 is ideally suited for a servo because you can use the 6526 CIA chip's built-in timer available on pin K of the user port. In the one-shot mode, timer A causes output PB6 to go high for a preset period and then return to the low state. The only hardware required is to connect the Commodore 64's pulse to the servo as shown in Figure 5–2.

FIGURE 5-2
Connection between the servo and the user port.



Your servo will have three wires coming out of it. The red wire is the power lead and requires +5 volts, found in pin 2 of the user port connector. The black wire is the ground and should connect to pin 1 on the user port. The third wire, which may be any color, is the signal line and should be connected to PB6 on pin K of the user port. The hobby shop will be able to supply you with a miniature socket to match the plug on the servo.

The Software

You can control the servo entirely from BASIC. Remember, however, that the signal pulse must be refreshed at about 30 to 60 times per second. The faster the refresh time the more briskly the servo will respond. The following program moves the servo in proportion to the number key pressed.

```
10 J = 4
20 POKE 56589,3
30 POKE 56580, 255
40 POKE 56581, J
50 POKE 56590, 15
60 GET A$: IF A$ = "" THEN 50
70 J = VAL (A$): GOTO 40
```

Line 10 of this program sets an initial value for J to the high-order 8 bits of Timer A. Line 30 puts a 255 in the 8 low-order bits. Thus, to start,

the timer is set to count 4*255 or 1024 cycles of the 6510's clock. Since the system clock runs at 1.02 megahertz, this will be 1024 microseconds or about a one-millisecond-long pulse. One millisecond, you will recall, was the pulse width required to center the servo. Line 50 sends a 15 to control register A to start Timer A in the one-shot mode and with PB6 output enabled. Finally, line 20 disables any interrupts from Timer A or B by masking out the appropriate bits in the interrupt register. Line 60 checks to see if a new number has been entered and, if not, loops back to line 50 to refresh the pulse. Striking the number keys from 1 to 7 should exercise the servo. This simple program changes the servo's signal only in increments of 256 microseconds. This control can be made to be in increments of one microsecond by the following additions to the program.

```
10 PRINT "TIME (500-1500)";
12 INPUT P
14 H=INT (P/256)
16 L=P-H*256
30 POKE 56580, L
40 POKE 55581, H
70 GOTO 10
```

When you run this program, it will ask for a number between 500 and 1500. Enter a number and hit return. The servo will move to the appropriate position. Strike any key to get the Commodore 64 to ask for a new number. Lines 14 and 16 break the number P into two 8-bit bytes—H, the high-order byte, and L, the low-order byte—for insertion into the timer registers.

Using the Servo Under Interrupt Control

The BASIC program given in the preceding sections will drive the servo, but notice that refreshing the servo pulse occupies BASIC almost completely. It would be nice if we could make an automatic system wherein BASIC had only to specify the position of the servo once, and thereafter the refresh would be automatic. This can be accomplished by putting a machine language pulse-generating program into the interrupt service routine of the Commodore 64. Every 60th of a second, the 6510 is interrupted and jumps to a special routine to see if a key has been depressed. The address, or vector, for that service routine is held at memory locations $0314 and $0315. Since 60 Hz is a perfect refresh rate for the servo, we will change the interrupt vector to point to the pulse-generating routine. In turn, this routine will transfer control to the normal interrupt routine when it is finished. The routine gets its position value from ad-

dress $00FB (251 decimal). $FB is an unused location on page 0 in the Commodore 64. Thus, all the BASIC program must do is POKE a number from 100 to 255 in address 251 and the servo will automatically move to that position and stay there until a different number is placed in 251. This will all be transparent to the BASIC user. The pulse-generator program appears in Listing 5–1.

LISTING 5–1
Servo program

```
LINE# LOC   CODE        LINE

00001  0000              ; PULSE-GENERATOR PROGRAM
00002  0000              *=$C000
00003  C000              ;-------------------------
00004  C000              ; EQUATES
00005  C000              IRQVEC=$0314            ;IRQ VECTOR
00006  C000              TMRAH=$DD05             ;TIMER HIGH BYTE
00007  C000              TMRAL=$DD06             ;TIMER LOW BYTE
00008  C000              CIAIR=$DD0D             ;CIA INTR. REG
00009  C000              CIACR=$DD0E             ;CIA CONT. REG.
00010  C000              OLDIRQ=$EA31            ;OLD IRQ ADDRESS
00011  C000              ;-------------------------
00012  C000  08          SINT    PHP             ;SAVE STATUS
00013  C001  78                  SEI             ;INHIBIT IRQ
00014  C002  A9 14               LDA #$14        ;LOW BYTE OF START
00015  C004  8D 14 03            STA IRQVEC
00016  C007  A9 C0               LDA #$C0        ;HIGH BYTE OF START
00017  C009  8D 15 03            STA IRQVEC+1
00018  C00C  A9 03               LDA #$03        ;MASK OUT TIMERA INTERUPT
00019  C00E  8D 0D DD            STA CIAIR
00020  C011  28                  PLP
00021  C012  58                  CLI             ;ALLOW INTERUPTS
00022  C013  60                  RTS
00023  C014              ;-------------------------
00024  C014  48          START   PHA             ;SAVE AC
00025  C015  A9 00               LDA #0          ;ZERO AC
00026  C017  85 FC               STA $FC         ;ZERO FC
00027  C019  A5 FB               LDA $FB
00028  C01B  18                  CLC
00029  C01C  0A                  ASL A           ;GENERATE TWO
00030  C01D  26 FC               ROL $FC         ;BYTE WORD FOR
00031  C01F  0A                  ASL A           ;THE TIMER
00032  C020  26 FC               ROL $FC
00033  C022  0A                  ASL A
00034  C023  26 FC               ROL $FC
00035  C025  8D 06 DD            STA TMRAL
00036  C028  A5 FC               LDA $FC
00037  C02A  8D 05 DD            STA TMRAH
00038  C02D  A9 0F               LDA #$0F
00039  C02F  8D 0E DD            STA CIACR       ;START TIMER
00040  C032  68                  PLA             ;RESTORE AC
00041  C033  4C 31 EA            JMP OLDIRQ
00042  C036              .END

ERRORS = 00000

SYMBOL TABLE

SYMBOL VALUE
 CIACR   DD0E    CIAIR   DD0D    IRQVEC  0314    OLDIRQ  EA31
 SINT    C000    START   C014    TMRAH   DD05    TMRAL   DD06
```

A BASIC program that locates the pulse-generating code in RAM at $C000 appears in Listing 5–2. That program, on startup, reads the program from the data statements and POKEs it into memory. It then does a SYS to the routine that changes the interrupt vector. Thereafter, the servo is automatically positioned according to the byte stored at address 251, even if the BASIC program is not running. This mode of operation can be cancelled at any time by simply resetting the Commodore 64, which restores the interrupt vector but does not delete the BASIC program.

Adding a Second Servo

Timer B can be used to drive a second servo. The program in Listing 5–1 can be modified by duplicating the code between line 25 and line 39. Also, the register addresses must be changed from those for Timer A to those for Timer B (see Chapter 3). Listing 5–3 shows a basic program which POKEs this program into memory. The output of Timer B is found on PB7 (pin L) of the user port. Timer B's pulse in that program is controlled by the number stored at $FD (253 decimal).

It is also possible to add up to four servos, but to do that we can no longer use the Commodore 64's built-in timer. Instead we must take a software approach. Listing 5–4 presents a machine language program

LISTING 5–2
BASIC as 5–1

```
READY.


0 REM ***INTERUPT MODE SERVO DRIVER***
1 DATA 8 ,120 ,169 ,20
5 DATA 141 ,20 ,3 ,169
9 DATA 192 ,141 ,21, 3
13 DATA 169 ,3 ,141, 13
17 DATA 221 ,40 ,88 ,96
21 DATA 72, 169 ,0 ,133
25 DATA 252 ,165, 251, 24
29 DATA 10 ,38 ,252, 10
33 DATA 38 ,252 ,10, 38
37 DATA 252, 141, 6, 221
41 DATA 165, 252 ,141, 5
45 DATA 221, 169, 13, 141
49 DATA 14 ,221, 104 ,76
53 DATA 49 ,234
100 FOR J=49152 TO 49152+53
110 READ X
120 POKE J,X
130 NEXT J
135 REM***TIMER IS SET UP*****
136 REM***PUT YOUR CODE  *****
137 REM***FROM HERE ON   *****
140 SYS 49152
145 POKE 251,128
150 INPUT J
160 POKE 251,J
170 GOTO 150

READY.
```

LISTING 5-3
2 servos

```
LINE# LOC   CODE        LINE

00001  0000             ; PULSE-GENERATOR FOR 2 SERVOS
00002  0000             *=$C000
00003  C000             ;-------------------------
00004  C000             ; EQUATES
00005  C000             IRQVEC=$0314            ;IRQ VECTOR
00006  C000             TMRAH=$DD05             ;TIMER HIGH BYTE
00007  C000             TMRAL=$DD06             ;TIMER LOW BYTE
00008  C000             TMRBH=$DD07             ;TIMERB HIGH BYTE
00009  C000             TMRBL=$DD06             ;TIMERB LOW BYTE
00010  C000             CIAIR=$DD0D             ;CIA INTR. REG
00011  C000             CIACRA=$DD0E            ;CIA CONT. REGA
00012  C000             CIACRB=$DD0F            ;CIA CONT. REGB
00013  C000             OLDIRQ=$EA31            ;OLD IRQ ADDRESS
00014  C000             ;-------------------------
00015  C000  08        SINT   PHP              ;SAVE STATUS
00016  C001  78               SEI              ;INHIBIT IRQ
00017  C002  A9 14            LDA #$14         ;LOW BYTE OF START
00018  C004  8D 14 03         STA IRQVEC
00019  C007  A9 C0            LDA #$C0         ;HIGH BYTE OF START
00020  C009  8D 15 03         STA IRQVEC+1
00021  C00C  A9 03            LDA #$03         ;MASK OUT TIMERA INTERUPT
00022  C00E  8D 0D DD         STA CIAIR
00023  C011  28               PLP
00024  C012  58               CLI              ;ALLOW INTERUPTS
00025  C013  60               RTS
00026  C014             ;-------------------------
00027  C014  48        START  PHA              ;SAVE AC
00028  C015  A9 00            LDA #0           ;ZERO AC
00029  C017  85 FC            STA $FC          ;ZERO FC
00030  C019  A5 FB            LDA $FB
00031  C01B  18               CLC
00032  C01C  0A               ASL A            ;GENERATE TWO
00033  C01D  26 FC            ROL $FC          ;BYTE WORD FOR
00034  C01F  0A               ASL A            ;THE TIMER
00035  C020  26 FC            ROL $FC
00036  C022  0A               ASL A
00037  C023  26 FC            ROL $FC
00038  C025  8D 06 DD         STA TMRAL
00039  C028  A5 FC            LDA $FC
00040  C02A  8D 05 DD         STA TMRAH
00041  C02D  A9 00            LDA #$00         ;SETUP TIMERB
00042  C02F  85 FC            STA $FC
00043  C031  A5 FD            LDA $FD          ;FD IS CONTROL REG
00044  C033  18               CLC
00045  C034  0A               ASL A
00046  C035  26 FC            ROL $FC
00047  C037  0A               ASL A
00048  C038  26 FC            ROL $FC
00049  C03A  0A               ASL A
00050  C03B  26 FC            ROL $FC
00051  C03D  8D 06 DD         STA TMRBL
00052  C040  A5 FC            LDA $FC
00053  C042  8D 07 DD         STA TMRBH
00054  C045  A9 0F            LDA #$0F
00055  C047  8D 0E DD         STA CIACRA;START TIMERA
00056  C04A  8D 0F DD         STA CIACRB;START TIMERB
00057  C04D  68               PLA              ;RESTORE AC
00058  C04E  4C 31 EA         JMP OLDIRQ
00059  C051             .END
```

LISTING 5–4
4 servos

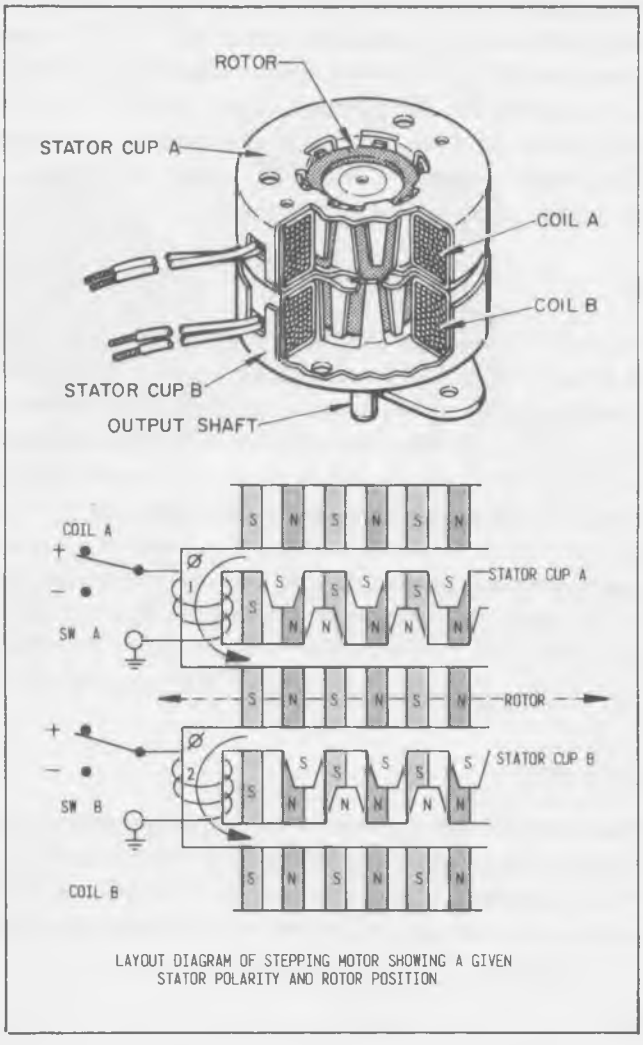
```
LINE# LOC   CODE        LINE

00001  0000             ;*********SERVO 4 *********
00002  0000             *=$C000
00003  C000             ; EQUATES:
00004  C000             IRQVEC=$0314
00005  C000             PORTB=$DD01
00006  C000             DDIR=$DD03
00007  C000             OLDIRQ=$EA31
00008  C000             ;-----------------------------
00009  C000  08         START  PHP
00010  C001  78                SEI              ;SAVE IRQ STATUS
00011  C002  A9 14             LDA #$14
00012  C004  8D 14 03          STA IRQVEC       ;CHANGE IRQ VECTOR
00013  C007  A9 C0             LDA #$C0
00014  C009  8D 15 03          STA IRQVEC+1
00015  C00C  A9 FF             LDA #$FF
00016  C00E  8D 03 DD          STA DDIR         ;CHANGE DATA DIR
00017  C011  28                PLP
00018  C012  58                CLI              ;ENABLE INTERUPTS
00019  C013  60                RTS
00020  C014             ;-----------------------------
00021  C014  08         ENTER  PHP              ;SAVE REGISTERS
00022  C015  48               PHA
00023  C016  8A               TXA
00024  C017  48               PHA
00025  C018  98               TYA
00026  C019  48               PHA
00027  C01A  A9 FF             LDA #$FF
00028  C01C  8D 57 C0          STA MASK         ;START ALL OUTPUTS
00029  C01F  8D 01 DD          STA PORTB        ;HIGH
00030  C022  A0 10             LDY #$10         ;Y IS THE COUNTER
00031  C024  A2 04       LOOP3 LDX #$04         ;X IS THE CHANNEL#
00032  C026  98           LOOP2 TYA
00033  C027  DD 5B C0          CMP COUNT-1,X
00034  C02A  B0 20             BCS WASTE        ;TIME OUT YET?
00035  C02C  AD 57 C0          LDA MASK
00036  C02F  3D 57 C0          AND TABLE-1,X
00037  C032  8D 01 DD          STA PORTB        ;CLEAR A BIT
00038  C035  8D 57 C0          STA MASK         ;SAVE IN THE MASK
00039  C038  CA           MORE  DEX
00040  C039  D0 EB             BNE LOOP2        ;ALL CHANNELS CKD?
00041  C03B  88                DEY              ;FALL THROUGH YES
00042  C03C  D0 E6             BNE LOOP3        ;COUNTDOWN OVER?
00043  C03E  A9 00             LDA #$00
00044  C040  8D 01 DD          STA PORTB        ;ALL POTR BITS 0
00045  C043  68                PLA              ;RESTOR ALL REGS
00046  C044  A8                TAY
00047  C045  68                PLA
00048  C046  AA                TAX
00049  C047  68                PLA
00050  C048  28                PLP
00051  C049  4C 31 EA          JMP OLDIRQ       ;RESUME IRQ
00052  C04C  C6 FB       WASTE  DEC $FB          ;WASTE 13 CYCLES
00053  C04E  AD 14 C0          LDA ENTER
00054  C051  AD 14 C0          LDA ENTER
00055  C054  4C 38 C0          JMP MORE
00056  C057  00           MASK  .BYTE  0
00057  C058  FE           TABLE .BYTE  $FE,$FD,$FB,$F7
00057  C059  FD
00057  C05A  FB
00057  C05B  F7
00058  C05C  00 00        COUNT .WORD  $0000,$0000
00058  C05E  00 00
00059  C060                     .END
```

that generates four independently controlled pulses on PB0 through PB3 (pins C–F on the USER PORT connector). Connect one servo input directly to each of these outputs and be sure not to forget the +5 volt and the ground leads from each of the servos. This program also runs in the interrupt mode so that its operation is again transparent to BASIC. The four control registers are $0398–$039B (920–923 decimal). The program works in the following manner. It first raises all four outputs and then simply loops for 1.5 milliseconds. As it loops, it checks each control register to see how many iterations were required for each channel. When that number is reached, the output of that channel is lowered to end the pulse. At the end of 1.5 milliseconds, it jumps to the normal interrupt processing routine in the Kernel (the Commodore 64's ROM-based operating system). There is one important compromise in this approach. The 6510 is simply not fast enough to check all four channels very many times before 1.5 milliseconds have elapsed. Thus, 16 is the largest number that you can POKE into a control register. A resolution of only about 15 increments can be achieved with this program, as opposed to the 150 or more increments achieved using the 6526's one-shot timer. For most applications, however, this will probably be a sufficient resolution. Listing 5–5 shows a BASIC program that puts the machine language program into the cassette buffer and resets the interrupt vector.

Stepping Motors

By far the most common computer-controlled mechanical actuator is the stepping motor. These low-cost motors can easily be interfaced to the Commodore 64 and lend themselves to a variety of applications such as robotics, digital plotting, and disk drive technology. The stepping motor, as its name implies, is a special motor that moves in discrete steps under digital computer control. Figure 5–3 shows the construction of a typical stepping motor. Permanent magnets on the moving rotor are surrounded by two stators, A and B. By controlling the direction of current through the windings of the two stators, the rotor can be caused to move one quarter of a pole pitch per polarity change. A two-phase motor with 12 pole pairs per stator coil would thus move 48 steps, or 7.5° per step. When the Commodore 64 controls the polarity changes so that they occur in the proper sequence, the motor can move in either direction one step at a time. Since the Commodore 64 can move the motor by both a predetermined number of steps and at a predetermined frequency, a very precise movement can be accomplished with a stepping motor.

FIGURE 5–3
Cutaway view of a two-phase permanent magnet stepping motor. Courtesy of Airpax Corp.



A Small Step for a Man

Stepping motors come in two varieties—unipolar and bipolar. In the unipolar models, the stator coils are centertapped. The polarity of each coil is switched by applying positive power to the centertap and closing a switch to ground at just one end of the coil. The polarity is reversed by opening that switch and closing the switch on the other end of the coil. A bipolar

motor has no centertap, so a singlepole, doublethrow switch is needed at both ends of each coil to change the polarity of the stators. For simplicity, we chose the unipolar type motor because it requires only half as many components to interface. One such motor that combines both low cost and high performance is the Airpax model K82701-P2, manufactured by North American Philips Controls Corp., Cheshire, CT 06410. (These motors are available for $30 postpaid from The Bit Stop, 5958 South Shenandoah Road, Mobile, AL 36608. Prices subject to change without notice.) This motor requires a +12VDC supply at 400 ma and develops 10.5 oz-inches of torque (see Figure 5-4).

The Hardware

Figure 5-5 shows the interface for the motor. The 4 low-order bits of port B of the USER PORT are connected to the four stator switches. A TTL inverting buffer is used for the connection. This serves two functions. First, it provides additional current drive to the Commodore 64's control lines so that the transistor switches can be fully turned on. Second, since these lines assume a high state when the Commodore 64 is powered up, the inverters cause all of the switches to assume an off condition until the program is loaded and started. This prevents overheating and possible damage to the motor. We chose the common 2N2222 for the stator switches, but any NPN switching transistor with a VCE of 20 volts or greater, an ICE of 600 ma or more, and a beta of 100 or greater will be suitable.

12VDC Power Supply

Since the 400 ma required by the stepping motor is a little more than can be stolen from the Commodore 64's 9VAC supply, we recommend you build a separate supply powered by 110VAC house current. Figure 5-6 shows a simple supply capable of powering several stepping motors.

The Software

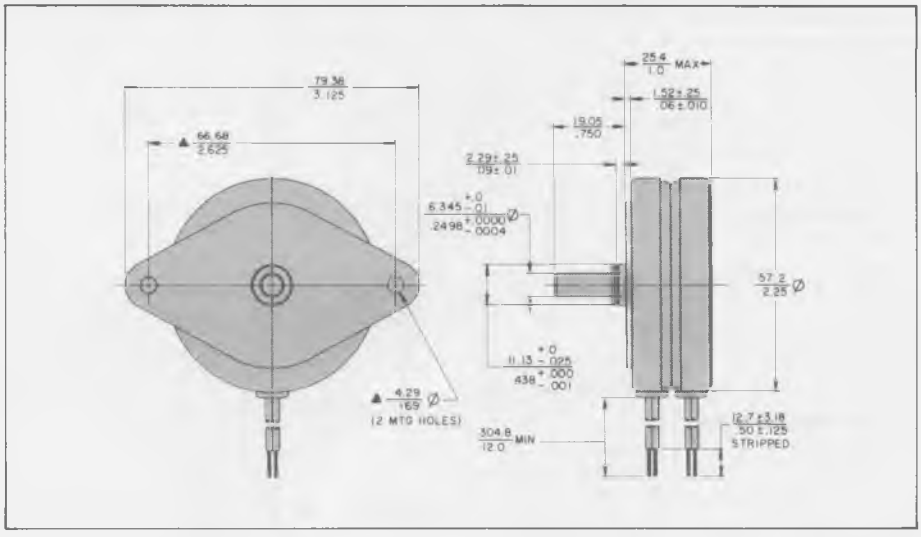
Rotation of the motor is accomplished by turning the four transistors off and on in the proper sequence.

TABLE 5-1: Stepping Motor Switching Sequence

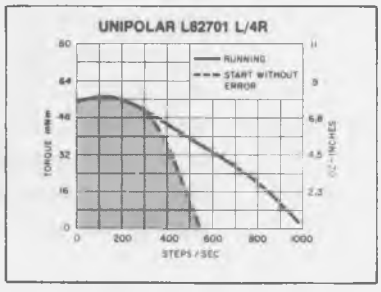
	STEP	*Q1*	*Q2*	*Q3*	*Q4*	
	1	ON	OFF	ON	OFF	
CW	2	ON	OFF	OFF	ON	CCW
Rotation	3	OFF	ON	OFF	ON	Rotation
	4	OFF	ON	ON	OFF	
	5	ON	OFF	ON	OFF	

FIGURE 5–4
Technical data for the Airpax K82701 unipolar stepping motor. Courtesy of Airpax Corp.

DIMENSIONS: MM/INCHES — SYMBOL ▲ ±.27/±.005 UNSPECIFIED ±.78/±.031



SPECIFICATIONS	BIPOLAR K82702		UNIPOLAR K82701		BIPOLAR L82702		UNIPOLAR L82701		BIPOLAR K83702		UNIPOLAR K83701	
SUFFIX DESIGNATION	-P1	-P2	-P1	-P2	-P1	-P2	-P1	-P2	-P1	-P2	-P1	-P2
DC Operating Voltage	5	12	5	12	5	12	5	12	5	12	5	12
Res. per Winding Ω	9.7	61	6.6	36	7.2	41	7.1	40	9.7	61	6.6	36
Ind. per Winding mH	18	110	5	26	15	92	6.4	46	21	96	4.5	24
Holding Torque mNm/oz-in	96/13.7		74/10.5		100/14.2		81/11.5		66/9.4		55/7.8	
Step Angle	7.5°				7.5°				15°			
Step Angle Tolerance	±0.5°				±0.5°				±1°			
Steps per Rev.	48				48				24			
Rotor Moment of Inertia g.m²	3.1 x 10⁻³				3.7 x 10⁻³				3.1 x 10⁻³			
Max. Operating Temp.	100°C											
Ambient Temp. Range												
Operating	—20°C to 70°C											
Storage	—40°C to 85°C											
Insulation Res. @ 500Vdc	100 mΩ											
Bearings	Bronze Sleeve											
Weight	230g/8oz											



LISTING 5-5
BASIC 4 servo

```
0 REM ***DRIVER FOR 4 SERVOS***
1 DATA 8 , 120 , 169 , 20 , 141
5 DATA 20 , 3 , 169 , 192 , 141
10 DATA 21 , 3 , 169 , 255 , 141
15 DATA 3 , 221 , 40 , 88 , 96
20 DATA 8 , 72 , 138 , 72 , 152
25 DATA 72 , 169 , 255 , 141 , 87
30 DATA 192 , 141 , 1 , 221 , 160
35 DATA 16 , 162 , 4 , 152 , 221
40 DATA 91 , 192 , 176 , 32 , 173
45 DATA 87 , 192 , 61 , 87 , 192
50 DATA 141 , 1 , 221 , 141 , 87
55 DATA 192 , 202 , 208 , 235 , 136
60 DATA 208 , 230 , 169 , 0 , 141
65 DATA 1 , 221 , 104 , 168 , 104
70 DATA 170 , 104 , 40 , 76 , 49
75 DATA 234 , 198 , 251 , 173 , 20
80 DATA 192 , 173 , 20 , 192 , 76
85 DATA 56 , 192 , 240 , 254 , 253
90 DATA 251 , 247
99 REM POKE THE PROGRAM INTO MEMORY
100 FOR X= 0 TO 91
110 READ Y
120 POKE 49152+X,Y
130 NEXT X
135 REM START THE PROGRAM
140 SYS 49152
145 REM USER CODE STARTS HERE
150 PRINT "CHANNEL#";
160 INPUT C
170 PRINT "POSITION";
180 INPUT P
190 POKE 49244+C,P
200 GOTO 150
```

Table 5-1 shows that the switching sequence moves in five successive steps. Moving down in the table rotates the shaft clockwise while moving up in the table rotates the shaft counterclockwise. Your program must provide a sequence of bit patterns on the outputs PB0–PB3 to duplicate the pattern in Table 5-1. The sequence of numbers 10, 6, 5, 9, and 10 will generate those bit patterns when successively POKED to address 56577, the address of the interface.

Listing 5-6 is a simple BASIC demonstration program for controlling the stepping motor. Two subroutines, one at 1000 and one at 2000, move the motor in either the clockwise or the counterclockwise direction, respectively. Each time the subroutine is called, the motor is moved one step in the indicated direction. Lines 10 through 30 define the variables while line 40 sets up the 6526 CIA's port B as an output. Lines 50–80 read the command and call the subroutines. Lines 50–80 can be replaced by your custom-written application code. Just remember that the motor moves one step each time one of the subroutines is called.

FIGURE 5–5
Interface for the stepping motor.

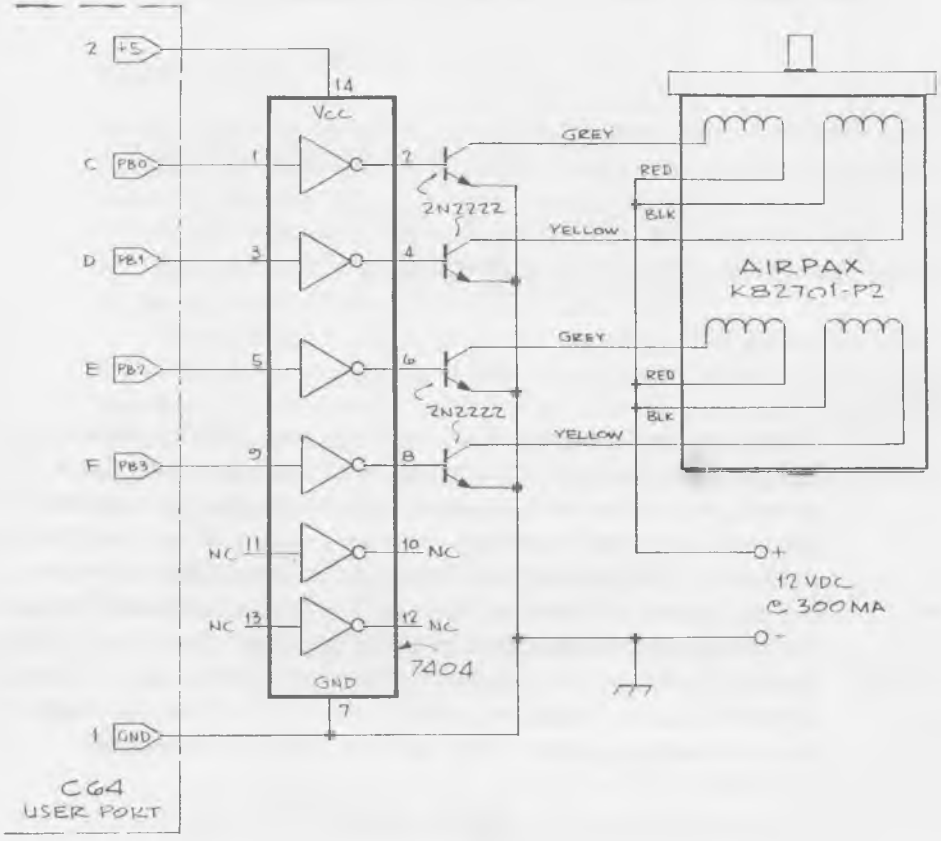
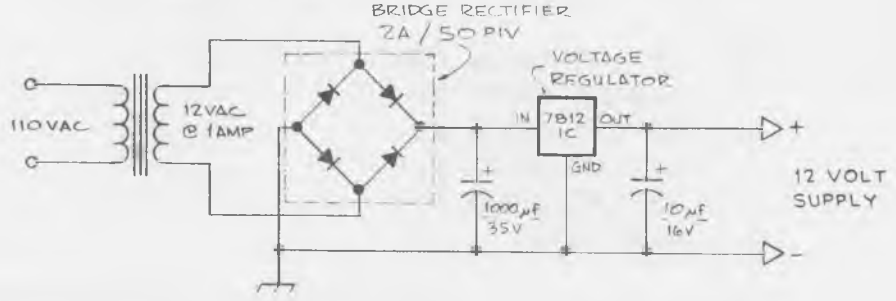


FIGURE 5–6
A power supply to provide the high current required by the stepping motor.



LISTING 5–6
1 stepping

```
10 DATA 10,6,5,9
20 FOR I=1 TO 4:READ Z(I):NEXTI
30 Z1=56577:Z2=1:Z3=5:Z4=0:Z5=4
40 POKE 56579,255
45 PRINT "DIR,STEPS";
50 INPUT A$,X
60 IF A$="R" THEN FORJ=1 TOX:GOSUB 1000:NEXTJ
70 IF A$="L" THEN FORJ=1 TOX:GOSUB 2000:NEXTJ
80 GOTO 50
1000 I=Z2+I:IF I=Z3 THEN I=Z2
1010 POKE Z1,Z(I):RETURN
2000 I=I-Z2:IF I=Z4 THEN I=Z5
2010 POKE Z1,Z(I):RETURN
```

Checkout

When you have constructed the interface, you will first want to test it. Plug the interface into the Commodore 64 user port, apply the +12-volt power, and turn on the Commodore 64. Verify that the Commodore 64 has come up properly and that you have a cursor. If not, turn the Commodore 64 off immediately and locate the problem. (Hint: Shorts between +5 and ground will blow the 3-ampere fuse located inside the Commodore 64.) Assuming that you have a properly operating system, verify that you have +12 volts on the collectors of Q_1-Q_4. Failure to see +12 volts on any of the collectors indicates a faulty transistor or failure to attach a lead from the stepping motor. Now type the following command:

```
POKE 37138, 255
```

All of the collectors should experience a transition to less than 2 volts. (Caution: As soon as you verify this test, reset the Commodore 64 by cycling the power switch to avoid overheating the stepping motor. Failure of any transistor's collector to assume a low voltage indicates either a bad transistor or miswiring of that transistor's control circuit.) Once you have verified the above tests, enter the following program and RUN it.

```
10 POKE 37138, 255
20 POKE 56577, 10
30 POKE 56577, 6
40 POKE 56577, 5
50 POKE 37236, 9
60 GOTO 20
```

Your motor should be rotating at about 100 rpm. Failure of the motor to rotate at this stage is probably related to a mix-up in the leads from

the motor. Be sure that the gray and yellow leads from stator A (the coil farthest from the shaft) are attached to Q1 and Q2, respectively.

Multiple Motors

An additional motor can easily be added to the interface by simply adding a second 7404 buffer and four more transistors. Connect the second motor in exactly the same way as the first motor, except that pins 1, 3, 13, and 11 from the 7404 should connect to PB4, PB5, PB6, and PB7, respectively. Thus, one motor is driven by the low-order 4 bits and the other motor by the high-order 4 bits of the 6526 CIA's port B.

The software required to operate the two motors appears in Listing 5-7. Four subroutines are provided—two for each motor—so that either direction can be achieved. A GOSUB to 1010 will step motor 1 clockwise, while a GOSUB to 2010 will step motor 1 counterclockwise. Similarly, a GOSUB to 3010 will step motor 2 clockwise, and a GOSUB to 4010 will step motor 2 counterclockwise.

LISTING 5-7
2 stepping

```
10 REM DRIVER FOR TWO STEPPER MOTORS
20 DATA 10,6,5,9
30 FOR I=1 TO 4:READ Z(I):NEXTI
40 Z1=56577:Z2=1:Z3=5:Z4=0:Z5=4
50 POKE 56579,255
60 I=1:J=1 : REM SET UP COUNTERS
70 REM*******START YOUR CODE HERE*******
80 REM
998 REM*********SUBROUTINES*************
999 REM
1000 REM MOTOR 1 CW
1010 I=I+Z2:IF I=Z3 THEN I=Z2
1020 POKE Z1,Z(I)+16*Z(J):RETURN
1999 REM
2000 REM MOTOR 1 CCW
2010 I=I-Z2:IF I=Z4 THEN I=Z5
2020 POKE Z1,Z(I)+16*Z(J):RETURN
2999 REM
3000 REM MOTOR 2 CW
3010 J=J+Z2:IF J=Z3 THEN J=Z2
3020 POKE Z1,Z(I)+16*Z(J):RETURN
3999 REM
4000 REM MOTOR 2 CCW
4010 J=J-Z2:IF J=Z4 THEN J=Z5
4020 POKE Z1,Z(I)+16*Z(J):RETURN
```

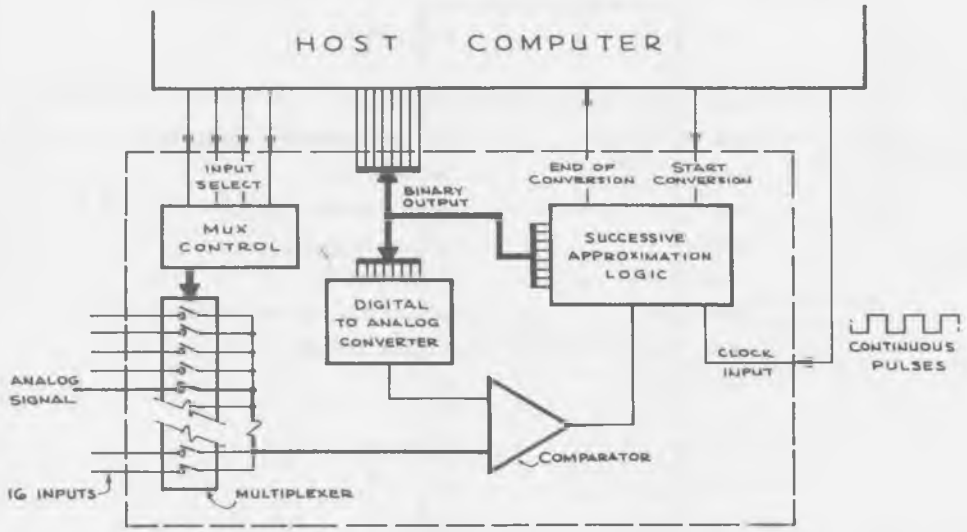
6

ANALOG-TO-DIGITAL CONVERSION

As its name implies, the analog-to-digital converter (ADC) generates a multiple-bit binary signal that is proportional to the voltage it is monitoring. The heart of the ADC is an internal digital-to-analog converter (DAC) combined with a computing circuit called the successive approximation logic. When the ADC is started, the successive approximation logic sends a bit pattern to the DAC. The voltage generated by the DAC is then compared to the input voltage. The result of this comparison is then sent to the successive approximation logic so that a new bit pattern can be generated that will bring the DAC's voltage closer to the input. When the DAC's voltage is as close as it can be to the input, the successive approximation logic sends an end-of-conversion signal to the computer. The computer responds by reading the bit pattern on the DAC through a parallel port.

A valuable addition to an ADC is a multiple-channel multiplexer, or MUX. The MUX selects one of many input signals to be converted. As shown in Figure 6–1, it can be thought of as a multiple pole switch that can be set by the computer. Because the computer can sample a voltage very quickly through the ADC, the addition of a MUX will allow the computer to rotate through the sampling of several analog signals and give the appearance of simultaneously measuring all the signals.

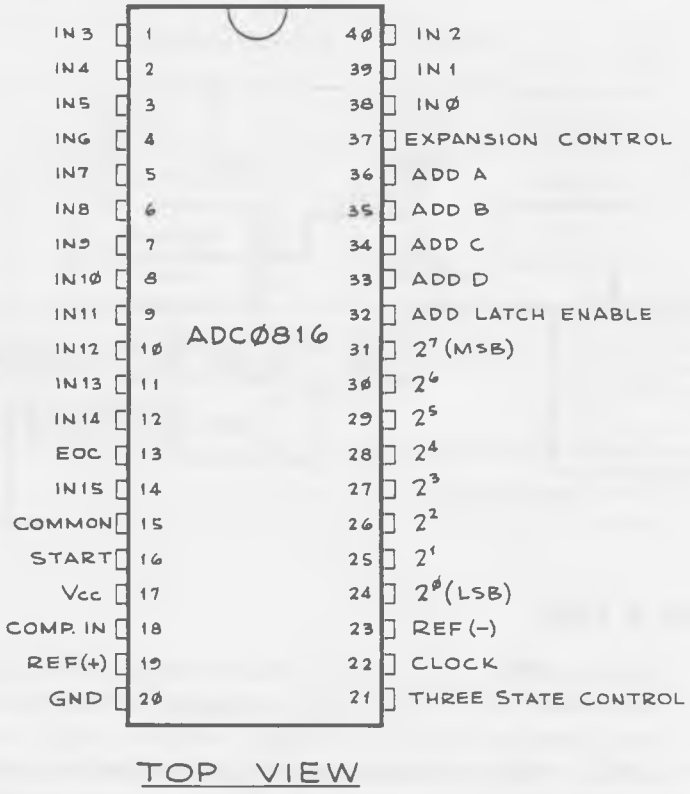
FIGURE 6–1
A block diagram of the ADC0816 analog-to-digital converter.



The ADC0816 Chip

Until recently, ADCs were expensive and out of the reach of most experimenters. Furthermore, they were quite complex and difficult to interface. These problems have happily been overcome, however, with the advent of National Semiconductor's ADC0816 chip (see Figure 6–2), which puts an entire analog-to-digital converter on a single chip. Three-state outputs and a 16-channel input multiplexer have also been thrown into this chip just for good measure. At the time of this writing, these chips are available for about $20 each. Although many other analog-to-digital converters are on the market, we feel that the ADC0816 is by far the most cost-effective of them all. The ADC0816 is an 8-bit converter; that is, it will discriminate to one part in 2^8 (256). This precision is better than one-half-percent accuracy (which is usually good enough for most applications). It also conveniently mates with the 8-bit word structure of the 6510 data bus. The ADC0816 will complete a conversion in under 100 microseconds. Figure 6–3 shows the interface for the ADC 0816 to the Commodore 64. Note that the address latch enable (ALE) and the START pins are connected together. When the address latch enable goes from a low to a high, the binary code for the channel to be sampled is loaded from the four address lines (A-D). Notice that these address lines are connected to bits 0–3 of port B. When START (start the conversion process) experiences a high-to-low transition, the analog-to-digital conversion sequence is started. When the conversion is finished, the digital value must

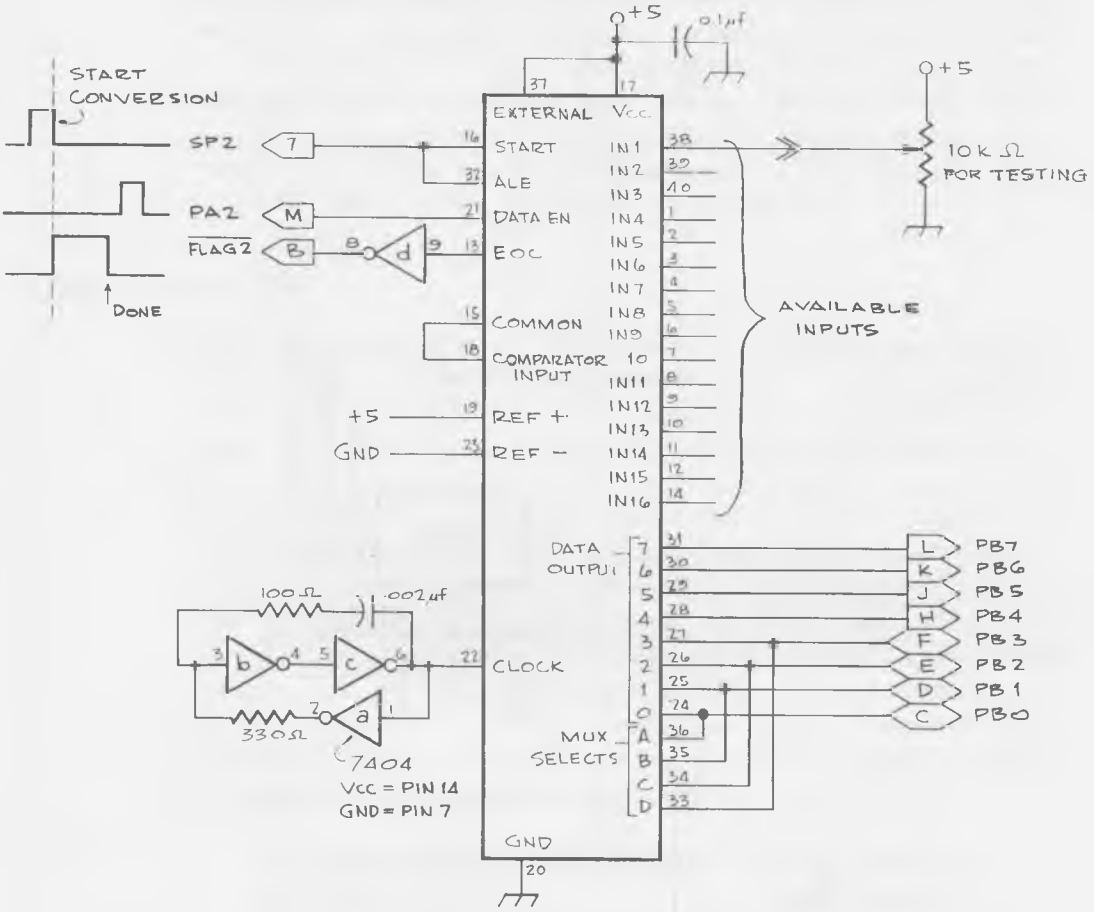
FIGURE 6–2
Pinouts for the ADC0816



be read from the eight data output lines. The data output lines are only active when the data enable pin is high.

We can use the output of the shift register, SP2, to generate a start pulse. When timer A is running, POKEing a 128 to the serial port will cause SP2 to go high for one clock cycle and then low for seven more cycles and stay low until the next POKE to the serial port register. To read the result, you must first change port B to inputs by POKEing a 0 to the data direction register. Next, PA2 must be raised high to enable the data output lines. A PEEK to port B will retrieve the digitized value. It is important that you don't try to read the ADC until the conversion is completed. An end-of-conversion signal (EOC) is available from the ADC0816 but is not at all needed with BASIC for the simple reason that the Commodore 64 takes much longer than the 100 microseconds conversion time to execute the POKEs and PEEK for accessing the ADC. Thus, using BASIC, it would be impossible to try to read the ADC before the conversion has been completed. If, however, you plan to program the ADC in machine language, then it will be necessary to use the EOC signal con-

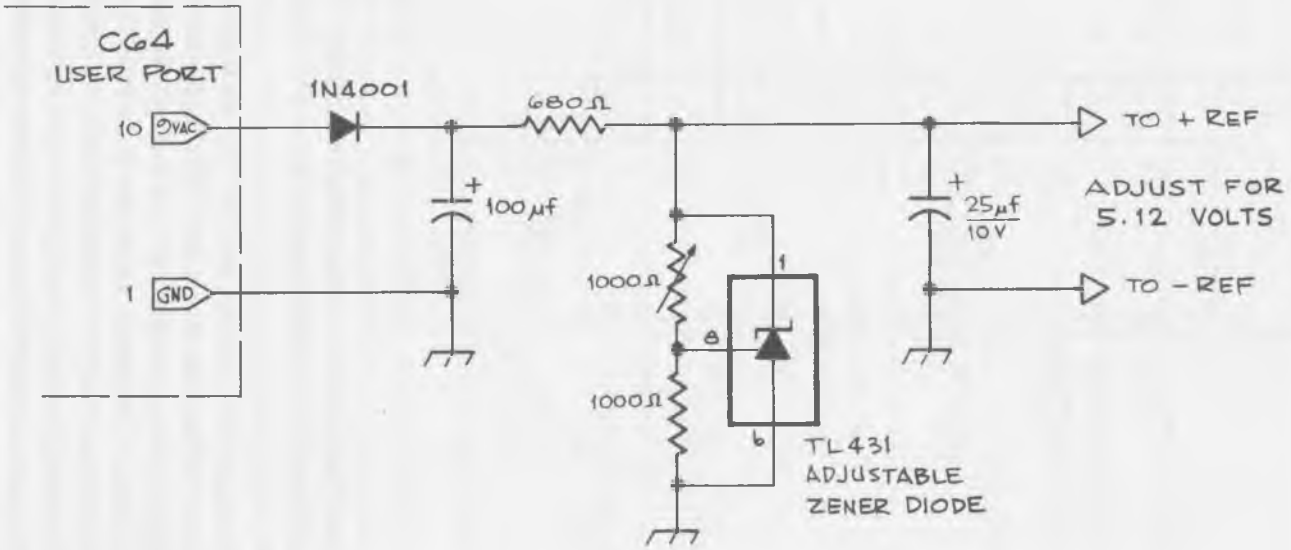
FIGURE 6–3
Interconnections between the ADC 0816 and the Commodore 64.



nected to FLAG2 and have the machine language program check its status (bit 4 of the CIA's interrupt-control register) before reading the ADC's output.

In Figure 6–3, we have tied REF(+) and REF(–) to Vcc and ground, respectively. The precision of the analog-to-digital converter will depend on the precision and stability of these voltages in your system. For example, if there is 60 Hertz ripple on your board's +5 volts, then the reproducibility of the ADC will be affected, because REF(+) and REF(–) are the internal reference voltages for the converter's DAC. If you find that you cannot obtain the degree of accuracy you desire in your system, you should then consider adding a precison voltage source like the one shown in Figure 6–4. Note that the reference supply is set for 5.12 volts rather than an even 5 volts. This setting has the simple advantage that 256,

FIGURE 6–4
A precision reference source for the ADC.



the maximum capacity of the 8-bit word, is an even multiple of 5.12. Thus each bit of the converted value will equal 5.12/256, or 0.02 volts. If you use the 5-volt supply as the reference, the calibration factor will be 5.00/256 or .01953, and that is slightly less convenient. We have found that the regulated 5 volts on the USER PORT are quite suitable for all but the most demanding applications. Finally, the ADC0816 needs a .5-mHz clock. This clock is generated with an oscillator constructed from three inverters of a 7404, two resistors, and a .002-mfd capacitor.

Check-out of the ADC

Make the connections as shown in Figure 6–3. Doublecheck the connections and only install the 7404 into its socket. Be sure all power and ground pins to the ICs are properly connected. Again, the reader should be warned that errors in wiring could damage your computer. Plug the ADC interface card into the computer. Turn on the monitor and turn on the Commodore 64. Check to see that the normal sign-on appears on the monitor. If it does not, turn off the power and locate the wiring error before proceeding. Now test for pulses from the clock oscillator on pin 6 of the 7404 with your logic probe. If they are present, turn off the power and put the ADC0816 in its socket. You will need a test voltage that can be varied between 0 and +5 volts. The wiper of a 10K pot between system power and ground does nicely. Put the voltage source on pin 38 (input 0) of the ADC0816. Turn on the computer and again verify that the sign-on is present. Enter the following program and start it.

```
10 POKE 56580,4 : POKE 56581,0     set up Timer A)
20 POKE 56590,69                   (start Timer A)
30 POKE 56579,15 : POKE 56577,0    (put a zero on port B)
40 POKE 56588,128                  (start pulse on SP2)
50 POKE 56579,0                    (port B as inputs)
60 POKE 56576, 151                 (raise PA2)
70 PRINT PEEK (56577)              (read value)
80 POKE 56576,147 : GOTO 30        (lower PA2 and loop)
```

Numbers should be scrolling up the screen. As you change the input voltage, the numbers should change in proportion to the input voltage. If no response is obtained, you should proceed as follows:

1. Is there a start pulse? Check pin 16 on the ADC with the logic probe while the above program is running. The line should be low with very short positive pulses. The pulses should stop when the program

is stopped. If not, check the wiring between pins 16 and 32 of ADC and PA2 or check program lines 20 and 30.

2. Is there a tristate strobe? Check pin 21 on the ADC with the logic probe as the program runs. The line should have short pulses like the start pin. If not, check wiring between PA3 and ADC pin 21, program line 30.
3. Is the multiplexer working? Check pin 38 on the ADC with a voltmeter and see if it varies as the test voltage is varied. After this condition is verified, place the voltmeter on pin 15, the output of the multiplexer. This voltage should be the same as the input voltage on pin 38. If not, check the address lines, pins 33 to 36, to see if they are wired correctly to PB3–PB0. Address is output in program line 20.
4. Is the converter working? Check for clock pulses on pin 22 of the ADC with the logic probe. If these are present, start the preceding program and check for an EOC pulse on pin 13. Lack of a pulse here indicates that the converter is not working. If all these tests are met but no EOC pulse is present, then your ADC0816 chip may be defective.

Programming the ADC

The ADC0816 is very easy to program in BASIC and will require machine language programming in only the most critical circumstances. The user port VIA must be initialized for bits 2 and 3 of port A as outputs both set low. The converter is started by making PA2 high, then low, with bits 0 through 3 of port B outputting 0 to 15, selecting a channel. Listing 6–1 illustrates how to initialize port A and select a channel.

This example of selecting an analog channel is straightforward. The access sequence starting in line 30 starts by setting bits 0–3 port B as out-

LISTING 6–1
ADC driver

```
10 REM ***ANALOG TO DIGITAL DRIVER***
20 REM     SET UP CIA CHIP
30 POKE 56580,4    :REM TIMER LOW BYTE
40 POKE 56581,0    :REM TIMER HI BYTE
50 POKE 56590,69   :REM START TIMERA
60 POKE 56588,128  :REM SET BP LOW
70 POKE 56576,147  :REM SET PA2 LOW
75 REM ****START YOUR CODE HERE****
80 PRINT "CHANNEL#";:INPUT CH:GOSUB 1000
90 PRINT X:GOTO 80
1000 REM ***ADC SUBROUTINE*************
1010 REM * CH= CHANNEL# (0-15)        *
1020 REM * X=DIGITIZED #              *
1030 REM ***************************
1040 POKE 56579,15:POKE 56577,CH:POKE 56588,128
1050 POKE 56588,0:POKE 56576,151:X=PEEK(56577):POKE 56576,147:RETURN
```

puts. The second POKE sends the channel number to the ADC, and the third POKE forces the ALE line high while loading the MUX address, then low to start conversion. The conversion takes less than 100 microseconds. There is no need to worry about trying to read the data before it is ready because BASIC is not fast enough to do so.

Many analog-to-digital conversion applications require sampling at present-time intervals. The internal real-time clock in the Commodore 64 can be used to accomplish this timing. As an example of this technique, a simple acquisition program is included. The code in Listing 6–2 causes the ADC to take 100 samples at a rate of two per second.

The subroutine starting at 100 does the complete channel select and reads the analog results. Keeping this subroutine near the top of the program helps with execution speed. The real-time clock, TI, is sampled in line 200, and line 220 delays until ½ second has elapsed (30 Jiffies). Variations of this code can allow multichannel sampling and varied timing. This program will run at better than 20 samples per second. Keep in mind, however, that the program overhead can become an upper limit to the number of samples you can make each second. Replacing all constants in lines 100, and 110 with predefined variables will further speed up execution time. If faster data rates are required, we recommend a machine language program.

Analog Input Conditioning

The ADC0816 has a high-impedance input and accepts a signal between 0 and +5 volts. If your signal falls in that range, then nothing further will be needed. More often than not, however, the signal you wish to

LISTING 6–2
ADC 100 points at 2/sec

```
5 REM COLLECT 100 POINTS AT 2 PER SECOND
10 DIM S(100)
20 REM     SET UP CIA CHIP
30 POKE 56580,4    :REM TIMER LOW BYTE
40 POKE 56581,0    :REM TIMER HI BYTE
50 POKE 56590,69   :REM START TIMERA
60 POKE 56588,128 :REM SET SP LOW
70 POKE 56576,147 :REM SET PA2 LOW
80 GOTO 200 :REM JUMP TO MAIN PROGRAM
100 POKE 56579,15:POKE 56577,CH:POKE 56588,128
110 POKE 56588,0:POKE 56576,151:X=PEEK(56577):POKE 56576,147:RETURN
199 REM MAIN PROG STARTS HERE
200 CH=0:T=TI
210 FOR J=1TO100   :REM 100 POINTS
215 T=T+30         :REM DELAY 30 JIFFIES
220 IF TI-T<1  THEN 220
230 GOSUB 100
240 S(J)=X
250 NEXT J
260 PRINT "ARRAY IS COMPLETE"
```

measure will fall outside that range. For this reason, some input conditioning will probably be required. If the signal is greater than +5 volts in amplitude and does not swing negative, then a simple resistive attenuator, as shown in Figure 6–5, will do. The design rules are: Select a desired input impedance, solve for R2 with equation 1, and then solve for R1 with equation 2.

If your signal swings negative or is considerably less than 5 volts, then an op-amp circuit will be needed, as shown in Figure 6–6. This circuit provides an offset shift to accommodate negative as well as positive signals and scales the signal so that the full input range produces a swing between 0 and 5 volts from the ADC0816. The pot is an offset adjustment; it should be adjusted so that 0 volts (input grounded) yields a converted value of 128, half-scale. Note that the op-amp inverts the signal as well as scaling it so that negative voltages of the input will yield converted values between 128 and 255. Positive voltages will yield values between 0 and 128. The polarity can be restored in software along with the scale factor. For example, suppose that a reference of 5.00 volts is used of the ADC, and RIN is 40K. By equation 1 in Figure 6–6, this value will accommodate an input signal between +1 and –1 volts. The following changes to the program in Listing 6–2 cause it to sample a voltage on channel 1 and set X equal to the same value in volts: 200 S(J) = (128–X)*(1/128)

FIGURE 6–5
Input attenuator for the ADC. Solve for R_1 and R_2 in the circuit by providing the desired input impedance, Z (be realistic), and the maximum signal voltage, Ein (max).

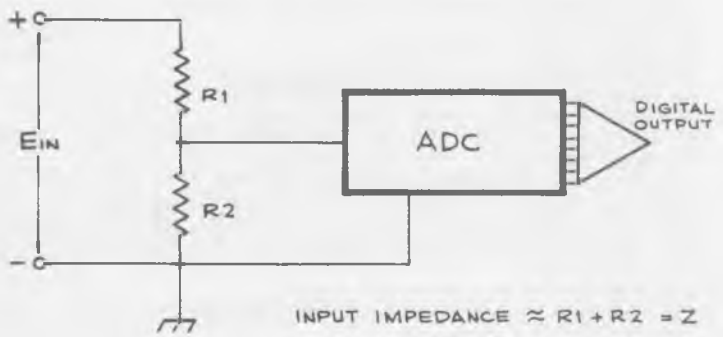
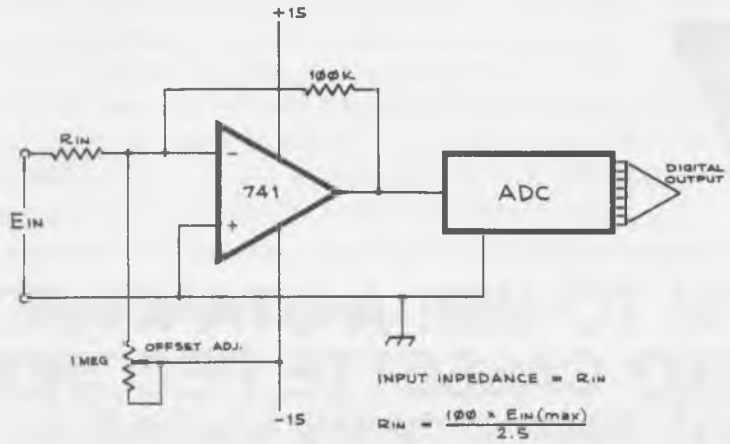


FIGURE 6–6
Op-amp signal conditioner. Use this circuit when Ein (max) is either less than 5 volts or when the signal is bipolar.



PARTS LIST

1 Vector 4.5 x 5.5 circuit card #3662-5 or equivalent
1 ADC0816 National Semiconductor, etc.
1 7404 hex inverter
1 40-pin wire-wrap socket
1 14-pin wire-wrap socket
1 100-ohm ¼W resistor
1 330-ohm ¼W resistor
1 .002-mfd ceramic capacitors
3 .1-mfd capacitors

Precision Reference

1 TL431 adjustable zener diode
1 8-pin wire-wrap socket
1 1N4001 rectifier diode
1 680-ohm ¼W resistor
1 1000-ohm ¼W resistor
1 1000-ohm trimpot
1 100-mfd 20V electrolytic capacitor
1 220-mfd 10V electrolytic capacitor

7

HOW TO USE A STANDARD AUDIO CASSETTE RECORDER WITH THE COMMODORE 64

This chapter describes how an ordinary portable audio-cassette tape recorder can easily be connected to the Commodore 64 and used to take the place of Commodore's own data cassette recorder. No changes are required to either your audio recorder or your Commodore 64. With the circuit described, a minimum of electronic parts will yield a simple-to-operate and reliable means for saving and loading programs and data. Furthermore, your new interface will allow complete compatibility with commercially available Commodore 64 tapes.

The Commodore cassette recorder is basically a converted audio recorder with modifications to make it usable only as a Commodore data device. A look inside reveals no speaker and different wiring for the control buttons. Also, Commodore's unit gets its power from the host computer. As it turns out, information saved on tape by a Commodore recorder is in fact stored as normal audio information.

By adding the simple electrical circuits given here, you can make your Commodore 64 think it is connected to a Commodore digital cassette recorder rather than an inexpensive standard audio recorder.

What Kind of Recorder?

For this project you will need a standard monophonic (single channel) portable audio-cassette recorder (we used the Radio Shack CTR 56 for testing

this project) with microphones and/or auxiliary input, earphone output, and a remote input for controlling the motor. The remote input jack is usually next to the microphone input and is smaller. Another feature that is desirable but not essential is a tape counter. This counter is handy for locating a particular record when multiple records are stored on one tape.

There is a slim possibility that not every recorder will work directly as described here. Since it was impossible to test all types of recorders on our circuit the reader should be aware of this possibility. There are some problems that we have anticipated, and these are discussed in the closing paragraphs for those in need of troubleshooting.

Storing Information on Magnetic Tape as Sound

Audio recorders are used in the microcomputer world primarily because they are a relatively inexpensive means of mass storage. A good information recording technique such as that found in the Commodore 64 allows you to use even cheap, low-grade tapes reliably. Audio recording is fairly slow compared to a floppy disk; nonetheless, the use of tape recording for computers will be around for a while to come.

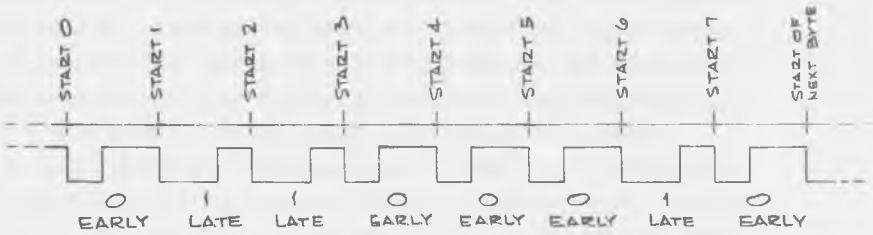
Before we dive deep into construction, let's first investigate the basics of how information can be put on tape as audio information and later be retrieved for reuse. The fine details of how the Commodore 64 saves information on the magnetic tape are beyond the scope of this book. However, a basic description of this process will help you understand the interface you will be building.

The unit of information saved is the bit. A record on tape is actually a long chain of bits sequentially making up the bytes of the original RAM memory block. Identifying the memory blocks for saving and loading is automatically taken care of for you by programs built into your Commodore 64 computer. The owner's manual that came with your Commodore 64 explains about SAVE, LOAD, OPEN, CLOSE, and files in general. Later on, we will discuss operation of the interface with these commands.

A byte containing the value 01100010 ($62) encoded as bits for recording is illustrated in Figure 7–1. Each bit going to the tape causes the output line of the recorder to first go to a low state (0 volts) to start the cycle, delay for a moment, then return to the high state (+5 volts) to finish the cycle. If the bit is 0, the low to high transition occurs early in the cycle. The transition occurs late in the cycle for a 1. The timing is, of course, computer controlled and very accurate.

You must be asking by now, "Just what does all this bit shifting have to do with audio?" These pulse trains of information result in frequencies within the audio range. By properly connecting your computer to your cassette recorder, the recorder will obligingly record the pulses as sound.

FIGURE 7-1
Serialized byte for the C64's recorder.



However, the peculiar noises on the tape will be music only to a computer's ear.

Schmitt Triggers

The computer outputs the signals as square wave pulses, but when played back, those nice fast transitions are rounded by the frequency response of the recorder. Examples of these wave shapes are given in Figure 7-2, A and B. The playback signals represent the original computer output, except for the rounded edges. This presents a problem that must first be corrected before the signal can be reloaded into the Commodore 64. A simple solution to this dilemma is the Schmitt Trigger (ST). This device will convert a rounded signal to square pulses, as shown in Figure 7-2, C. The ST has a fast switching characteristic so that its output is only in the high or low state. Figure 7-3 (lower panel) gives a simple circuit using one Schmitt Trigger gate of CMOS 4093 quad NAND gate to square the playback signal from the recorder.

Since the 4093 is an inverter, the recorded data should also be inverted in order to preserve the polarity of the information we started with. Since there are four separate gates in the 4093, we can use one gate, again wired as an inverter, in the computer-to-recorder connection to invert the output. In this case, the Schmitt Trigger characteristic is not required;

FIGURE 7-2
Reconstruction of the square waves from the recorder's playback signal.

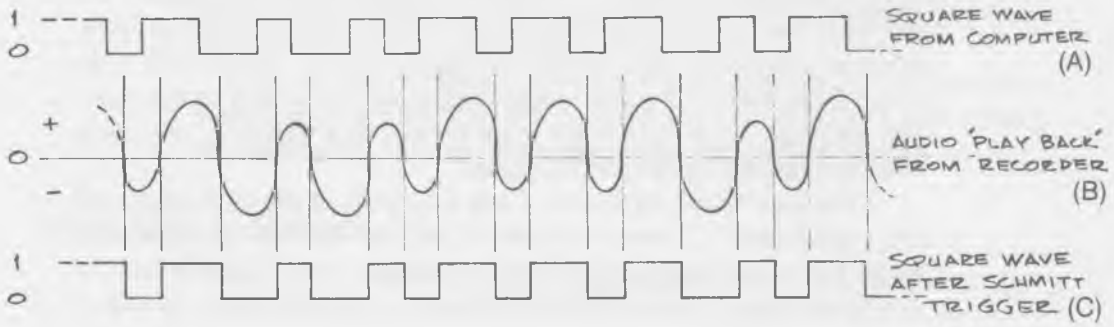
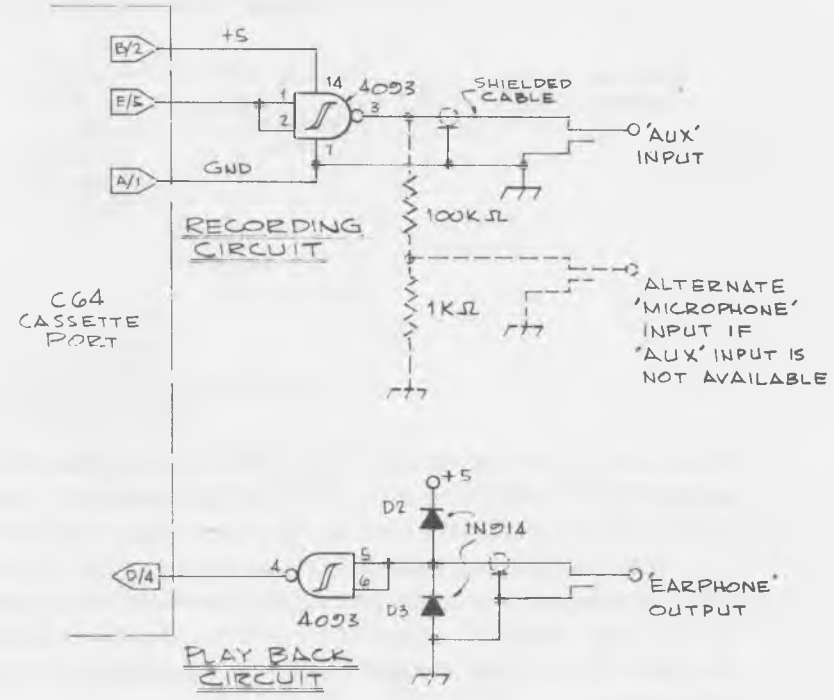


FIGURE 7–3
The interface between the audio recorder and the C-64.



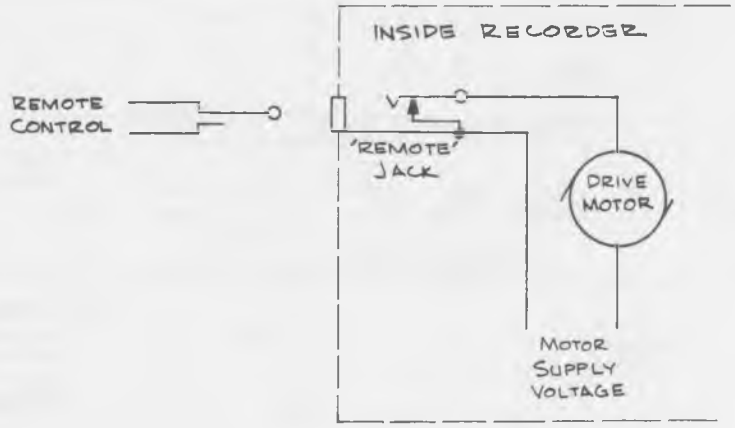
however, this characteristic does not adversely affect the desired action and allows us to use only one IC.

Finally, the diodes in the playback circuit are needed to limit the input-signal voltage level from the recorder during playback. This is necessary because the recorder could output a voltage to the 4093 high enough to damage the chip if the volume control on the recorder is set high. With the diodes in our circuit as shown, any signal voltage greater than 5 volts peak-to-peak will be clipped to 5 volts, and safe operation is ensured.

Motor Control

We do not recommend that you try to power your recorder with power from the Commodore 64. Use the 115VAC adaptor or batteries. You will, however, have to control the motor through the remote input of the recorder. Shorting the contacts on a plug inserted into the remote jack will allow the recorder to play. Breaking the connection between the contacts will stop the recorder motor. Figure 7–4 shows the typical remote circuit inside a recorder. By making or breaking the remote leads, the Commodore

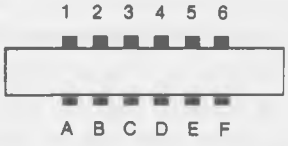
FIGURE 7–4
Remote control circuit in the recorder.



64 can start and stop the recorder motor. This is accomplished with a 5-volt, normally open, single-pole relay from the Commodore 64 cassette port, pin C/3 (see Figure 7–5 for pinouts). The relay circuit is shown in Figure 7–6. Note the spike-suppression diode across the coil of the relay. This diode is required to prevent the generation of unwanted voltage spikes when the coil is deenergized. If this diode is left out, the spikes created can be recognized as sporadic computer data and can cause your computer to malfunction.

Part of the motor control circuit is a switch circuit to tell the Commodore 64 when you have the tape recorder ready to SAVE or LOAD. The Commodore recorder PLAY buton is connected to pins F or 6 to tell the Commodore 64 when the PLAY button is pressed. Our motor control circuit must also provide this feedback signal to keep the Commodore 64 happy. The input on pin F/6 has to be brought logically low before the Commodore 64 will start the motor. The simple circuit in Figure 7–7 illustrates how a toggle switch can be used to generate this signal. Remember, this switch does not directly control the motor but tells the Commodore 64 when the tape recorder is ready. The Commodore 64 will take care

FIGURE 7–5
Pinouts for the cassette port. Courtesy of Commodore Business Machines.



PIN #	TYPE
A-1	GND
B-2	+5V
C-3	CASSETTE MOTOR
D-4	CASSETTE READ
E-5	CASSETTE WRITE
F-6	CASSETTE SWITCH

FIGURE 7–6
Motor control for the recorder interface. The relay is available from Radio Shack (part number 275–243).

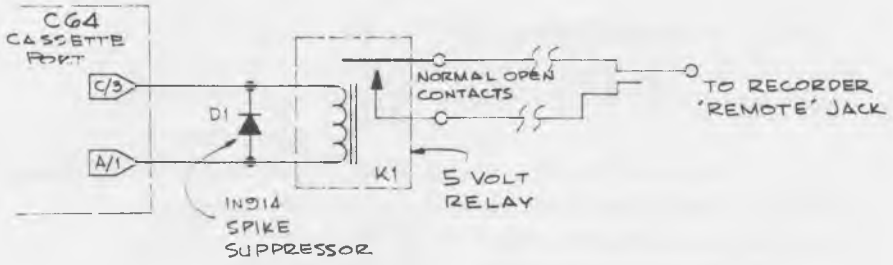
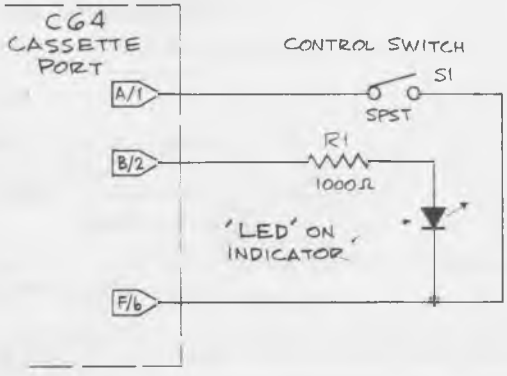


FIGURE 7–7
Ready control circuit for the recorder interface.



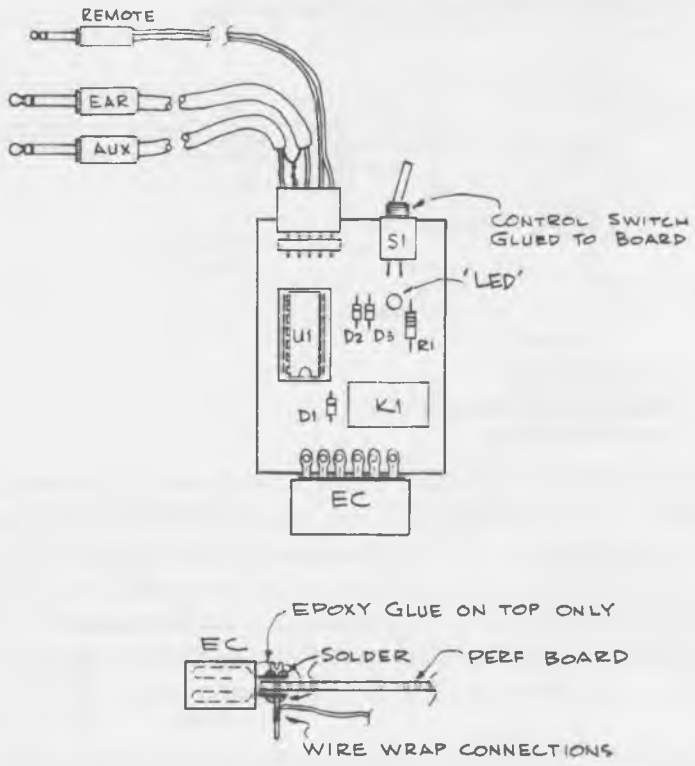
of starting and stopping the motor, but only so long as the Commodore 64 thinks the PLAY button is down. Thus, it is up to you to throw the switch whenever the PLAY button is depressed. The LED tells you when F/6 is low and is handy for seeing at a glance the status of the control switch. This addition is not necessary but recommended.

Construction

To begin constructing your cassette interface, you will want to choose a suitable board for mounting the parts. An example layout is given in Figure 7–8. This layout is for a 1½-inch by 2-inch piece of phenolic experimenter's board with predrilled holes at 1/10-inch centers. The size given in our layout is ample for the few parts needed.

We mounted a 6/12-pin edge connector across one of the 1½-inch ends of the board. Since the upper and lower pins of the Commodore 64 cassette port are the same, they may be connected together. An easy way

FIGURE 7–8
Mechanical details of the recorder interface.



to connect the female edge connector to the perf board is detailed in Figure 7–8. Sandwich the perf board between the pins of the edge connector and push a wire-wrap terminal post through the holes in both the edge connector solder tails and the perf board. Then solder the wire-wrap post to each solder tail on both sides of the perf board. Do this for all six connectors. If the edge connector is loose, put some fast-setting epoxy glue across the top legs, but leave the underside clean for connecting the circuit.

We used 5-pin, right-angle, single-row male and female header connectors for attaching the three cables from the interface board to the recorder. These could be soldered directly to wire-wrap pins on the board, however. The input and output cables should be made of small-diameter, audio-grade shielded cable with the miniature phone plugs soldered on.

The cable for the remote control can be shielded but does not have to be. The shield leads of the input and output cables can be tied together at the interface end and connected to a ground pin of the header. We don't recommend that either side of the motor control circuit be tied to ground. Manufacturers differ as to what side of the recorder's power is switched

by the motor control jack, and the possibility of a short exists if you try to ground one of these wires.

PARTS LIST

1 1½″×2″ phenolic project board predrilled @ .1″×.1″ grid (Cut from Radio Shack 4½″×5-5/8″ Cat. No. 276-1392 or equal)
1 SPDT DIP relay, 5VDC coil (Radio Shack 275-243 or equal)
1 4093 dual-input quad NAND gate
1 SPST toggle switch (Radio Shack 275-624 or equal)
1 5-pin, right-angle, single-row male and female header connector (Cut from a 36-pin AP Products Incorporated connector Digi-Key parts nos. 929835-02 male & 929974 female or equal)
1 6/12 edge connector with a .156″ pin spacing
2 Mini phono jacks (AUX & EAR plug-ins)
1 Micro phono jack (REMOTE plug-in)
1 1000-ohm ¼W resistor
1 LED 15-ma (low power type)

Miscellaneous

4 ft. small-diameter shielded cable
2 ft. #20 twin lead

Alternate

To replace the toggle switch with the RS flip-flop, omit the toggle switch and substitute the following:

2 SPST subminiature momentary-contact switch (Radio Shack 275-1571)
2 2.2K resistors

Testing and Operation

Doublecheck your wiring. An electrical circuit is like a computer program in that a circuit will only work properly if it is constructed exactly as it should be. With the Commodore 64 turned off, connect the interface to both the computer and the recorder. Set the recorder volume control to ¾ full and power up the Commodore 64. If the normal power-up message does not appear on the screen, turn off the power and check your wiring again, especially the +5 volt and ground connections. Shorts in the +5 volt wiring may blow the fuse inside the Commodore 64 or power-pack.

If all systems are go, turn the toggle switch off and on. The LED indicator should go off and on, and you should also be able to hear the relay click open and close. If things check out so far, you are ready to try recording.

Type in, but do not RUN, the following short program:

```
10 DATA 84,69,83,84,0,79,75
20 ? : ?
30 FOR A=1 TO 7
40 READ B
50 ?CHR$(B);
60 NEXT
```

Put a tape in the recorder and flip the toggle switch to the off position. Set your recorder to RECORD. The motor should not turn on. Now turn on the toggle switch and the recorder should start. Let it run for about 15 seconds to make a leader before recording, then turn off the toggle switch to stop the motor. You will always want to start a new tape in this manner to move past the blank portion at the beginning of the tape.

Type in SAVE "TEST" and hit RETURN. The computer should respond with PRESS RECORD AND PLAY, but your recorder should already be this way. Respond to the computer's prompt by turning on the toggle switch. The computer will come back with SAVING TEST. The recorder will spin for a few seconds and stop automatically; at that point the Commodore 64 will output READY.

Now let's see if it worked. Press STOP and then REWIND on the recorder. The motor will not start until you turn the switch off, then back on. When the tape is rewound, turn the switch off and press STOP on the recorder. Type in NEW to erase the program from memory and then press both the SHIFT and RUN/STOP keys on the Commodore 64 at the same time. This is the auto-load/run request and the Commodore 64 will command PRESS PLAY. Press PLAY on the recorder and turn on the interface switch. The recorder motor will start, run for a moment, and then stop automatically. If everything went right, the program that was loaded will tell you so.

Troubleshooting

If nothing loaded, check the wiring in the 4093 read and write circuits. A simple trick to see if data is going to the tape is to disconnect the cables to the recorder and listen to the tape after recording as described above. If information was recorded, you will hear a high-pitched tone as the data is played back.

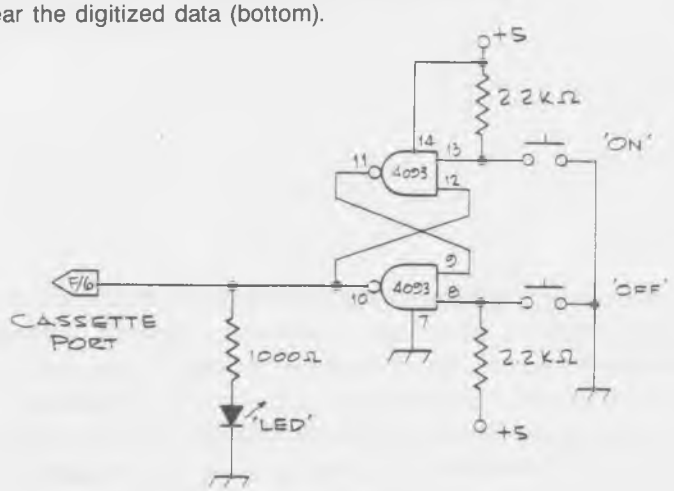
You may have to experiment with the volume-control setting to get reliable loading. If you have access to a logic probe or if you built the one illustrated in Chapter 3, you can thoroughly test your interface. While saving a program, watch for pulses on pin E/5, then at the output of the

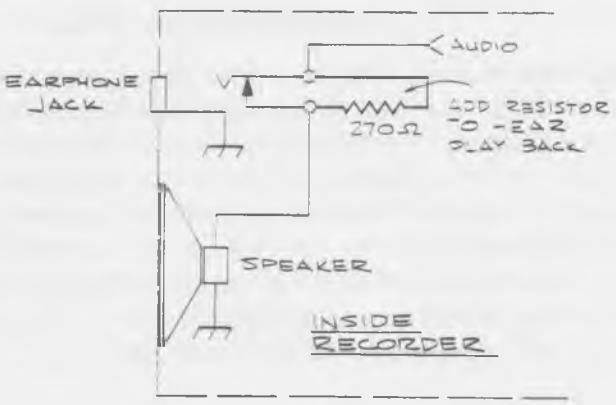
inverter. This will tell you if the signal is going to the recorder. A similar test can be performed in the playback circuit to see if information is coming to the computer from the recorder.

Another potential problem area is the connections on the plugs on the cables to the recorder. Make sure there aren't any shorts and be certain of polarity; that is, don't cross up grounded leads with signal leads.

Your new interface does not operate in exactly the same way as does the Commodore's data recorder, but with a little practice you will learn to use the toggle switch along with the PLAY button on the audio recorder.

FIGURE 7–9
Pushbutton option for the recorder-ready switch (top) and a monitor modification that allows you to hear the digitized data (bottom).





For more practice, try the experiment in the Commodore 64's owner's manual that describes saving and loading data from program control.

Interface Extras

Figure 7–9 offers an alternative to the toggle switch. By using the remaining two NAND gates of the 4083 wired as an RS flip-flop, you can control the interface with push buttons. Another trick that you might like and we have found to be practical is to connect a 270-ohm resistor across the input earphone jack inside the tape recorder. This allows you to hear, at low volume, a tape as it is being loaded. (You may not want to do this if the recorder is new and still under warranty.) A diagram is given for this additional connection in Figure 7–9.

8

PROGRAMMING EPROMS

It is convenient to place frequently used programs into a ROM (read-only memory) because a program or subroutine in ROM is always available to the system. It need not be loaded from tape when the system is powered-up and will not be lost after a power glitch. It is even possible to place an auto-starting machine language or BASIC program into ROM, so that the program automatically starts itself whenever the power is turned on. In this chapter we will present the hardware and software required to program ROMs for your Commodore 64.

About ROMS, PROMS, EPROMS, and EEPROMs

The buzzword ROM applies to all varieties of read-only memories. Many chips of widely differing characteristics are currently used for permanent storage of data or programs. One of the earliest was the fusible-link ROM, generally sold under the name PROM (programmable ROM). The data was entered by blowing or not blowing individual fuses that correspond to the individual data bits. Like the fuse on your electric power circuits, any fuses in this type PROM that are blown cannot be unblown. Thus, it can only be programmed once. Mask-programmed ROMs have the data entered during a stage of manufacture of the silicon chip. This type of

ROM is suitable for large production runs only, because of the high set-up cost. Your Commodore 64 uses mask-programmed ROMs.

EPROMs (Erasable PROMs) are low-cost and ideal for small production runs and one-of-a-kind projects. They range in capacity from ¼K to 16K bytes or more. EPROMs come from the factory in the erased condition, generally with all bits set to 1s. They can be programmed in the field using electrical pulses to convert 1s to 0s. EPROMs can be erased by exposure to ultraviolet light through a quartz window. The serious EPROM user should obtain a safe UV light, available for $50–$100, to erase EPROMs for reprogramming. This chapter covers the programming of this type of ROM only, and henceforth we will use ROM to mean EPROM.

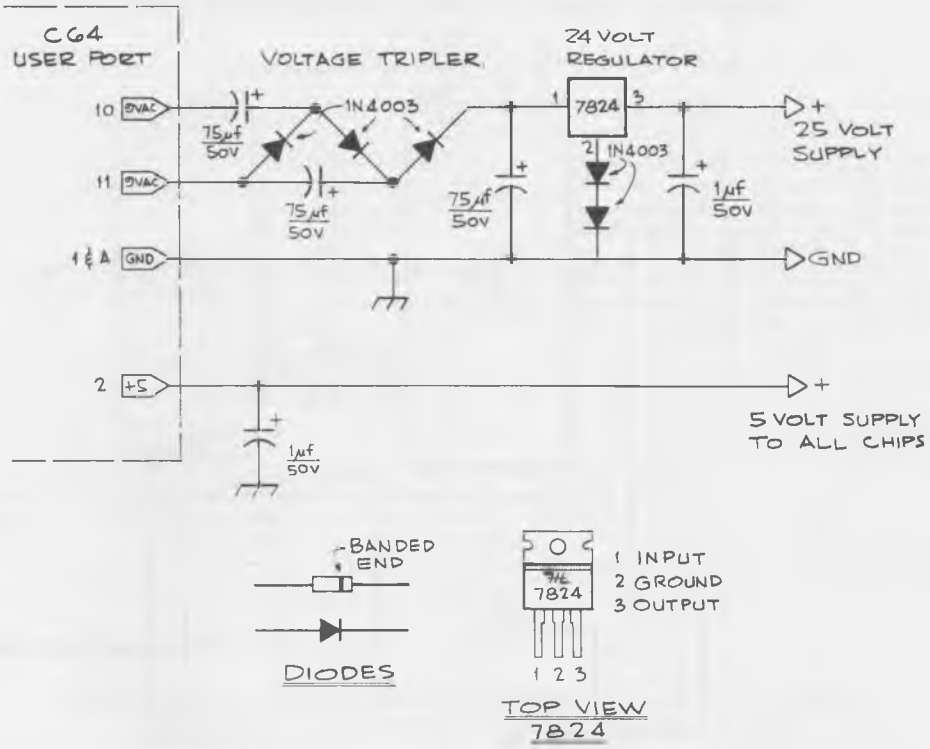
EEPROMs (electrically erasable PROMs) are a recently developed set of devices that are programmed in similar fashion to EPROMs but can be erased by an electrical pulse. Such ROMs are useful when in-circuit reprogramming is desired. Some are byte-erasable, and others are completely erased by the erase pulse. A large amount of support circuitry is required for all of the EEPROMs except the newest of these devices, which have enough built-in circuitry to make them appear as slow RAM to the system. Because of their complexity, they will not be considered here, but they are sure to become more popular as simpler types become available.

Programming Hardware

The hardware required to program ROMs using the Commodore 64 is simple because of the availability of the parallel ports on the USER PORT connector. Three integrated circuits, a circuit to generate 25 volts, a ROM socket, and two tablespoonfuls of miscellaneous parts are all of the parts required for the ROM programmer. The circuitry is shown in Figures 8–1, 8–2, and 8–3. A complete parts list for the project is given below. Total cost should be about $30, plus $10 if a ZIF (Zero Insertion Force) socket is used.

We built our circuit on a 4.5-inch by 4.5-inch card (VECTOR 3662–5) having a 22/44 connector positioned on one end as gold-plated fingers. The board has no other foil, but it does have a pattern of holes on a 0.1-inch square grid. We soldered a 12/24-position edge connector to the first 12 positions of the card so that the extra pins would extend to the left side (as viewed from the front) of the Commodore 64 and thus not interfere with the other I/O connectors. It was necessary to cut about 1/8-inch off the unused connectors so that the card and connector would fit properly into the recess for the Commodore 64 user port.

FIGURE 8–1
Power supplies for the ROM programmer.

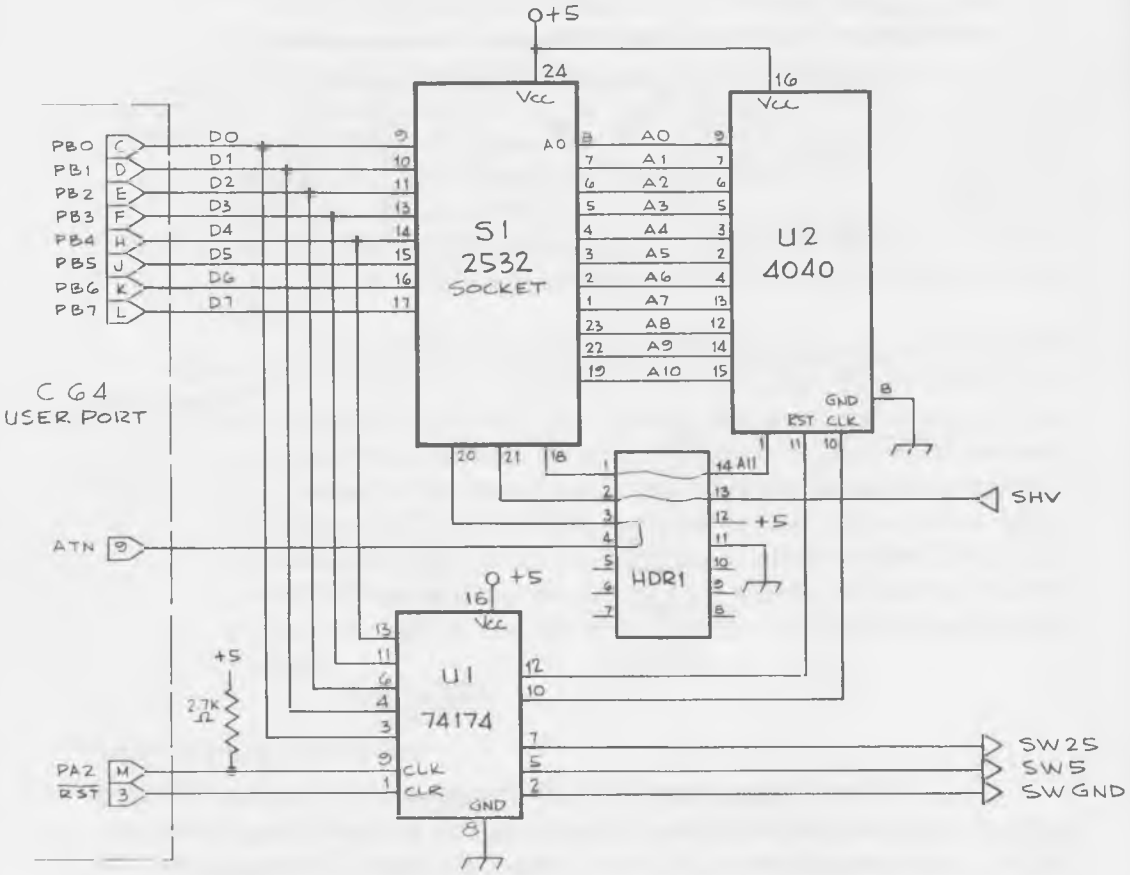


We ran heavy ground and +5V busses along the bottom of the board for convenient tie points. Discrete parts may be placed through holes and soldered together on the bottom side of the board. We suggest that you use wire-wrap sockets for the ICs, the header module (optional), and the ROM socket. These wire-wrapped sockets are connected by special wire using a wire-wrap tool.

PARTS LIST

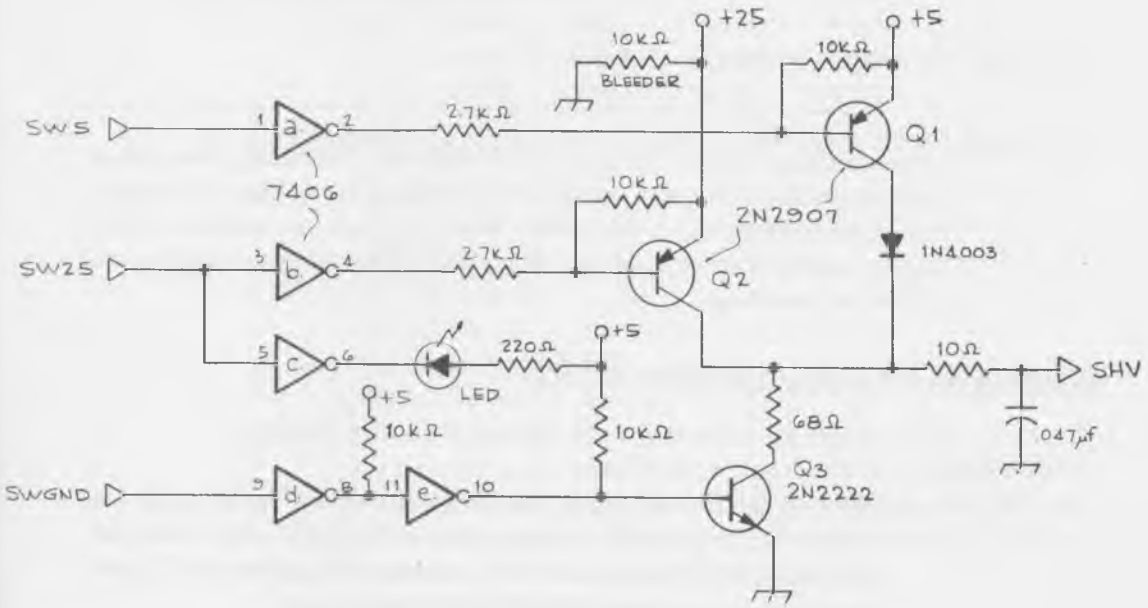
1 Vector 4.5 x 4.5 circuit card #3662–5 or equivalent
1 Edge connector w/solder eyelets, 12/24 position
1 4040 12-stage binary counter
1 74LS174 6-bit latch
1 7406 hex-inverting buffer (open-collector)
1 14-pin wire-wrap socket
3 16-pin wire-wrap sockets
1 24-pin wire-wrap socket. For extended use, substitute ZIF (zero insertion force) socket ($10).

FIGURE 8–2
Counter and ROM socket for the ROM programmer. The header socket allows you to change ROM types by simply changing header modules.



1 7824 voltage regulator, TO–220 case
6 IN4003 rectifier diodes
3 75-mfd 50V electrolytic capacitors
1 .047-mfd 50V capacitor
2 1 mfd 50V tantalum capacitors
1 2N2222 transistor (Q3)
2 2N2907 transistors (Q1, Q2)
1 red LED
5 10,000-ohm ¼W resistor
1 220-ohm ¼W resistor
1 68-ohm ¼W resistor
1 10-ohm ¼W resistor
3 2700-ohm ¼W resistor
Optional: one or more header plugs (16-pin)

FIGURE 8–3
Pulse driver for the ROM programmer.



How the Hardware Works

The set of three diodes and three capacitors (Figure 8–1) constitutes a voltage tripler that generates 33 volts DC from the 9VAC that is available on the Commodore 64 user port. This DC voltage is regulated to 25V by the 7824, which has two diodes in series in its ground leg to raise its output voltage from 24 to 25.

Addresses are provided to the ROM directly from the 4040 12-stage binary counter (Figure 8–2). When programming is about to start, the counter's RESET line is briefly activated by strobing control words into the latch. After each byte is programmed, the counter is advanced one step by bringing its CLOCK momentarily high. Data is supplied to the ROM using all 8 lines of port B. The SHV (Switched High Voltage) line goes to the ROM pin required by the ROM type selected. The SHV can be set to 0, 5, or 25V under software control. The SHV can be hard-wired to pin 21 for 2532-type ROMs or can go through the optional header module, into which different header plugs can be placed for various ROM types. Address line A11 is also available for suitable routing on the header module. The ATN signal is provided for controlling the program pulse and can also be used as A12 for programming 8K ROMs.

The pulse-switching circuit (Figure 8–3) uses 3 outputs of the 74LS174 latch to control the SHV signal via transistors Q1, Q2, and Q3. The following table shows the voltage resulting from various latch outputs:

Pin 2	*Pin 5*	*Pin 7*	*SHV*
LOW	LOW	LOW	Floating (power-on condition)
LOW	LOW	HIGH	25V
LOW	HIGH	LOW	5V
HIGH	LOW	LOW	0V

This switching system is designed to be fail-safe. Normally, the system will not inadvertently turn on the 25V programming voltage. Misoperation of the software can cause failure, however. The Commodore 64 RST line is connected to the CLR pin of the 74LS174 so that the outputs are all low at power-up.

Building and Testing the ROM Burner

We suggest the following construction and testing strategy:

1. Build up the 25-volt supply, but do not connect it to the switching section. Watch carefully for the polarity of the diodes and electrolytic capacitors. Plug the card into the Commodore 64 and test for 33 volts going into the regulator and 25 volts coming out.
2. Wire the three ICs, the ROM socket, and the header module and test them with a logic probe as you type in the following keyboard commands:

   a. Check out the Ports

   ```
   POKE 56578, 255
   POKE 56579, 255
   (this makes all bits of ports A and B outputs)
   POKE 56577, 0
   (all port B lines should go low)
   POKE 56577, 255
   (all port B lines should go high)
   POKE 56576, 0
   (ROM pin 20 should go low. Pin 9 of the 74LS174 should go
   low)
   POKE 56576, 255
   (both these pins should reverse to high
   ```

   b. Check out the Counter

   ```
   POKE 56578, 255
   POKE 56579, 255
   POKE 56577, 16
   POKE 56576, 147
   POKE 56576, 151
   ```

```
POKE 56577, 0
POKE 56576, 147
POKE 56576, 151
```
(this sequence should bring all counter outputs and ROM address lines low)

After clearing the counter as above, enter and RUN this program:

```
 10 POKE 56577, 8
 20 GOSUB 100
 30 POKE 56577, 0
 40 GOSUB 100
 50 GOTO 10
100 POKE 56576, 147
110 POKE 56576, 151
120 RETURN
```
(this should produce square waves on all the ROM address pins and 4040 outputs, fast pulses on AO, and progressively slower pulses on the higher address lines. It will take about 4 minutes for all to go through a complete cycle)
PRESS RUN/STOP
(this should leave random highs and lows on the address lines A0–A11)

3. Wire the pulse-switching circuit and connect up the 25-volt supply.

```
POKE 56578, 255
POKE 56579, 255
POKE 56577, 1
POKE 56576, 147
POKE 56576, 151
```
(pin 21 of the ROM should be at ground voltage)
```
POKE 56577, 2
POKE 56576, 147
POKE 56576, 151
```
(pin 21 of the ROM should go to about 4 volts)
```
POKE 56577, 4
POKE 56576, 147
POKE 56576, 151
```
(pin 21 of the ROM should go to 25V and the red LED should come on)

If the Commodore 64 should cease to function normally during any of the above tests, turn off power immediately and check the ROM board for miswiring, shorts, bad components, and so on. If your board passed

all the tests above, you are now ready to enter the software required to program ROMs.

Selection of an EPROM

The hardware we describe is capable of programming many different EPROMs. The manufacturers have standardized the pinouts for most of the 24-pin ROMs and, in some cases, have kept the pinouts compatible for 28-pin ROMs. The only differences between most 24-pin ROMs are for pins 18, 19, 20, and 21; the data lines and all except the uppermost address lines (A10 and higher) are on the same pin numbers.

Your Commodore 64 uses two 8K ROMs for the operating system and one 4K ROM for the character sets. Game or utility cartridges that plug into the expansion port use 4 or 8K ROMs. All are 24-pin ROMs, and, for convenience, we would like to use a compatible device. The MCM68764 (see Table 8–1) is pin-for-pin compatible with the Commodore 64 system's 8K ROMs and some of Commodore 64's game packs. Unfortunately, its present cost of about $40 at the time of this writing is somewhat high for experimenters. The 4K 2732 and 2732A have different printouts on pins 18–21 from that used in the Commodore 64 cartridges. With adaptors, however, they are suitable. The 4K 2532 from Motorola and Texas Instruments meets our requirements of compatibility and easy programmability, and it is very cost-effective. We will therefore describe in detail only the software used to program and read the 2532 ROM. While the hardware can be used for any ROM in Table 8–1 (except the 28-pin one), the software and header plug must be changed for each of the other ROMs because the "recipe" for each ROM is different. For example, the TTL level pulse that controls programming may in one case have to be low and in another case high. Disaster in the form of ROM destruction is the result if this sort of minor detail is overlooked.

TABLE 8-1: Programmable EPROMs

EPROM TYPE	*MANUFACTURERS*	*#BYTES*	*APPROXIMATE PRICE**	*COMMENTS*
2716 (5v.)	Various	2K	$4–7	Small size
2732 or 2732A	Intel	4K	5–10	Noncompatible pinout
MCM2532 or	Motorola	4K	5–10	Our choice
TMS2532	Texas Instruments	4K	5–10	
2764	Intel	8K	10–15	28 pins
MCM68764	Motorola	8K	20–40	High price

*Prices at the time of this writing (Spring 1984).

Programming the 2532

The particular "recipe" for programming a 2532 requires the following steps:

1. With the normal 5V power applied, set pin 20 (E/PROGR) (see pinout in Fig. 8–4) high (+5 volts) to prevent any bytes from being inadvertently programmed, and so that the application of the high programming voltage will not ruin the ROM.
2. Apply 25V to pin 21 and hold this voltage until the programming sequence is completed. This step can be simultaneous with step 1.
3. Place the address to be programmed on the 12 address lines and place the data to be programmed on the eight data lines.
4. Bring pin 20 (E/PROGR) to 0 volts for exactly 50 milliseconds and then return to 5.0 volts. This step does the actual programming.
5. Repeat steps 3 and 4 until all desired addresses are programmed.
6. While pin 20 is still high, remove the 25V programming voltage on pin 21 by bringing it into the 0–5V range. At this stage, the ROM can be removed from the socket, although we prefer simply to switch off the Commodore 64 entirely. Alternately, you may verify the ROM by going into read mode before removing it from the socket.

Unlike some earlier ROMs, the bytes of the 2532 can be programmed in any order you desire, and as many or as few bytes can be programmed as you require. Any address not programmed will retain its previous contents and can be programmed at a later session. Those bytes that have not been programmed since erasure will contain all ones, that is, contain a value of decimal 255 or $FF. You may need to change only a single bit from 1 to 0, which is feasible. Remember, however, that changing any bit from 0 to 1 requires complete erasure of the ROM under UV light and complete reprogramming.

Due to the constraints of our simple hardware, we can only clear our address counter or step the addresses sequentially from 0 to 4095. Therefore, to reach a particular address we want to program, our software must clear the counter and then issue the required number of step pulses, omitting the programming pulse for those bytes we want to skip over. With our hardware, it is most convenient to program our addresses in increasing order; otherwise, programming would be extremely time-consuming. Even under the sequential mode, programming all 4096 bytes still requires 3–12 minutes.

LISTING 8-1
Machine code for 2532 EPROM

```
:ASM

                    1000 * PROGRAM 2532 IN C64
                    1010         .OR $C000
                    1020         .TA $0800
                    1030 *--------------------------------
DD00-               1040 CIA2    .EQ 56576     $DD00
DD00-               1050 PORTA   .EQ CIA2+0    PORT A I/O REGISTER
DD01-               1060 PORTB   .EQ CIA2+1    PORT B I/O REGISTER
DD02-               1070 DDRA    .EQ CIA2+2    A DATA DIREC.REG.
DD03-               1080 DDRB    .EQ CIA2+3    B DATA DIREC.REG.
                    1090 *
00FB-               1100 RAMBEG .EQ $FB        ADH OF DATA - USER-SET
00FC-               1110 PAGCNT .EQ $FC        PAGE COUNTER
00FD-               1120 ADRPTR .EQ $FD & FE ADDRESS POINTER
                    1130 *
0318-               1140 NMIVEC .EQ $0318      SYSTEM NMI VECTOR
0334-               1150 RTILOC .EQ $0334      WE PUT RTI CODE HERE
                    1160 *--------------------------------
C000- AD 00 DD      1170 START   LDA PORTA     GET OLD PORT A
C003- 48            1180         PHA           SAVE ON STACK
C004- AD 02 DD      1190         LDA DDRA      GET OLD DDRA
C007- 48            1200         PHA           SAVE ON STACK
C008- A9 FF         1210         LDA #$FF      ALL OUTPUT
C00A- 8D 02 DD      1220         STA DDRA
C00D- 8D 03 DD      1230         STA DDRB
C010- A9 00         1240         LDA #$00
C012- 8D 01 DD      1250         STA PORTB     ALL LOW
C015- 85 FC         1260         STA PAGCNT    CLEAR PAGE COUNT
C017- 85 FD         1270         STA ADRPTR    BEGINNING OF PAGE
C019- A2 10         1280         LDX #$10      NOW RESET COUNTER
C01B- 8E 01 DD      1290         STX PORTB
C01E- AD 00 DD      1300         LDA PORTA     STROBE IT IN
C021- 29 FB         1310         AND #$FB
C023- 8D 00 DD      1320         STA PORTA
C026- 09 04         1330         ORA #$04
C028- 8D 00 DD      1340         STA PORTA
C02B- A2 00         1350         LDX #$00
C02D- 8E 01 DD      1360         STX PORTB     RESET OFF
C030- 29 FB         1370         AND #$FB      STROBE IT IN
C032- 8D 00 DD      1380         STA PORTA
C035- 09 04         1390         ORA #$04
C037- 8D 00 DD      1400         STA PORTA
C03A- A5 FB         1410         LDA RAMBEG    POINT TO ORIGINAL
C03C- 85 FE         1420         STA ADRPTR+1
C03E- A9 40         1430         LDA #$40      RTI OPCODE
C040- 8D 34 03      1440         STA RTILOC
C043- AD 18 03      1450         LDA NMIVEC    SAVE OLD NMI
C046- 48            1460         PHA           ON STACK
C047- AD 19 03      1470         LDA NMIVEC+1
C04A- 48            1480         PHA
C04B- A9 34         1490         LDA #RTILOC   DISABLE NMI'S
C04D- 8D 18 03      1500         STA NMIVEC
C050- A9 03         1510         LDA /RTILOC
C052- 8D 19 03      1520         STA NMIVEC+1
C055- 78            1530         SEI           NO IRQ'S
C056- AD 00 DD      1550 D02532 LDA PORTA
C059- 29 F7         1560         AND #$F7      BRING ROM PIN 20 HIGH
C05B- 8D 00 DD      1570         STA PORTA
C05E- A2 04         1580         LDX #$04      TURN ON 25V.
C060- 8E 01 DD      1590         STX PORTB
C063- AD 00 DD      1600         LDA PORTA     STROBE IT IN
C066- 29 FB         1610         AND #$FB
C068- 8D 00 DD      1620         STA PORTA
C06B- 09 04         1630         ORA #$04
```

```
C06D- 8D 00 DD 1640         STA PORTA
C070- A0 00    1650         LDY #0
C072- A5 FC    1660 ZAPLP   LDA PAGCNT
C074- C9 10    1670         CMP ##10      DONE 16 PAGES ?
C076- F0 4F    1680         BEQ DONE
C078- B1 FD    1690 ZAPLP2  LDA (ADRPTR),Y DATA BYTE
C07A- C9 FF    1700         CMP ##FF
C07C- F0 1F    1710         BEQ STEP      DONT PGM FF'S
C07E- 8D 01 DD 1720         STA PORTB     DATA TO PORTB
C081- AD 00 DD 1730 ZAP     LDA PORTA
C084- 09 08    1740         ORA ##08      PROGRAM !
C086- 8D 00 DD 1750         STA PORTA
C089- 98       1760         TYA           SAVE Y
C08A- A2 28    1770         LDX #40
C08C- A0 00    1780         LDY #0
C08E- 88       1790 LP50    DEY           DELAY 50 MS
C08F- D0 FD    1800         BNE LP50
C091- CA       1810         DEX
C092- D0 FA    1820         BNE LP50
C094- A8       1830         TAY           RESTORE Y
C095- AD 00 DD 1840         LDA PORTA
C098- 29 F7    1850         AND ##F7
C09A- 8D 00 DD 1860         STA PORTA     PGM PULSE OFF
C09D- A2 0C    1870 STEP    LDX ##0C      KEEP 25V, BUMP THE COUNTER
C09F- 8E 01 DD 1880         STX PORTB
C0A2- AD 00 DD 1890         LDA PORTA     STROBE IT IN
C0A5- 29 FB    1900         AND ##FB
C0A7- 8D 00 DD 1910         STA PORTA
C0AA- 09 04    1920         ORA ##04
C0AC- 8D 00 DD 1930         STA PORTA
C0AF- A2 04    1940         LDX ##04      COUNT OFF
C0B1- 8E 01 DD 1950         STX PORTB
C0B4- 29 FB    1960         AND ##FB      STROBE IT
C0B6- 8D 00 DD 1970         STA PORTA
C0B9- 09 04    1980         ORA ##04
C0BB- 8D 00 DD 1990         STA PORTA
C0BE- C8       2000 INCADR  INY           INCREMENT ADDRESS
C0BF- D0 B7    2010         BNE ZAPLP2
C0C1- E6 FE    2020         INC ADRPTR+1
C0C3- E6 FC    2030         INC PAGCNT
C0C5- D0 AB    2040         BNE ZAPLP     ALWAYS BRANCH
C0C7- A2 00    2060 DONE    LDX ##00      ALL OFF
C0C9- 8E 01 DD 2070         STX PORTB
C0CC- AD 00 DD 2080         LDA PORTA     STROBE IT IN
C0CF- 29 FB    2090         AND ##FB
C0D1- 8D 00 DD 2100         STA PORTA
C0D4- 09 04    2110         ORA ##04
C0D6- 8D 00 DD 2120         STA PORTA
C0D9- 68       2130         PLA           RESTORE OLD NMI
C0DA- 8D 19 03 2140         STA NMIVEC+1
C0DD- 68       2150         PLA
C0DE- 8D 18 03 2160         STA NMIVEC
C0E1- 68       2170         PLA           RESTORE OLD PORT A
C0E2- 8D 02 DD 2180         STA DDRA
C0E5- 68       2190         PLA
C0E6- 8D 00 DD 2200         STA PORTA
C0E9- 58       2210         CLI           ALLOW IRQ'S
C0EA- 00       2220         BRK           CHANGE TO 60-RTS FOR
               2230 *                     CALL FROM BASIC
               2240 *---------------------------------
```

Software for the Programmer

The software we will present is designed to be the minimum required to do the ROM programming (and ROM reading) job and will be adequate

for the casual user. With this software as a core, you can develop more elegant programs that will satisfy the serious programmer. The software is in the form of machine language and BASIC programs; the former are generally four times faster but are harder to understand.

Listing 8–1 is a machine language routine that can be used to copy any 4K block of data existing in ROM or RAM into a 2532-type ROM. The data must start on a page boundary, that is, the starting address must be $XX00 or evenly divisible by 256 decimal. The listing shows the program located at $C000, but it can be relocated anywhere in RAM (or ROM) without change. If you intend to call the routine from BASIC, the last byte should be $60 (RTS) instead of $00 (BRK).

The program sets up the CIA #1 ports to be outputs and brings all outputs low. The counter is reset, pointers are set up, interrupts are disabled, and then programming begins. Once the program is called, pressing RUN/STOP and RESTORE will no longer work, so you must sit on your hands for the 3½ minutes required to run through 4K.

The code at $C056 turns on the high voltage and brings pin 20 of the ROM high. The programming pulse is applied at $C086 and after 50 ms delay is turned off at $C09A. The counter is incremented at $C09D, and the program loops 4096 times. Upon completion, the voltages are reduced to safe levels, and control is returned to the keyboard.

To use this program, the byte at $FB (decimal 251) must first be set to the high-order address of the start of data. For example, if you are copying a ROM at $A000, you must store $A0 (decimal 160) into location $FB. Thus, to activate the program from BASIC, enter the following commands:

```
POKE 251,160
SYS 12*4096
```

The 160 can be changed to the high-order address of other data locations. If you have relocated our machine code, change the 12*4096 accordingly.

This machine language system is best entered and called from a machine language monitor, such as that included in Commodore's disk bonus-pack.

The BASIC language in the Commodore 64 is fast enough to operate the ROM programmer because the 50 ms. pulse can be accurately controlled by a BASIC program. Listing 8–2 gives a BASIC program that has some embellishments over the machine language version. It prompts you to enter pairs of numbers to describe a map of those bytes you want to program and where the data is located. This feature allows you to program any portion of the ROM as well as leave any portion unprogrammed. The first number in each pair is the decimal value of the relative address

LISTING 8–2
BASIC EPROM program

```
100  PRINT "ENTER PAIRS OF NUMBERS"
110  PRINT "LAST SHOULD BE 4096,0"
120  PRINT "SEE INSTRUCTIONS"
130  DIM B(16,1):B = 0
140  INPUT B(B,0),B(B,1)
150  IF B(B,0) = 4096 AND B(B,1) = 0 THEN 180
160 B = B + 1: IF B > 16 THEN  STOP
170  GOTO 140
180  PRINT "INSERT 2532 IN "
190  PRINT "SOCKET AND PRESS 9 "
200  PRINT "THEN RETURN KEY"
210  INPUT K: IF K <  > 9 THEN 180
220 PA = 56576:PB = PA + 1
230 DA = PA + 2:DB = PA + 3
240 OD =  PEEK (DA):OP =  PEEK (PA)
250  POKE DA,255: POKE DB,255
260  POKE PB,0
270 CM = 16: GOSUB 800
280 CM = 0: GOSUB 800
290  POKE PA,( PEEK (PA) AND 247)
300 CM = 4: GOSUB 800
310 B = 0:K = 0
320  FOR I = 0 TO 4095
330  IF B(B,1) = 0 THEN 400
340 BY =  PEEK (B(B,1) + K)
350  IF BY = 255 THEN 400
360  POKE PB,BY
370  POKE PA,( PEEK (PA) OR 8)
380  FOR BY = 1 TO 45: NEXT
390  POKE PA,( PEEK (PA) AND 247)
400 CM = 12: GOSUB 800
410 CM = 4: GOSUB 800
420 K = K + 1
430  IF I + 1 = B(B + 1,0) THEN K = 0:B = B + 1
440  PRINT I: NEXT I
450 CM = 0: GOSUB 800
460  POKE DA,OD: POKE PA,OP
470  PRINT "DONE": END
800  POKE PB,CM
810  POKE PA,( PEEK (PA) AND 251)
820  POKE PA,( PEEK (PA) OR 4)
830  RETURN
```

in the ROM. This number should be 0 for the first pair and 4096 for the last pair; each succeeding ROM address should be larger than the previous one. The second number in a pair is the decimal location of the start of data for this block. If the second number is 0, the block is not programmed.

The simplest example is for programming the entire ROM from a single contiguous block of data. Entries

```
0,40960
4096,0
```

will tell the program to start getting data from address 40960 ($A000) and keep programming for all 4096 bytes of the ROM. Note that 4095 is the last byte of the ROM because the first address is 0, not 1. Thus, the command directs the programming to stop at byte 4096.

A more complex example illustrates byte-skipping and multiple data locations, as follows:

ENTRY	*EXPLANATION*
0,0	Do not program the first 3 bytes.
3,4099	Program starting at byte 3 with data starting at address 4099.
256,24832	Program starting at byte 256 with data starting at address 24832.
512,0	Do not program starting at byte 512.
4096,0	End of entries.

If you have a complex map such as this example, it would be wise to sketch it out on paper before entering the data. After starting the programming procedure, there is no turning back.

Be sure to keep your hands away from the keyboard once this program starts, because RUN/STOP is still enabled. If you stop the program, the odds are that the 25 volts will be ON and your ROM will be destroyed. (Jog around the block, spank the kids, smoke if you want to, and so on, for the 12 minutes required.)

LINE-BY-LINE EXPLANATION OF LISTING 8–2

Lines 100–120 prompt the user.
Line 130 allows for 16 pairs of numbers to be used in the memory map.
Lines 140–170 input the pairs of numbers.
Lines 180–210 prompt and request a 9 to be entered.
Line 220 sets the port addresses.
Line 230 sets the addresses of the data-direction register.
Line 240 saves the port A data and DDR registers.
Line 250 makes ports A and B both outputs.
Line 260 sets port B outputs all low.
Line 270 brings the counter reset line high.
Line 280 brings the counter reset line low.
Line 290 brings ROM pin 20 high.
Line 300 turns on the 25V to pin 21.
Line 310 sets B (block counter) to 0 and sets K (offset in current block) to 0.
Line 320 starts a loop to process 4096 bytes.
Line 330 bypasses bytes which are not to be programmed.
Line 340 fetches a byte from data location.
Line 350 bypasses bytes which are all ones.
Line 360 stores the data in port B.
Line 370 turns on the program pulse.

Line 380 delays for 50 msec.
Line 390 turns off the program pulse.
Line 400 brings the counter's CNT pin high.
Line 410 brings the counter's CNT pin low.
Line 420 increments the position in the block.
Line 430 tests for block done. If done, advance to next block.
Line 440 prints status report and goes to next byte.
Line 450 puts low on all latch outputs.
Line 460 restores old port A values.
Line 470 winds it up.
Line 800 stores the current command in port B.
Line 810 brings the latch's CLK line low.
Line 820 returns the latch's CLK line high.
Line 830 returns to caller.

Reading the ROM

A simple program to read a 2532 ROM into RAM is given in Listing 8–3. We leave it to the reader to adapt this routine as a VERIFY routine.

LINE-BY-LINE EXPLANATION OF LISTING 8–3.
Line 500 establishes a place in RAM to store the data.
Line 510 sets addresses of ports A and B.
Line 520 sets addresses of data-direction registers.
Line 530 saves old port A data and DDR.

LISTING 8–3
Reading the ROM (BASIC)

```
500 RAM = 4096
510 PA = 56576:PB = PA + 1
520 DA = PA + 2:DB = PA + 3
530 OD =  PEEK (DA):OP =  PEEK (PA)
535  POKE DA,255: POKE DB,255
540  POKE PB,0
550 CM = 16: GOSUB 800
560 CM = 0: GOSUB 800
570  FOR I = 0 TO 4095
580  POKE PA,( PEEK (PA) OR 8)
590  POKE DB,0
600  POKE RAM + I, PEEK (PB)
610  POKE DB,255
615  POKE PA,( PEEK (PA) AND 247)
620 CM = 8: GOSUB 800
630 CM = 0: GOSUB 800
640  PRINT I: NEXT I
650  POKE DA,OD: POKE PA,OP
660  END
800  POKE PB,CM
810  POKE PA,( PEEK (PA) AND 251)
820  POKE PA,( PEEK (PA) OR 4)
830  RETURN
```

Line 535 sets both ports as outputs.
Line 540 sets port B outputs low.
Line 550 sends a high to the counter RST pin.
Line 560 sends a low to the counter RST pin.
Line 570 sets up to read 4096 bytes.
Line 580 brings ROM pin 20 low to select the ROM.
Line 590 changes port B to all inputs.
Line 600 reads the data byte and stores it in RAM.
Line 610 changes port B to all outputs.
Line 615 brings ROM pin 20 high to de-select the ROM.
Line 620 bumps the counter with a high.
Line 630 brings CNT pin low.
Line 640 prints status and loops if not done.
Line 650 restores old port A values.
Line 660 stops the program.
Lines 800–830 are the same as in Listing 8–2.

If you can tolerate a blank screen for several minutes, omit the PRINT statements, because number evaluation and screen scrolling are time-consuming. It would be an interesting challenge to speed up the BASIC program by suitable revisions.

The user can be more casual in operating programs that do ROM reads than do ROM writes, since the 25-volt programming voltage is never applied during our read program.

Converting a Game Cartridge to an EPROM Cartridge

If you are tired of playing the old game you bought last Christmas, you can do some interesting things with the printed circuit board inside. Open the plastic cover and remove the board inside. You should see one or two ROMs on the board. The particular one we worked on was VISIBLE SOLAR SYSTEM from Commodore. It had one ROM and a position to hold another ROM. The significant connections are shown in Figure 8–4. (The remaining connections are the standard power, ground, address, and data lines.)

You will note that the board was made as versatile as possible by the provision of several places where board traces can be cut and/or connected with a blob of solder. Figure 8–4 shows the original connections, with traces at J2 and J5 (yes, the board is conveniently labelled). J1 and J2 determine whether the ROM occupies low ROM space ($8000–9FFF) or high ROM space ($E000–FFFF). The Solar System ROM was wired for the latter. Occupying high ROM space overlaps the Kernel ROM, which is disabled by the combination at J3, J4, and J5; originally the connec-

FIGURE 8–4
Chip data for the six most popular EPROMs. Courtesy of Motorola.

MOTOROLA

MCM2532

4096 × 8-BIT UV ERASABLE PROM

The MCM2532 is a 32,768-bit Erasable and Electrically Reprogrammable PROM designed for system debug usage and similar applications requiring nonvolatile memory that could be reprogrammed periodically. The transparent window in the package allows the memory content to be erased with ultraviolet light.

For ease of use, the device operates from a single power supply and has static power-down mode. Pin-for-pin compatible mask programmable ROMs are available for large volume production runs of systems initially using the MCM2532.

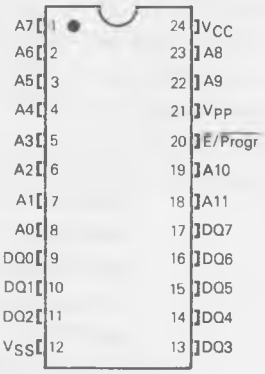
- Single +5 V Power Supply
- Organized as 4096 Bytes of 8 Bits
- Automatic Power-Down Mode (Standby)
- Fully Static Operation (No Clocks)
- TTL Compatible During Both Read and Program
- Maximum Access Time = 450 ns MCM2532
- Pin Compatible with MCM68A332 Mask Programmable ROMs
- Power MCM2532
  - Active — 150 mA Max
  - Standby — 25 mA Max

MOS
(N-CHANNEL, SILICON-GATE)

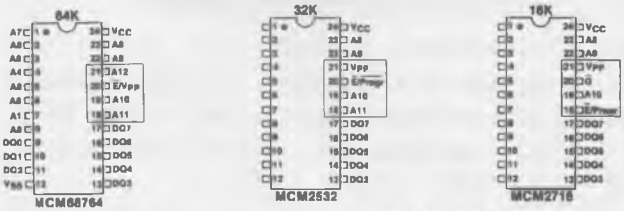
4096 × 8-BIT
UV ERASABLE PROM

C SUFFIX
FRIT-SEAL CERAMIC PACKAGE
CASE 623A-02

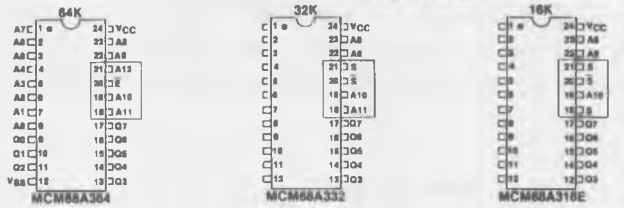
PIN ASSIGNMENT



MOTOROLA'S PIN-COMPATIBLE EPROM FAMILY



MOTOROLA'S PIN-COMPATIBLE ROM FAMILY



INDUSTRY STANDARD PINOUTS

*PIN NAMES	
A	Address
DQ	Data Input/Output
$\overline{E}$/Progr	Dual Function Enable (Power-Down/Program Pulse)

*New Industry standard nomenclature

tions grounded the GAME signal feedback to the computer and pulled up the EXROM signal (in our case, the IO2 line simply acts as a pull-up to 5V).

We wanted an auto-start system, which must have ROM at $8000, so we cut the traces at J2, J3, and J5 and put blobs of solder at J1 and J4. A 2532 can now be used to replace the original ROM. We carefully removed the original, using solder-removing braid, and put a 24-pin low-profile socket in its place to hold the 2532. Actually, an 8K 68764 will work in the same socket. It will occupy the full $8000-9FFF address space, while the 2532 will have two "copies" of the same bytes, which in most applications will do no harm.

Of course, you can fabricate a wire-wrap board that can do the same job. Either way, you may place frequently used routines, such as a printer routine, in the ROM so that it is available to you with a simple SYS call.

How to Make an Auto-Start System

Commodore provided for automatic program startup in the Commodore 64 by including a routine in the Kernel ROM that tests for the presence of special "signature" bytes in the cartridge ROM when the system is powered up or when RUN/STOP/RESTORE is pressed. The expansion ROM, which is usually a game or utility pack, must reside at address $8000 (decimal 32768). If the plug-in ROM has the arbitrary signature pattern

$8004	$C3
8005	C2
8006	CD
8007	38
8008	30

then normal startup does not take place. Instead, the program control is transferred to the address in the two bytes at $8000 and $8001 (low-order byte first, in standard 6510 notation). Similarly, when RUN/STOP/RESTORE is pressed, the signature test takes place, and, if the signature is present, control is passed to the address contained in the two bytes $8002 and $8003. Normally, these two addresses will point to a location in the expansion ROM.

You can create your own auto-start machine language ROM by simply following the above procedure. Assume that you want to start your machine language immediately after the signature and that you want RUN/STOP/RESTORE to act the same as power-up. The contents of your new ROM will start as follows:

8000	09	ADL of your program
8001	80	ADH of your program

8002	09	ADL of your program
8003	80	ADH of your program
8004	C3	Signature byte
8005	C2	Signature byte
8006	CD	Signature byte
8007	38	Signature byte
8008	30	Signature byte
8009	Your code starts here	

Remember that no initialization has taken place when your program starts up. You may use Kernel ROM subroutines to initialize the VIC chip, CIAs, and so on, if you wish. The use of some of these kernel routines is illustrated in the BASIC program auto-start example below. The new 2532 ROM must be wired to respond to the address range $8000–8FFF by having its chip-select pin 20 tied to connector pin 11 (ROML).

A BASIC Language Auto-Start ROM

The possibility of having a BASIC program in an auto-start ROM leads to many exciting possibilities. In industrial work, the Commodore 64 could serve as a dedicated process controller, laboratory controller, and so on. It could automatically function as a controller immediately on power-up and remain in the supervisory mode indefinitely.

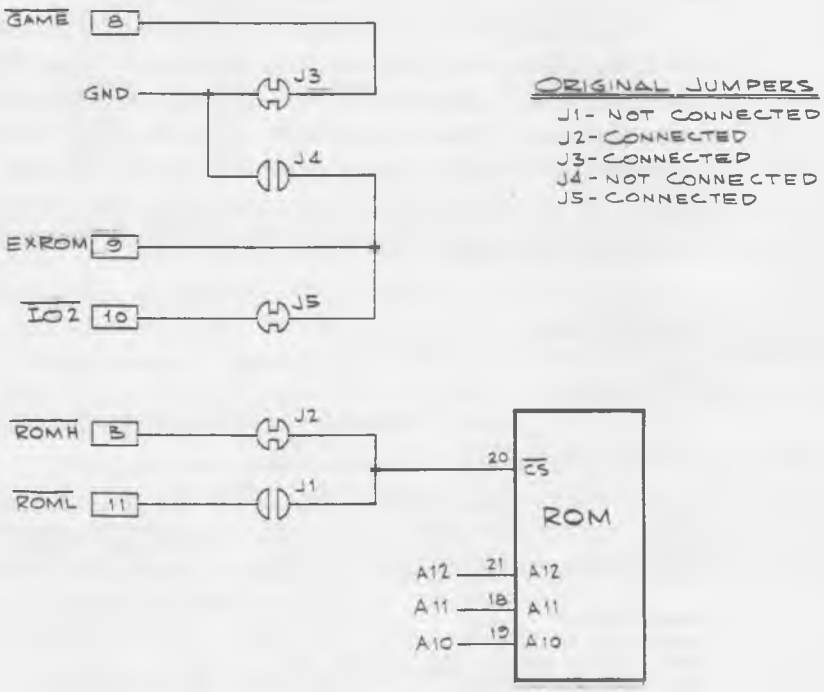
In this configuration, your ROM will need to have the signature bytes as well as calls to several initialization subroutines. Your ROM will also have to trick the Commodore 64 into thinking someone had typed RUN on the keyboard. It has to tell BASIC where the ROM BASIC program is located, and, finally, it must direct control to the BASIC warm start location. A typical preamble for accomplishing these steps is given in Listing 8–4. This program assumes that the BASIC program is located

LISTING 8–4
Auto-start driver

```
:ASM

                1000 * PREAMBLE FOR BASIC AUTO-START PROGRAM
                1010         .OR $8000
                1020         .TA $0800
                1030 *-----------------------------------
002B-           1040 SOB     .EQ $2B       START OF BASIC POINTER
002D-           1050 SOV     .EQ $2D       START OF VARIABLES POINTER
00C6-           1060 KEYCNT  .EQ $C6       # CHARS. IN KBD BUFFER
0277-           1070 KEYBUF  .EQ $0277     START KEY BUFFER
                1080 *-----------------------------------
8000- 0E 80     1090         .DA COLD      WHERE TO GO ON POWERUP
8002- 5E FE     1100         .DA $FE5E     RUN/STOP/RESTORE
8004- C3 C2 CD
8007- 38 30     1110         .HS C3C2CD3830 SIGNATURE
8009- 93 52 55
```

```
800C- 4E 0D     1120 RUN     .HS 9352554E0D 'RUN' LITERAL
800E- 8E 16 D0  1130 COLD    STX $D016
8011- 20 A3 FD  1140         JSR $FDA3      INIT I/O
8014- 20 50 FD  1150         JSR $FD50      INIT RAM,TAPE,SCREEN
8017- 20 15 FD  1160         JSR $FD15      MOVE HOOKS
801A- 20 5B FF  1170         JSR $FF5B      INIT EDITOR
801D- 20 53 E4  1180         JSR $E453      MORE HOOKS
8020- 20 BF E3  1190         JSR $E3BF      INIT BASIC
8023- 20 22 E4  1200         JSR $E422      PRINT 'CBM'
8026- A2 FB     1210         LDX ##FB
8028- 9A        1220         TXS            INIT STACK
8029- A5 2B     1230         LDA SOB        MOVE VARIABLES DOWN
802B- 85 2D     1240         STA SOV
802D- A5 2C     1250         LDA SOB+1
802F- 85 2E     1260         STA SOV+1
8031- A9 01     1270         LDA ##01       SET SOB TO THIS ROM
8033- 85 2B     1280         STA SOB
8035- A9 81     1290         LDA ##81
8037- 85 2C     1300         STA SOB+1
8039- 20 60 A6  1310         JSR $A660
803C- A2 05     1320         LDX ##05       SET FOR 5 CHAR
803E- 86 C6     1330         STX KEYCNT
8040- BD 08 80  1340 LOOP    LDA RUN-1,X    STORE 'RUN' IN BUFFER
8043- 9D 76 02  1350         STA KEYBUF-1,X
8046- CA        1360         DEX
8047- D0 F7     1370         BNE LOOP
8049- 58        1380         CLI            ALLOW INTERRUPTS
804A- 4C 74 A4  1390         JMP $A474      TO BASIC WARM
                1400 *---------------------------------
```

FIGURE 8–5
A game cartridge modified for versatile use.



at $8101 instead of $0801, as would be the normal case. The ROM will hold 15 256-byte pages of BASIC program, allowing the first page for the signature and preamble. When a properly linked BASIC program is blown into the ROM along with the preamble, it will start up automatically. As a bonus, all the RAM from $0801 through $7FFF is available for variables, arrays, and strings.

When a program is copied from RAM to the new ROM, you must change the address links to reflect its start at $8100 instead of $0800. We have included in Listing 8–5 a short machine language program that will revise these links in situ and allow the BASIC program to be copied from

LISTING 8–5
Relinker

```
:ASM

                1000 * SUBROUTINE TO RE-LINK FOR BASIC IN ROM
                1010         .OR $C000
                1020         .TA $0800
                1030 *--------------------------------
002B-           1040 SOB     .EQ $2B       START OF BASIC POINTER
0022-           1050 WORK1   .EQ $22       2-BYTE WORK POINTER
0024-           1060 WORK2   .EQ $24       2-BYTE WORK POINTER
8101-           1070 ROMBAS  .EQ $8101     START OF ROM BASIC TEXT
                1080 *--------------------------------
C000- A5 2B     1090 START   LDA SOB       COPY SOB
C002- 85 22     1100         STA WORK1
C004- A5 2C     1110         LDA SOB+1
C006- 85 23     1120         STA WORK1+1
C008- A9 01     1130         LDA #ROMBAS   POINT TO BASIC IN ROM
C00A- 85 24     1140         STA WORK2
C00C- A9 81     1150         LDA /ROMBAS
C00E- 85 25     1160         STA WORK2+1
C010- 18        1170         CLC
C011- A0 01     1180 LOOP    LDY #$01
C013- B1 22     1190         LDA (WORK1),Y
C015- D0 01     1200         BNE SKIP
C017- 60        1210         RTS           EXIT
C018- A0 04     1220 SKIP    LDY #$04
C01A- C8        1230 GETEND  INY
C01B- B1 22     1240         LDA (WORK1),Y
C01D- D0 FB     1250         BNE GETEND
C01F- C8        1260         INY
C020- 98        1270         TYA
C021- 48        1280         PHA
C022- 65 24     1290         ADC WORK2
C024- A0 00     1300         LDY #$00
C026- 91 22     1310         STA (WORK1),Y
C028- 85 24     1320         STA WORK2
C02A- A5 25     1330         LDA WORK2+1
C02C- 69 00     1340         ADC #0
C02E- C8        1350         INY
C02F- 91 22     1360         STA (WORK1),Y
C031- 85 25     1370         STA WORK2+1
C033- 68        1380         PLA
C034- 65 22     1390         ADC WORK1
C036- 85 22     1400         STA WORK1
C038- A9 00     1410         LDA #0
C03A- 65 23     1420         ADC WORK1+1
C03C- 85 23     1430         STA WORK1+1
C03E- 90 D1     1440         BCC LOOP      ALWAYS BRANCH
                1450 *--------------------------------
```

its RAM location to the new ROM. (Be sure to copy the zero at $0800 to $8100, because otherwise the BASIC interpreter will balk!) Once relinked, the program cannot be listed, because the links are not pointing to valid addresses. When the new ROM is plugged in, however, the program will list normally but, of course, cannot be edited.

If you have used the preamble of listing 8–4, the BASIC program in ROM will run at power-up. If RUN/STOP/RESTORE is activated, the control goes to BASIC's warm start. To revert to a normal Commodore 64,

```
POKE 44,8:NEW
```

To return to the auto-start program, emulate power-up by entering

```
SYS64738
```

In the auto-start mode, entering NEW gives an "out of memory" error. If you enter

```
PRINT FRE (0)
```

you will see that the auto-start has added space for variables equal to the space that the BASIC program normally occupies.

If you have difficulty accomplishing these steps for placing your BASIC program in ROM, a programming service is available from The Bit Stop, 5958 South Shenandoah Rd., Mobile, AL 36608, Attn: Don Rindsberg. Finally, if your program will not fit into a 2532 ROM, you may want an MCM68764, which has twice the capacity.

9

INTERFACING A PARALLEL PRINTER

Parallel printers are very easily interfaced to your Commodore 64 because all the active components required are on the Commodore 64's board and all the required leads are on the user port. Parallel printers require at least 10 signal wires plus a ground connection. The information presented over eight of these wires represents the code for the character to be printed. Serial printers, on the other hand, get their information over a single pair of wires (see Chapter 11). Parallel printers are generally cheaper because they can omit serial data decoding circuits. The material in this chapter was inspired by Joel Swank in Part 3 of his series of articles, "The Enhanced Commodore 64" in *Byte Magazine*, April, 1983. The printer he interfaced was the Epson MX-80 with the Graftrax option. We have also interfaced and tested the MX-70 (the predecessor of the MX-80). These and most other parallel printers have the same connector (the Centronics connector). Also, most of these printers have a few unique connections that provide signals to and from the printer, such as an "out-of-paper" signal or a printer-reset signal, that are not necessary in our application.

The Epson printers, and many others, form their characters by the dot matrix method, in which characters are formed by controlling eight or nine impact pins in the print head that are aligned in a vertical row. The Epson printers also have a special mode of operation that allows the control, by the host computer, of every dot on the page. This is common-

ly referred to as the printer's graphics mode. Parallel printers that form their characters in typewriter-like fashion cannot be used for such graphics but, of course, can still be used for printing text.

We will describe the hardware connections required for all parallel printers, the software needed to drive a parallel printer in the text mode, and the slightly more complicated software required for printing the entire Commodore 64 character set in the graphics mode on the Epson printers. This last item is probably of greatest interest to Commodore 64 owners because all of the graphics characters, such as reverse hearts, can be printed. We used this software for printing the BASIC listings in this books.

Connecting the Hardware

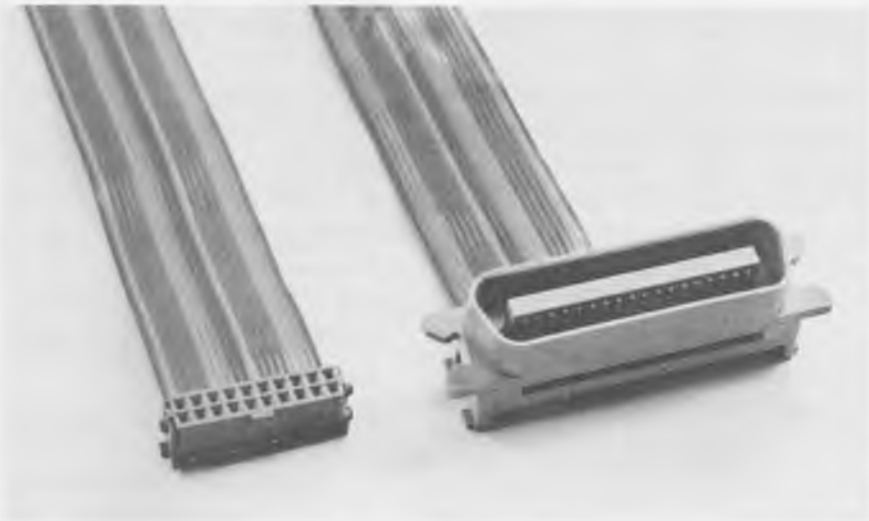
Any method of connecting 10 signal wires and one ground wire from the Commodore 64 user port to the parallel printer will be satisfactory. We used a 4-inch by 4.5-inch Radio Shack board with 0.1-inch on center perforations and a 22/44-position edge connector. We sawed off the unused connector area and soldered a 12/24-pin edge connector to the board so that the board and its connector could be plugged into the user port, as was done to the board described in Chapter 8. An alternative approach is to use the adapter connector shown in Figure 3–7. A 20-pin header plug was glued to the board, and the connections given in Table 9–1 were made. A cable was made with a 20-pin socket on one end and a 36 pin Centronics connector on the other end (see Figure 9–1). Check the entire assembly with an ohmmeter for wiring continuity and shorts before the printer and computer are turned on.

TABLE 9-1: Interconnection Data for Parallel Printer with Commodore 64

MX-70 OR MX-80 PIN NUMBER	*SIGNAL DESCRIPTION*	*COMMODORE 64 USER PORT PIN*	*CIA FUNCTION*
1	STROBE to printer	8	PC2
2	DATA 1	C	PB0
3	DATA 2	D	PB1
4	DATA 3	E	PB2
5	DATA 4	F	PB3
6	DATA 5	H	PB4
7	DATA 6	J	PB5
8	DATA 7	K	PB6
9	DATA 8	L	PB7
10	ACKNLG	B	FLAG 2
19*	SIGNAL GROUND	A	GROUND

*Printer pins 19–30 are all ground connections.

FIGURE 9-1
Photo of printer cable.



Software for Printing in Text Mode

Printing in the text mode is the normal operation of any parallel printer and is several times faster than the graphics mode. The only limitation when this mode is used with the Commodore 64 is that the computer must send only printable characters and control codes that the printer understands.

Listing 9-1 is a very simple machine language program that opens

LISTING 9-1
EPSON ASCII

```
:ASM

                1000 * PRINT ON EPSON IN ASCII MODE (C64)
                1010         .OR $C800
                1020         .TA $0800
                1030 *-----------------------------------
DD01-           1040 PRB     .EQ $DD01     I/O REG B
DD03-           1050 DDRB    .EQ $DD03     DATA DIREC. B
DD0D-           1060 ICR     .EQ $DD0D     INTERRUPT CNTRL.REG.
                1070 *-----------------------------------
F1CA-           1080 C64OUT .EQ $F1CA     C64 CHAR.OUTPUT ROUTINE
                1090 *-----------------------------------
0326-           1100 DISVEC .EQ $0326     VECTOR TO CHAR. OUTPUT ROUTINE
                1110 *-----------------------------------
C800- A9 25     1120 OPEN    LDA #ASCOUT  REROUTE THE CHAR. OUTPUT VECTOR
C802- 8D 26 03  1130         STA DISVEC     TO POINT TO OUR ASCOUT
C805- A9 C8     1140         LDA /ASCOUT
C807- 8D 27 03  1150         STA DISVEC+1
C80A- A9 7F     1160 INITOT LDA #$7F
C80C- 8D 0D DD  1170         STA ICR       NO INTERRUPTS
C80F- A9 FF     1180         LDA #$FF      SET ALL BITS TO OUTPUT
C811- 8D 03 DD  1190         STA DDRB
C814- A9 00     1200         LDA #0        START WITH A NULL
C816- 8D 01 DD  1210         STA PRB
```

```
C819- 60       1220        RTS
               1230 *-----------------------------------
C81A- A9 CA    1240 CLOSE  LDA #C64OUT  REROUTE VECTOR TO NORMAL
C81C- 8D 26 03 1250        STA DISVEC
C81F- A9 F1    1260        LDA /C64OUT
C821- 8D 27 03 1270        STA DISVEC+1
C824- 60       1280        RTS
               1290 *-----------------------------------
C825- 20 CA F1 1300 ASCOUT JSR C64OUT   SEND TO SCREEN
C828- C9 0D    1310        CMP #$0D     CHECK FOR CR
C82A- D0 0E    1320        BNE PUSHA    BRANCH IF NOT
C82C- A9 10    1330        LDA #$10
C82E- 2C 0D DD 1340 WAIT1  BIT ICR
C831- F0 FB    1350        BEQ WAIT1
C833- A9 0A    1360        LDA #$0A     GET LINE FEED
C835- 8D 01 DD 1370        STA PRB      OUTPUT IT
C838- A9 0D    1380        LDA #$0D     GET CR BACK
C83A- 48       1390 PUSHA  PHA          SAVE CHAR.
C83B- A9 10    1400        LDA #$10
C83D- 2C 0D DD 1410 WAIT2  BIT ICR      WAIT FOR FLAG
C840- F0 FB    1420        BEQ WAIT2
C842- 68       1430        PLA
C843- 8D 01 DD 1440        STA PRB      OUTPUT CHAR.
C846- 18       1450        CLC          MAKE C64 HAPPY
C847- 60       1460        RTS
               1470 *-----------------------------------
```

the printer for output, closes out the printer, and outputs a character to the printer. Doing a SYS (see Chapter 2) to the open operation reroutes the Commodore 64's output vector to point to the new output routine and sets up the user port. A SYS to the close routine restores the output to the video screen only. The character output routine ASCOUT puts the character on the screen and then, after waiting for the printer to be ready, outputs to the printers.

The software is assembled for address $C800 (decimal 51200), but it can be reassembled to reside in any convenient location. We selected $C800 as a space normally unused by the operating system. Since the Commodore 64 does not issue line-feeds to start a new line, but just carriage returns, our routine does this little chore. The OPEN routine is called by executing SYS51200 either in keyboard mode or from within a BASIC program. The printer is closed by SYS51226. Note: When you open the printer, the Commodore 64 will hang up if the printer's power is not on or if the cable is not connected. Also, trying to print the Commodore 64's graphics characters may cause weird things to happen to the printout, because these codes are probably undefined for your particular printer.

Software for Printing in Graphics Mode

In order to print all of the Commodore 64 graphics characters, it is necessary to use the printer's graphics mode. The software must be tailored to each printer's demands. The program we describe here will operate the Epson MX-70, Epson RX-80, the Epson MX-80 with Graftrax ROMs, or any Epson-compatible printer. Printing in the graphics mode slows the opera-

tion considerably. We suspect the printer manufacturer did this to keep the print head from overheating.

In the graphics mode, we can control every dot on the printed page, as contrasted with the text mode, where we must rely on the printer's internal character generator to provide dot information for each character. Joel Swank's article showed how to obtain dot information for each character from the computer's character generator. Video character generators are organized by *rows* of dots (witness the horizontal lines on the TV screen), whereas printers require the information as vertical *columns* of dots. We must make the transformation in our print routine. The only trick is to bank-switch just before and just after accessing the character generator.

We have used Mr. Swank's routine, but we have reassembled it to occupy a different address and altered the dot-transformation routine so

LISTING 9-2
EPSON graphic

```
:ASM

                1000 * PRINT ON EPSON IN GRAPHICS MODE - C64
                1010         .OR $C800
                1020         .TA $0800
                1030 *-----------------------------------
00C7-           1040 RVSMOD .EQ $C7       REVERSE MODE FLAG
00D4-           1050 QUOTMO .EQ $D4       QUOTE MODE FLAG
00FB-           1060 CHRPTR .EQ $FB       POINTER TO CHAR.
00FD-           1070 ASAVE  .EQ $FD       SAVE ACC. TEMP.
00FE-           1080 LCOUNT .EQ $FE       COUNT ON CURRENT LINE
                1090 *-----------------------------------
0001-           1100 R6510  .EQ $0001     6510 ON-CHIP PORT
DD01-           1110 PRB    .EQ $DD01     I/O REG B
DD03-           1120 DDRB   .EQ $DD03     DATA DIREC. B
DD0D-           1130 ICR    .EQ $DD0D     INTERRUPT CTRL.REG.
D018-           1140 VICMCR .EQ $D018     VIC MEM.CTRL.REG.
                1150 *-----------------------------------
F1CA-           1160 C64OUT .EQ $F1CA     C64 CHAR.OUTPUT ROUTINE
                1170 *-----------------------------------
0326-           1180 DISVEC .EQ $0326     VECTOR TO CHAR. OUTPUT ROUTINE
                1190 *-----------------------------------
000D-           1200 CR     .EQ $0D       ASCII CARRIAGE RET.
0014-           1210 INSDEL .EQ $14       INSERT/DELETE CHAR.
001B-           1220 ESC    .EQ $1B       ASCII ESCAPE
000A-           1230 LF     .EQ $0A       ASCII LINEFEED
003C-           1240 LINLIM .EQ 60        CHARS./LINE
000C-           1250 LPI6   .EQ 12        CODE FOR 6 LINES/INCH
                1260 *-----------------------------------
C800- A9 38     1270 OPEN   LDA #GRFOUT   REROUTE THE CHAR. OUTPUT VECTOR
C802- 8D 26 03  1280        STA DISVEC     TO POINT TO OUR GRFOUT
C805- A9 C8     1290        LDA /GRFOUT
C807- 8D 27 03  1300        STA DISVEC+1
C80A- A9 7F     1310 INITOT LDA #$7F
C80C- 8D 0D DD  1320        STA ICR       NO INTERRUPTS
C80F- A9 FF     1330        LDA #$FF      SET ALL BITS TO OUTPUT
C811- 8D 03 DD  1340        STA DDRB
C814- A9 00     1350        LDA #$00      START WITH A NULL
C816- 8D 01 DD  1360        STA PRB        TO SET FLAG
C819- A9 1B     1370 INITPR LDA #ESC      ESC-A-# SETS LINES/INCH
C81B- 20 12 C9  1380        JSR PUTCHR     ON EPSON PRINTER
```

```
C81E- A9 41     1390         LDA #'A
C820- 20 12 C9  1400         JSR PUTCHR
C823- A9 0C     1410         LDA #LPI6
C825- 20 12 C9  1420         JSR PUTCHR
C828- 4C 1F C9  1430         JMP INILIN    INITIALIZE FIRST LINE
                1440 *--------------------------------
C82B- A9 CA     1450 CLOSE   LDA #C64OUT   REROUTE VECTOR TO NORMAL
C82D- 8D 26 03  1460         STA DISVEC
C830- A9 F1     1470         LDA /C64OUT
C832- 8D 27 03  1480         STA DISVEC+1
C835- 4C AB C8  1490         JMP FILLIN    FINISH LAST LINE
                1500 *--------------------------------
C838- 85 FD     1520 GRFOUT  STA ASAVE     SAVE A,X,Y
C83A- 48        1530         PHA
C83B- 8A        1540         TXA
C83C- 48        1550         PHA
C83D- 98        1560         TYA
C83E- 48        1570         PHA
C83F- A5 FD     1580         LDA ASAVE
C841- 20 CA F1  1590         JSR C64OUT    CHAR. TO SCREEN
C844- AA        1600         TAX           SET SIGN FLAG
C845- 10 1C     1610         BPL BITOFF    BYPASS IF POSITIVE
C847- 29 7F     1620         AND #$7F      TURN OFF BIT 7
C849- C9 7F     1630         CMP #$7F      CONVERT $7F TO $5E
C84B- D0 02     1640         BNE NOT7F
C84D- A9 5E     1650         LDA #$5E
C84F- C9 20     1660 NOT7F   CMP #$20      CONTROL CHAR?
C851- B0 0C     1670         BCS NOTCTL    BRANCH IF NOT
C853- C9 0D     1680         CMP #CR       CAR.RET?
C855- F0 47     1690         BEQ FINLIN    BRANCH IF SO
C857- A6 D4     1700         LDX QUOTMO    IN QUOTE MODE?
C859- F0 49     1710         BEQ GRFBAK    NO, THEN IGNORE IT
C85B- 09 C0     1720         ORA #$C0      SET BITS 6 & 7
C85D- D0 26     1730         BNE SAVPOK    ALWAYS BRANCH
C85F- 09 40     1740 NOTCTL  ORA #$40      TURN ON BIT 6
C861- D0 1C     1750         BNE CKRVS     ALWAYS BRANCH
C863- C9 0D     1760 BITOFF  CMP #CR       IS IT RETURN?
C865- F0 37     1770         BEQ FINLIN    IF SO, END LINE
C867- C9 20     1780         CMP #$20      CONTROL CHAR?
C869- B0 0A     1790         BCS NOCTL     BRANCH IF NOT
C86B- C9 14     1800         CMP #INSDEL   DELETE?
C86D- F0 35     1810         BEQ GRFBAK    IGNORE IT
C86F- A6 D4     1820         LDX QUOTMO    QUOTE MODE ON?
C871- D0 10     1830         BNE HIBIT     IF SO, PRINT IT
C873- F0 2F     1840         BEQ GRFBAK    IF NOT, IGNORE
C875- C9 60     1850 NOCTL   CMP #$60      OVER $60?
C877- 90 04     1860         BCC LOWER     BRANCH IF NOT
C879- 29 DF     1870         AND #$DF      BIT 5 OFF
C87B- D0 02     1880         BNE CKRVS     BRANCH ALWAYS
C87D- 29 3F     1890 LOWER   AND #$3F      BITS 6/7 OFF
C87F- A6 C7     1900 CKRVS   LDX RVSMOD    REVERSE MODE?
C881- F0 02     1910         BEQ SAVPOK    BRANCH IF NOT
C883- 09 80     1920 HIBIT   ORA #$80      BIT 7 ON
C885- 85 FD     1930 SAVPOK  STA ASAVE     SAVE THE CODE
C887- A5 FE     1940         LDA LCOUNT    CHECK CHAR COUNT
C889- C9 3C     1950         CMP #LINLIM   AT LIMIT?
C88B- D0 08     1960         BNE NOTFUL    BRANCH IF NOT
C88D- A9 0A     1970         LDA #LF       LINEFEED TO PRINTER
C88F- 20 12 C9  1980         JSR PUTCHR
C892- 20 1F C9  1990         JSR INILIN    START NEW LINE
C895- A5 FD     2010 NOTFUL  LDA ASAVE     GET CHAR. BACK
C897- 20 BF C8  2020         JSR SEND      SEND TO PRINTER
C89A- E6 FE     2030         INC LCOUNT    BUMP CHAR.COUNT
C89C- D0 06     2040         BNE GRFBAK    BRANCH ALWAYS
C89E- 20 AB C8  2050 FINLIN  JSR FILLIN    FILL LINE WITH BLANKS
C8A1- 20 1F C9  2060         JSR INILIN    START NEW LINE
C8A4- 68        2070 GRFBAK  PLA           RESTORE Y,X,A
C8A5- A8        2080         TAY
```

```
C8A6- 68       2090         PLA
C8A7- AA       2100         TAX
C8A8- 68       2110         PLA
C8A9- 18       2120         CLC            MAKE C64 HAPPY
C8AA- 60       2130         RTS            AND RETURN
               2140 *-----------------------------------
C8AB- A6 FE    2150 FILLIN  LDX LCOUNT     GET CHAR COUNT
C8AD- E0 3C    2160         CPX #LINLIM    AT LIMIT?
C8AF- B0 09    2170         BCS NXTLIN     IF SO, DO LINEFEED
C8B1- A9 20    2180         LDA #$20       SEND BLANK
C8B3- 20 BF C8 2190         JSR SEND
C8B6- E6 FE    2200         INC LCOUNT
C8B8- D0 F1    2210         BNE FILLIN     BRANCH ALWAYS
C8BA- A9 0A    2220 NXTLIN  LDA #LF        SEND LINEFEED
C8BC- 4C 12 C9 2230         JMP PUTCHR
               2240 *-----------------------------------
C8BF- 85 FB    2250 SEND    STA CHRPTR     SAVE POKE CODE
C8C1- A9 00    2260         LDA #0
C8C3- 85 FC    2270         STA CHRPTR+1
C8C5- A0 02    2280         LDY #2         MULTIPLY BY 8 TO GET
C8C7- 18       2290 MULT8   CLC             OFFSET INTO CURRENT CHAR SET
C8C8- 26 FB    2300         ROL CHRPTR
C8CA- 26 FC    2310         ROL CHRPTR+1
C8CC- 88       2320         DEY
C8CD- 10 F8    2330         BPL MULT8      LOOP 3 TIMES
C8CF- A2 D8    2340         LDX #$D8       ASSUME ALT.CHAR.SET
C8D1- AD 18 D0 2350         LDA VICMCR     READ VIC CHIP
C8D4- 29 02    2360         AND #$02       IS ALT.SET ?
C8D6- D0 02    2370         BNE ADDEM      BRANCH IF ALT.
C8D8- A2 D0    2380         LDX #$D0       USE PRIMARY SET
C8DA- 8A       2390 ADDEM   TXA            NOW CALCULATE ADDRESS OF
C8DB- 18       2400         CLC             CHARACTER SET
C8DC- 65 FC    2410         ADC CHRPTR+1
C8DE- 85 FC    2420         STA CHRPTR+1
C8E0- A9 01    2440         LDA #1         THIS ROUTINE CONVERTS DOT ROWS TO
C8E2- 48       2450 BYTLP   PHA             DOT COLUMNS, SAVES ON STACK,
C8E3- AA       2460         TAX             AND OUTPUTS THE COLUMNS TO PRINTER
C8E4- A0 07    2470         LDY #7
C8E6- 78       2480 BITLP   SEI
C8E7- A5 01    2490         LDA R6510      ACCESS CHAR.GEN.
C8E9- 29 FB    2500         AND #$FB
C8EB- 85 01    2510         STA R6510
C8ED- B1 FB    2520         LDA (CHRPTR),Y
C8EF- 48       2530         PHA
C8F0- A5 01    2540         LDA R6510      ACCESS I/O
C8F2- 09 04    2550         ORA #$04
C8F4- 85 01    2560         STA R6510
C8F6- 68       2570         PLA
C8F7- 58       2580         CLI
C8F8- 0A       2590 SHFTLP  ASL
C8F9- CA       2600         DEX
C8FA- D0 FC    2610         BNE SHFTLP
C8FC- 66 FD    2620         ROR ASAVE
C8FE- 68       2630         PLA
C8FF- 48       2640         PHA
C900- AA       2650         TAX
C901- 88       2660         DEY
C902- 10 E2    2670         BPL BITLP
C904- A5 FD    2680         LDA ASAVE
C906- 20 12 C9 2690         JSR PUTCHR
C909- 68       2700         PLA
C90A- 18       2710         CLC
C90B- 69 01    2720         ADC #1
C90D- C9 09    2730         CMP #9
C90F- D0 D1    2740         BNE BYTLP
C911- 60       2750         RTS
               2760 *-----------------------------------
C912- 48       2770 PUTCHR  PHA            SAVE CHAR.
```

```
C913- A9 10    2780         LDA #$10      TEST INTERRUPT BIT
C915- 2C 0D DD 2790 WAITO   BIT ICR
C918- F0 FB    2800         BEQ WAITO     WAIT UNTIL PRINTER READY
C91A- 68       2810         PLA           GET CHAR.
C91B- 8D 01 DD 2820         STA PRB       OUTPUT TO CIA
C91E- 60       2830         RTS
               2840 *--------------------------------
C91F- A9 00    2850 INILIN  LDA #0         CLEAR LINE COUNT
C921- 85 FE    2860         STA LCOUNT
C923- A9 1B    2870         LDA #ESC      SET PRINTER FOR 480 DOT COLUMNS
C925- 20 12 C9 2880         JSR PUTCHR
C928- A9 4B    2890         LDA #'K
C92A- 20 12 C9 2900         JSR PUTCHR
C92D- A9 E0    2910         LDA #480
C92F- 20 12 C9 2920         JSR PUTCHR
C932- A9 01    2930         LDA /480
C934- 4C 12 C9 2940         JMP PUTCHR
               2950 *--------------------------------
```

that it uses only the microprocessor stack for temporary storage of the eight columns of dots (see Listing 9–2). As a result, our routine can be placed in ROM. For test purposes, the routine can be reassembled to occupy RAM space, but we find it very convenient to have it resident in ROM at all times. Both program listing and printing from within a program work well. Sixty characters can be printed on each line, using standard 8½-inch-wide paper.

10

THE GAME PORT

The two game ports, located on the right side of the Commodore 64, provide yet another means by which the computer can communicate with the outside world. Commodore has already used this capability for its game paddles, joysticks, and light pen. In this chapter, we will explore these and some additional capabilities of the game ports.

The Hardware

Figure 10–1 shows the pin assignments of the game port connectors. The game port requires a 9-pin, female, type "D," subminiature connector (available from Radio Shack). The game port's functions actually come from several different devices in the Commodore 64. The joystick leads come from a 6526 CIA chip and provide 10 TTL level-sensitive input lines. The game paddle inputs both come from the 6581 SID chip. The game paddle provides 4 pseudo–analog-to-digital converters. Finally, the light pen (pin 6 port 1) provides the ability to sense a light pen's position on the monitor's screen.

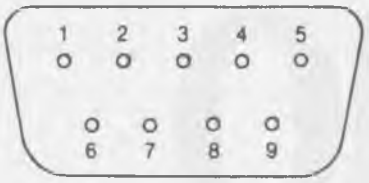
Joystick Functions

The joystick lines come from ports A and B of CIA#1. Pins 1 through 4 of game port 1 connect to PA0 through PA3, respectively. Similarly,

FIGURE 10–1
Pin assignments for the C64's game port. Courtesy of Commodore Business Machines.

Control Port 1

Pin	Type	Note
1	JOYA0	
2	JOYA1	
3	JOYA2	
4	JOYA3	
5	POT AY	
6	BUTTON A/LP	
7	+5V	MAX. 50mA
8	GND	
9	POT AX	



Control Port 2

Pin	Type	Note
1	JOYB0	
2	JOYB1	
3	JOYB2	
4	JOYB3	
5	POT BY	
6	BUTTON B	
7	+5V	MAX. 50mA
8	GND	
9	POT BX	

pins 1 through 4 of game port 2 connect to PB0 through PB3, respectively. Pin 6, the fire button pin, connects to PA4 for port 1 and PB4 for port 2. Since the 6526 CIA chip has internal pull-up resistors, these pins will read as 1s if they are unconnected. Grounding any pin will cause it to return a logical 0 for that bit. Commodore's joystick consists of 5 normally open switches to ground. Deflecting the joystick or pushing the fire button simply closes the appropriate switch to ground and pulls that line low. The data direction registers (see chapter 3) for CIA#1 are automatically set by BASIC to be in the input mode. Thus, all that is needed to read the status of these lines is a PEEK to either register A for port 1 or register B for port 2. These registers are located at $DC00 (56320) for port A and $DC01 (56321) for port B.

Although you might think that you could use the joystick lines as outputs as well as inputs, the latter task is not easy. The joystick lines serve double duty in the Commodore 64. They connect to the keyboard matrix and are used every 60th of a second to scan the keyboard to see if a key has been depressed. Thus any bit pattern you might try to output over these lines would surely be corrupted 60 times per second. To get

around that problem, you could have a machine language program inhibit interrupts with a SEI instruction and stop the keyboard scan. Under those conditions, you could store an $FF in the data direction register at $DC02 (port A) or $DC03 (port B) to define the joystick lines as outputs. A return to BASIC, however, will immediately clear the interrupt mask.

The keyboard scan doesn't cause any problems when the joystick lines are in the input mode since BASIC will never try to read these lines during an interrupt. Obviously, however, various signals on the joystick lines can interfere with the operation of your keyboard. For example, if pin 4 of port 1 is grounded, it is equivalent to typing a 2 on the keyboard. Thus, try to write your programs so that signals will be sensed at a time when the keyboard is not active.

Demonstrating the Operation of a Switch Closure

Connect a jumper wire to pin 8 (ground) of game port 1. Be careful not to short pin 8 to pin 7, which is connected to +5 volts. Such a short will blow the fuse in your computer! Next, enter the following program.

```
10 X = PEEK(56320)
20 X = 15 - (X AND 15)
30 PRINT X
40 GOTO 10
```

When you RUN the program, the screen will fill with 0s. Now touch the free end of the wire to pin 1 of game port 1. Notice that the 0s are replaced with 1s. Touching the wire to pin 2 will result in 2s and touching the wire to pin 3 will produce 4s. Remember, the 10K resistors in the CIA chip pull these lines to a logic 1. Shorting the line to ground causes it to change to a logic 0. Thus, to detect any switch closure, connect one side of the switch to the line and the other side to ground. One kilohm or less resistance to ground is required to pull the line to a logic low.

Optical Detectors

A useful application is to have the Commodore 64 sense a changing light level. This has many applications, such as in intrusion alarms, event counting on production lines, automatic dusk to dawn lighting, and so on. Two types of photocell are popular today, the cadmium sulfide cell (CdS) and the phototransistor. The CdS cell is popular for low-light applications, while the phototransistor has rapid response characteristics. Figures 10–2 and 10–3 show each of these interfaced as light sensitive triggers. In both circuits, shining light on the cell will cause a logic 0 state.

FIGURE 10–2
A cadmium-sulfide light detector interfaced to a joystick input.

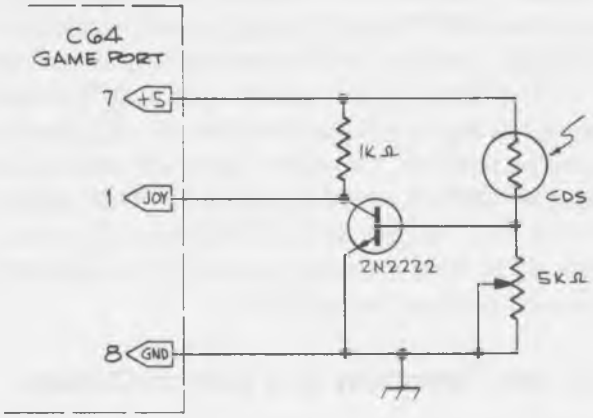
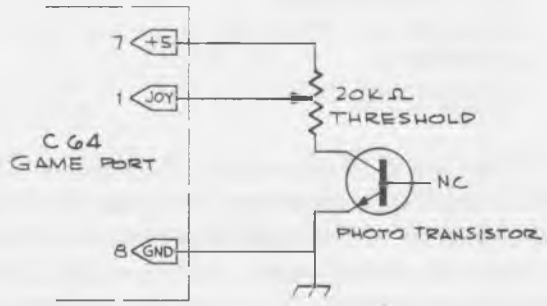


FIGURE 10–3
A phototransistor interfaced to a joystick input. Radio Shack #276–130 recommended.



Infrared Detectors

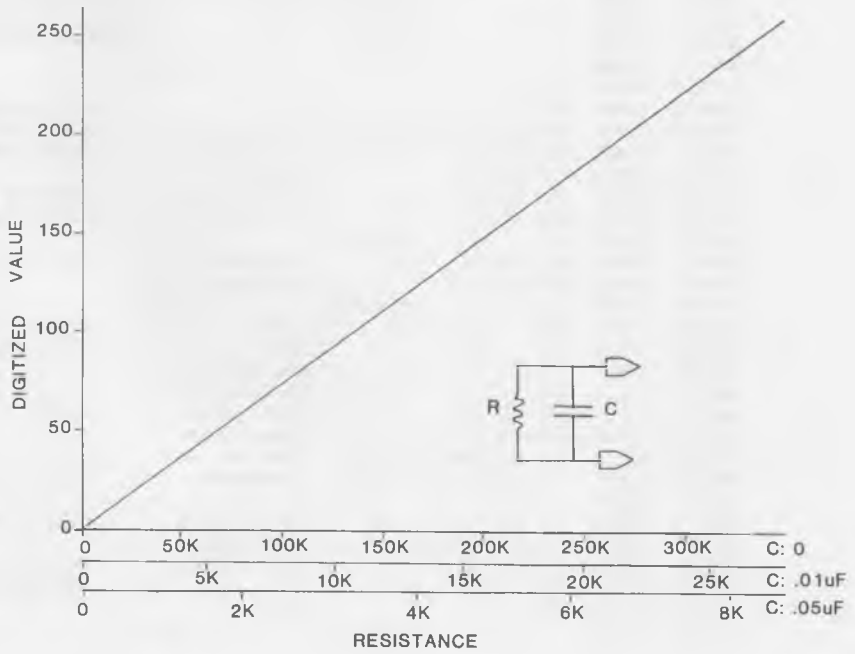
One problem with optical detectors in many applications, such as industrial counting operations, is that ambient light interferes with their operation. A clever way to avoid that problem is to use an infrared emitter and detector. An invisible infrared beam travels between the emitter and the sensor. Any object interrupting that path will be detected, yet the sensitivity will not be affected by any change in ambient light levels. Radio Shack currently offers a matched pair of devices under the part number 276–142. Figure 10–4 shows a pair of these interfaced to the Commodore 64 via a joystick line. The emitter, an infrared-emitting diode, has a lens on it and casts a narrow beam that is highly focused. For proper operation the beam must be carefully aligned to shine on the detector. We found the system worked fine with light paths as long as a foot or more. The out-of-

paper detector in our line printer uses an infrared emitter-detector system. Useful applications abound!

The Game-Paddles

The Commodore 64 was conceived in the middle of the video-game age. As a result, an important consideration in the design was to include a provision for game-paddles. These are analog-to-digital converters that allow the machine to sense the position of a potentiometer. Four of these are implemented in the Commodore 64 and are on pins 5 and 9 of each of the game ports. Actually these are pseudo–analog-to-digital converters. They actually measure the resistance between the +5 volt supply and the input line. The game paddle inputs are not capable of sensing a voltage directly. The paddle resistance is read by two converters in the 6581 sound interface device (SID). Four paddles can be monitored by only two converters by means of a multiplex circuit. A 4056 FET switch connects the selected port to the x and y inputs of the SID chip. The multiplexer is, in turn, controlled by port A of CIA#1. When bit 7 of port A is high ($80), game port 2 is selected. Game port 1 is selected when bit 6 is high ($40).

FIGURE 10–4
An infrared emitter and detector interfaced to a joystick input. The infrared detector is unaffected by ambient light. Radio Shack #276–142 infrared emitter-detector pair is recommended.



When both bits 6 and 7 are high ($C0) both game ports are selected at once and the parallel equivalent of the resistances on both ports will be seen by the computer. Obviously, this latter condition should be avoided.

Reading the Paddles

Although the multiplexer circuit allowed Commodore to add a second game port to the 64, an improvement over the VIC 20, the addition was not without cost. The converters are automatic and normally would only need to be read with a PEEK at the appropriate registers in the SID chip—$D419 and $D41A. Unfortunately, there is a slight problem that prevents such simple operation. The bits 6 and 7 of port A are also used to scan the keyboard matrix. Thus, 60 times a second the multiplexer will get spurious

LISTING 10-1
Paddle read program

```
LINE# LOC   CODE        LINE

00001  0000             ;*********************************
00002  0000             ; FOUR PADDLE READ ROUTINE
00003  0000             ; PADDLE REGS AT 155-159 (XA,XB,YA,YB)
00004  0000             ;*********************************
00005  0000             PORTA=$DC00
00006  0000             CIDDRA=$DC02
00007  0000             SID=$D400
00008  0000             *=$C000
00009  C000  4C 08 C0          JMP PDLRD
00010  C003             PDLX    *=*+2
00011  C005             PDLY    *=*+2
00012  C007             TEMP    *=*+1
00013  C008             PDLRD
00014  C008  A2 01              LDX #1
00015  C00A  78                SEI                 ;NO KEYBOARD SCANS
00016  C00B  AD 02 DC           LDA CIDDRA          ;GET DATA DIR REG
00017  C00E  8D 07 C0           STA TEMP            ;SAVE FOR A WHILE
00018  C011  A9 C0              LDA #$C0
00019  C013  8D 02 DC           STA CIDDRA          ;ENABLE BITS 6&7
00020  C016  A9 80              LDA #$80            ;MUX CODE FOR PRTA
00021  C018             PDLRD1
00022  C018  8D 00 DC           STA PORTA           ;SET MUX
00023  C01B  A0 80              LDY #$80
00024  C01D             PDLRD2
00025  C01D  EA                NOP                 ;DELAY LONG ENOUGH
00026  C01E  88                DEY                 ;FOR SID TO READ
00027  C01F  10 FC              BPL PDLRD2
00028  C021  AD 19 D4           LDA SID+25
00029  C024  9D 03 C0           STA PDLX,X          ; GET X
00030  C027  AD 1A D4           LDA SID+26
00031  C02A  9D 05 C0           STA PDLY,X          ; GET Y
00032  C02D  A9 40              LDA #$40            ;MUX CODE FOR PRTB
00033  C02F  CA                DEX
00034  C030  10 E6              BPL PDLRD1          ;BOTH PORTS READ?
00035  C032  AD 07 C0           LDA TEMP
00036  C035  8D 02 DC           STA CIDDRA          ;RESTORE DATA DIR
00037  C038  58                CLI                 ;RE-ENABLE KBD
00038  C039  60                RTS
00039  C03A             .END
```

signals. If you happen to PEEK the registers right after a keyboard scan, you will not get an accurate number. To get around that problem, Commodore recommends that you read the paddles only with a machine language subroutine. The *Commodore 64 Programmers Reference Guide* (Howard Sams, Inc., 1983) presents a compact machine language routine. We have modified that program slightly to produce the program in Listing 10–1. The essence of that program is as follows: line 15 sets the interrupt mask so that keyboard scans will be inhibited during the read routine. In line 22, the multiplexer is set to the appropriate port. Once the multiplexer has been set, the program must delay long enough for the resistor to charge the capacitor. The delay occurs in lines 23 to 27. After the delay is over, the registers are read by lines 28 and 30. These are stored in the free locations at the start of the program, $C003–$C006. After the ports are read, the interrupt mask is cleared in line 37 and the control returns to BASIC via the RTS in line 60.

The BASIC program in Listing 10–2 shows how to use the machine language subroutine. The machine language subroutine is loaded by reading the program in the data statements and POKEing the codes into place. First, the program is executed with a SYS to 49152 ($C000). Then, the 4 paddle storage locations 49155–49159 can be read with PEEK statements. To repeat the reading, the SYS must be executed again.

The resistance is determined by measuring the RC time constant of the external resistor and an internal .001-μf (microfared) capacitor. Figure

LISTING 10–2
Paddle (BASIC)

```
READY.

0 DATA 76 , 8 , 192 , 255 , 255
5 DATA 255 , 255 , 255 , 162 , 1
10 DATA 120 , 173 , 2 , 220 , 141
15 DATA 7 , 192 , 169 , 192 , 141
20 DATA 2 , 220 , 169 , 128 , 141
25 DATA 0 , 220 , 160 , 128 , 234
30 DATA 136 , 16 , 252 , 173 , 25
35 DATA 212 , 157 , 3 , 192 , 173
40 DATA 26 , 212 , 157 , 5 , 192
45 DATA 169 , 64 , 202 , 16 , 230
50 DATA 173 , 7 , 192 , 141 , 2
55 DATA 220 , 88 , 96
100 A=49152:FOR I=0 TO 57
110 READX:POKE A+I,X
120 NEXT I:REM  PROGRAM HAS BEEN POKED IN
125 SYS A: REM READ PADDLES
130 P1=PEEK(A+3):REM PADDLE 1
140 P2=PEEK(A+4):REM PADDLE 2
150 P3=PEEK(A+5):REM PADDLE 3
160 P4=PEEK(A+6):REM PADDLE 4
170 PRINT P1,P2,P3,P4
180 GOTO 125

READY.
```

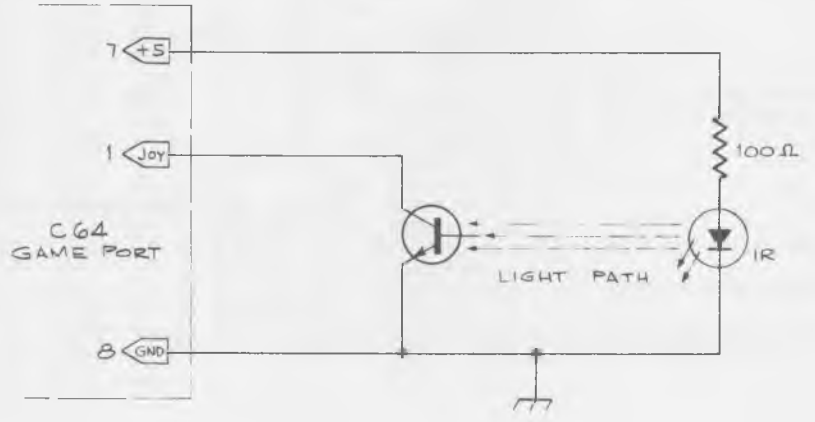
10–5 shows a graph of the value computed by the SID chip on the vertical axis and the resistance across the input on the horizontal axis. Note that the scales can be changed by increasing the capacitance with an external capacitor across the leads. Thus, about any resistance range you desire can be achieved by simply adding the proper amount of external capacitance. When a large capacitor is used, the resistance should not be too small; otherwise, the capacitor cannot be discharged between cycles. About 200 ohms was as low as we could use with the .05 μF capacitor. If the resistance is too low, the oscillator simply locks up.

Obviously, any variable resistance such as a potentiometer can be coupled to the game paddle to become an analog sensor. Place two properly spaced electrodes in a cup of fluid and you have a conductance meter. Also, the Commodore 64 makes a fairly accurate ohmmeter for calibrating any unmarked resistors you may have laying around.

Photosensors

Just as the CdS cell can be used to control the joystick input, it is ideally suited to the game paddle input as well. The added advantage here is that the CdS cell, placed on the game paddle lines, makes the Commodore 64 an accurate light meter. The CdS cell is connected between +5V and the game paddle input. If you are interested in low light sensitivity, as in an enlarger meter, then you probably won't need any parallel capacitance. On the other hand, if you want sensitivity right up to bright room light, you should add a .05 μF capacitor in parallel with the photocell.

FIGURE 10–5
A graph that relates the number returned by the C64 as a function of the resistance across the game paddle's input. The 3 horizontal scales result from three different values of capacitance in parallel with the resistance.

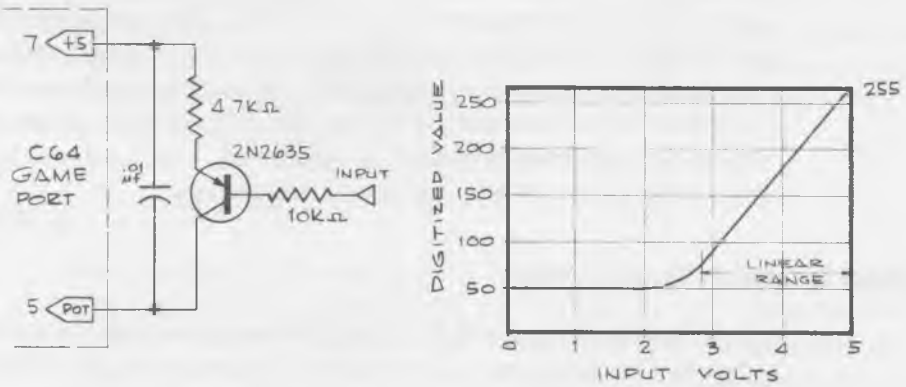


Connect the CdS cell as described above, add a relay on one of the user port lines (which has been configured as an output) and you have all of the makings of an automatic enlarger timer for your darkroom. Don't forget that the time-of-day function (TI) in the Commodore 64 makes critical timing tasks a snap.

A Poor Man's A to D Converter

In the beginning of this chapter, we called the game paddle input a pseudo-analog-to-digital converter. By substituting the circuit shown in Figure 10-6 for the resistance across the game paddle input, the Commodore 64 can be made to sense analog voltages in the range of 2 to 5 volts. The PNP transisitor is operated as an emitter follower. In this mode, the collector becomes a current source, where the current equals (5—Ein)/47000. The charging rate of the timing capacitor will be proportional to the current, and thus the digitalized value will vary with the input voltage, Ein. The graph in Figure 10-6 indicates that, in the range between 2.5 and 5 volts, the response is indeed a linear function of Ein. By level shifting and amplifying the input signal with an op-amp, almost any range of input voltages can be transposed into the 2.5 volt range. For more demanding applications, we recommend that you use the A to D converter described in Chapter 6.

FIGURE 10-6
A "poor man's" analog-to-digital converter. The circuit is very linear in the range of input voltages between 2.5 and 5 volts.



11

USING THE RS-232 PORT ON THE COMMODORE 64

An important mode of communication between the outside world and a computer is through an RS-232 channel. More computer peripherals are available with this mode of communication than any other. Fortunately, the engineers at Commodore had the foresight to include RS-232 capabilities in the Commodore 64 design, which gives it a tremendous advantage over its predecessor, the PET 2001. This capability allows the Commodore 64 to support serial printers, modems, remote terminals, x-y plotters, ROM programmers, digitizers, and an almost endless list of other peripheral devices currently on the market. This chapter explains how the RS-232 is implemented in the Commodore 64, what hardware you must provide to make it work, and how it can be used.

What Is Meant by RS-232?

RS-232 refers to a protocol set up for information exchange between digital devices. The digital data is sent in serial fashion, that is, one bit of information after another over a single wire. Since this data is usually organized as a byte (8 bits), careful timing must be observed to be sure that the byte can be reconstructed again by the receiving device. Furthermore, the protocol is such that the communication can be asynchronous, that is, a byte can be sent any time the transmitting device has data to send. The

transmitter does not have to synchronize the transmission with some event in the receiver. That, of course, means the receiver must be ready to accept data at all times.

Let's look at how a byte of data is actually transmitted. A transmit cycle is broken down into small, equal time periods. Let's assume that we have 8 data bits, 1 start bit, and 2 stop bits. (The number of bits in the format varies from system to system, as you will see later, but the 1, 8, and 2 arrangement is the most common). At the beginning of the cycle, the voltage on the signal line indicates a logic 1 (referred to as a "mark" in RS-232 jargon). The beginning of the interchange is indicated by the line assuming the logic 0 state (referred to as a "space") for one time period. This is the start bit. The start bit signals the receiver that data is about to be sent, and the start bit must always be a space. Next come the 8 data bits in sequence for the next eight time periods. The least significant bit is first, and the most significant bit is transmitted last in this scheme. Finally, the line is held in the mark state for two time periods to signify the end of the sequence. These are the 2 stop bits. After the stop bits, the mark state may continue if the line is to be idle for a while, or it may immediately change to a space, which is the start bit for the next word. The receiver must be ready to receive another word right after the stop bits.

Baud Rate

The duration of the time periods determines the rate at that the data can be transmitted over the line. We refer to this parameter as the baud rate. The baud rate is simply the number of time periods per second. For example, a baud rate of 110 has 110 time periods each second. Since each byte of information requires 11 time periods—1 start bit, 8 data bits, and 2 stop bits—a maximum of 10 bytes of data can be transmitted over a 110-baud line each second.

Level Shifting

Today almost all computers use the convention of +5 volts representing a logic 1 and 0 volts representing a logic 0. That was not always the case, however. The early computers used a wide variety of voltage representations. In an attempt to standardize computer peripherals, two interface conventions emerged in the 1960s as most popular for serial devices and are still in use today. These are the 20-ma current loop and the RS-232. In the former, a mark is represented by a 20-ma current flow and a space by open circuit. In the latter a mark is represented by -12VDC, and a space is +12VDC. When Commodore says that the Commodore 64 sup-

ports RS-232 devices, that is not exactly true. The Commodore 64 can indeed transmit and receive serial data in the RS-232 format, but the signal lines are arranged so that a mark is +5 volts and a space is 0 volts. *The user must provide a circuit that attaches to the Commodore 64's user port to convert those TTL signals to either the RS-232 or 20-ma current loop levels.* Later on in this chapter, we will show you circuits that will accomplish this level shifting.

Handshaking

Although an RS-232 connection may only involve two wires, a signal line, and a ground line, there are provisions for more lines, called handshaking lines. Often it is not possible to make a receiver that will be ready for data all the time. For this circumstance a CTS (clear to send) line is provided over which the receiver can signal the transmitter that it is not ready for data. For example, most printers today have the ability to store several hundred words of data before they are actually printed. Let's say that the printer can actually print about 30 characters per second. It could, therefore, receive and print data at 300 baud continuously. Over two seconds would be required, however, to send an 80 column line of text. This would tie up the computer for two seconds every time a single line was to be printed. Such delays can be very annoying. If, however, the baud rate were increased so that the buffer in the printer were quickly filled, the computer could dump its line of text and then be free to do other things. If the printing task was a long one, the printer's buffer would quickly fill, and then the printer would have to signal the computer through the CTS line that it must now wait until a character is printed before another one is to be sent. Thus, at least short periodic records could be transmitted at high speed, whereas the longer records would still be slow due to the printing speed. Other signal lines are defined in the RS-232 protocol and can be used to ensure an orderly flow of information. These signals and their meanings are shown in Figure 11-1. Most of these handshake lines are implemented in the Commodore 64.

Parity

Another important aspect of the RS-232 protocol is the parity bit. One of the goals of computer design is to have an error-free flow of information within the system. One major source of error, especially in the early systems, was the RS-232 connection. These often involved the use of unreliable mechanical relays to generate the marks and spaces. Also, slight differences in the baud rates between the transmitter and receiver led to

FIGURE 11–1
Pin assignments for the RS–232 line on the C64's user port. Courtesy of Commodore Business Machines.

(6526 DEVICE #2 Loc. $DD00—$DD0F)						
PIN ID	6526 ID	DESCRIPTION	EIA	ABV	IN/ OUT	MODES
C	PB0	RECEIVED DATA	(BB)	S_{in}	IN	1 2
D	PB1	REQUEST TO SEND	(CA)	RTS	OUT	1*2
E	PB2	DATA TERMINAL READY	(CD)	DTR	OUT	1*2
F	PB3	RING INDICATOR	(CE)	RI	IN	3
H	PB4	RECEIVED LINE SIGNAL	(CF)	DCD	IN	2
J	PB5	UNASSIGNED	( )	XXX	IN	3
K	PB6	CLEAR TO SEND	(CB)	CTS	IN	2
L	PB7	DATA SET READY	(CC)	DSR	IN	2
B	FLAG2	RECEIVED DATA	(BB)	S_{in}	IN	1 2
M	PA2	TRANSMITTED DATA	(BA)	S_{out}	OUT	1 2
A	GND	PROTECTIVE GROUND	(AA)	GND		1 2
N	GND	SIGNAL GROUND	(AB)	GND		1 2 3

MODES:
1) 3-LINE INTERFACE (S_{in},S_{out},GND)
2) X-LINE INTERFACE
3) USER AVAILABLE ONLY (Unused/unimplemented in code.)
* These lines are held high during 3-LINE mode.

corruption of data due to framing errors. To detect lost data, the computer engineers used the parity system. A running sum of the first 7 data bits was taken and the eighth bit of the transmission, the parity bit, indicated the result of the sum. If the sum of the 7 bits resulted in an even number, the parity bit would be a mark. If the result was odd, the parity bit would be a space. The receiver would add up the first 7 bits and compare its parity count to the bit 8 it had received. If they agreed, then it was unlikely that any bits were lost.

Parity systems again differ from machine to machine. For example, the system above describes an *even parity* system. In an *odd parity* system, the parity bit is a mark when the sum of the bits is odd. In a *mark parity* system, bit 8 is always a mark.

The electronics for serial transmission and reception are much more sophisticated today, and data loss over an RS–232 link has virtually been eliminated. Thus, most modern systems do not use a parity error check and use either mark parity or *no parity*. In a no parity system, bit 8 becomes another data bit so that a full byte of data is sent on each transmission. Remember that the ASCII code that the Commodore 64 uses to send alphanumeric symbols (see Chapter 1) only requires 7 bits, and a full 8-bit byte is not usually required for most RS–232 applications.

As you can see from the description above, there are many options to consider when trying to interface an RS–232 device to the Commodore 64. Let's say that you have a printing ASCII terminal that you want to interface to the Commodore 64. This will give you both a hard-copy capability and a remote keyboard. First you must determine at what baud rate the terminal operates. This can be determined by either: (1) consulting the owner's manual, (2) examination of the markings on the baud rate switches or jumpers on the terminal's logic boards, or (3) consulting a knowledgeable friend. The baud rate information is usually easy to obtain. Next, you must find out if the terminal is RS–232 or 20-ma current loop. All ASR33 teletypes and most of Digital Equipment's devices are 20 ma. Most other terminals are RS–232. If the terminal has the standard RS–232 connector on it (see Figure 11–2), then you can almost be sure that it is RS–232. Finally, you must determine what parity system it uses. This can be a little more elusive because most terminals can be user-configured for any of the possibilities. An owner's manual is the best source of this information. If no manual is available, you may have to experiment to determine the parity protocol.

An RS–232 Level Shifting Interface

If your device uses the RS–232 signal levels, then you have one of two choices: (1) you can purchase Commodore's RS–232 interface cable, or (2) you can build the one shown in Figure 11–3. The interface plugs into the user port. Since the RS–232 protocol uses +12 volts as a space and −12 volts as a mark, the interface needs both a positive and a negative supply on the board. An MC1488 line driver converts the TTL output on pin M of the user port to the RS–232 levels. RS–232 inputs are in turn sensed by the MC1489 receiver chip, which converts them to the TTL levels required for the user port inputs on pins K, B, and C.

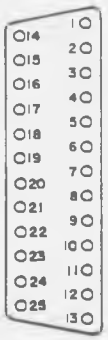
20-ma Current Loop

To our knowledge, there are no current loop interfaces commercially available for the Commodore 64. Since ASR33 teletypes are readily available and make fine low-cost printers, we will describe a current loop interface for the Commodore 64. 20-ma current loop interfaces are either "active" or "passive." The active side of the loop supplies the voltage source for current flow whereas the passive side does not. Thus, there must always be an active side and a passive side for any current loop connection. The sending side acts as a switch, either open or closed, while

FIGURE 11-2
Pin assignments for the RS-232 "D" connector. Pins 2, 3, and 7 are used for a three-wire connection (no handshake). The remaining pins are handshake lines and can be used with the C64 as explained in the text. Transmitted data (pin 2) is, by convention, from the terminal to the computer.

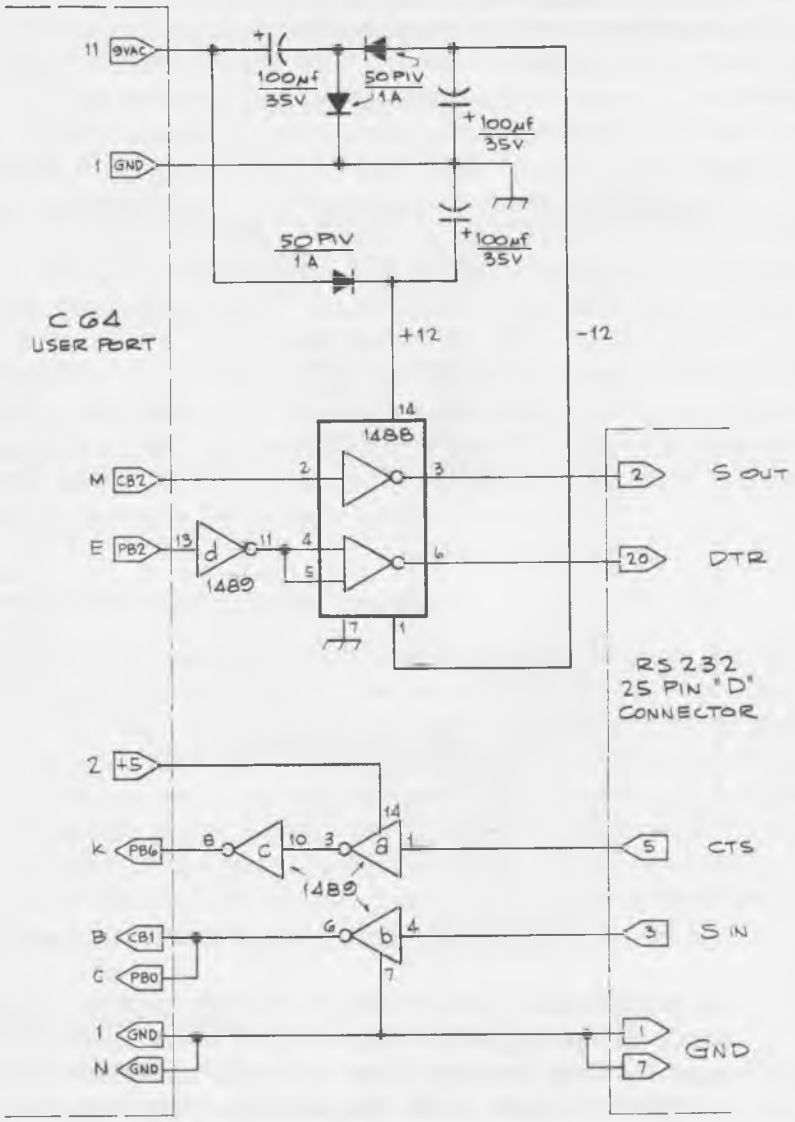
RS-232 CONNECTIONS

RS-232 CONNECTOR PIN	FUNCTION	USER PORT PIN
1	PROTECTIVE GROUND	A & N
2	TRANSMITTED DATA	M
3	RECEIVED DATA	B & C
4	REQUEST TO SEND	D
5	CLEAR TO SEND	K
6	DATA SET READY	L
7	SIGNAL GROUND	A & N
8	CARRIER DETECT	H
9	(NOT USED)	
10	"	
11	"	
12	"	
13	"	
14	"	
15	"	
16	"	
17	"	
18	"	
19	"	
20	DATA TERMINAL READY	E
21	(NOT USED)	
22	"	
23	"	
24	"	
25	"	



the receiver senses current flow. In the early machines, the switch was usually the opening and closing of a relay's contacts, while the sensor was usually the coil of a relay. Today we usually use opto-isolators as shown in Figures 11-4 and 11-5. Most terminals are passive and thus require an active interface on the Commodore 64. On the other hand, most computers and commercial modems are active and require a passive interface. A quick way to determine your requirement is to put a voltmeter across the two *input* lines to the terminal. If you find 0 volts, then the device is passive. If you see 5-12 volts on the line, then the device is active. ASR33 teletypes are passive.

FIGURE 11-3
Schematic of an RS-232 interface. This configuration is for the C64 as a remote terminal. If the C64 is the computer rather than the terminal, then interchange pins 2 and 3 and pins 20 and 5.



Software Considerations

Assuming that you have a functional interface, it is now time to program it. You must first tell the Commodore 64 the baud rate, word length, number of stop bits, and parity convention so that they match the requirements of the terminal. This is all done at the time the user opens the data chan-

FIGURE 11–4
A 20-ma "active" interface.

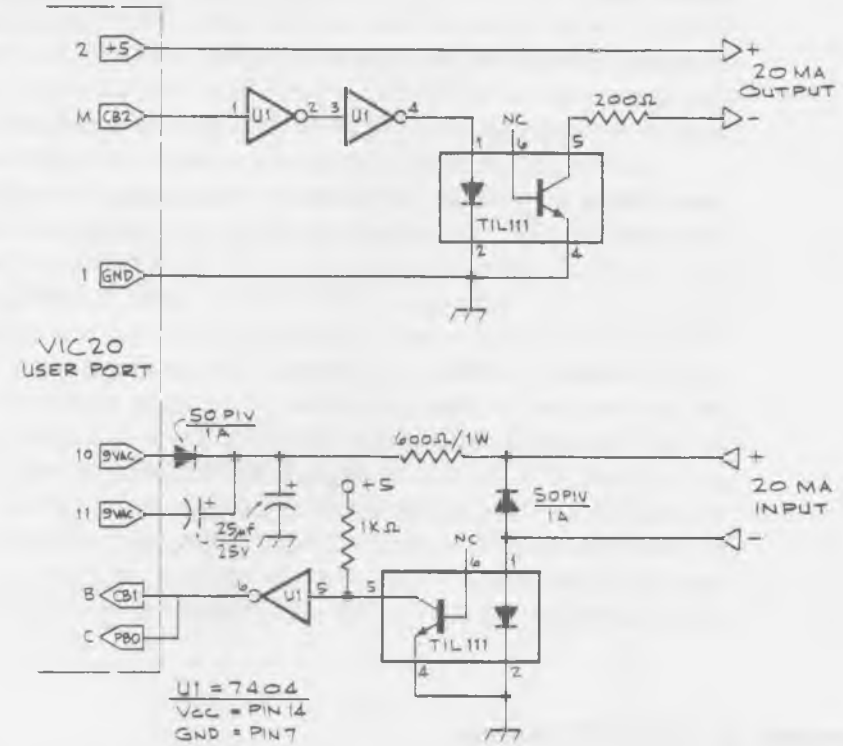
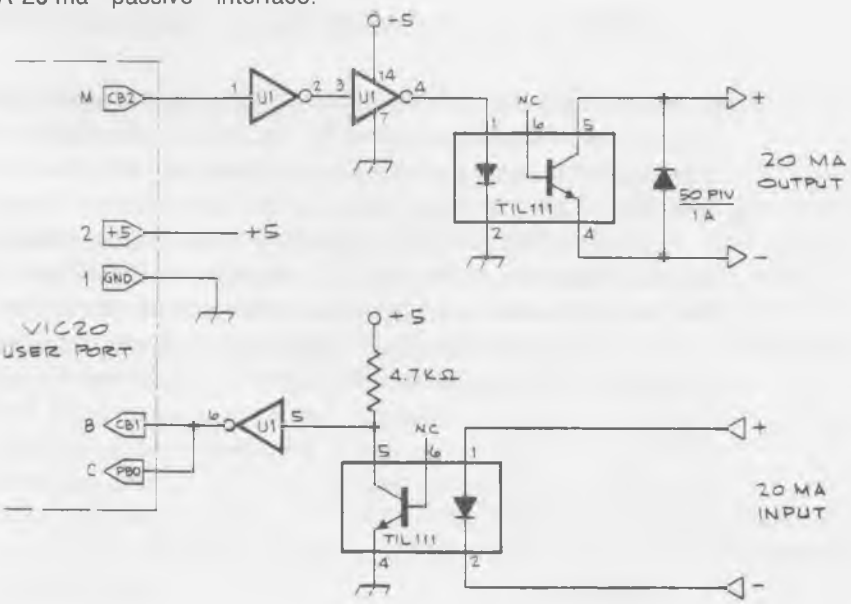


FIGURE 11–5
A 20-ma "passive" interface.



nel. The Commodore 64 software supports the RS-232 channel in a very sophisticated way. Briefly, the channel is treated as system device 2. The channel can be written to from BASIC with a PRINT# command. That command supports all the features of BASIC's PRINT command except that output goes to the RS-232 device rather than the screen. Similarly, input is through either the GET# or the INPUT# commands.

The RS-232 characters are generated under interrupt mode so that transmission or reception of characters occurs simultaneously with the operation of BASIC. To facilitate this background operation of the RS-232 port, the Commodore 64 reserves two 256-byte buffers at the very top of memory. One is for output and one is for input. Incoming characters are stored in the buffer and are removed by the user's program. Should the input buffer overflow, the overflow data is simply lost. It behooves the user to never let data accumulate in the input buffer for too long a period. The output buffer allows BASIC to place its output in the buffer and proceed. Characters are automatically removed from the buffer and sent out the RS-232 channel as fast as the data rate will allow. If the output buffer should fill, however, BASIC stops dead in its tracks. BASIC must then feed output to the buffer one character at a time as characters leave the buffer. No data is lost—the system just slows down.

Opening an RS-232 Channel

The syntax for opening an RS-232 channel is as follows:

```
OPEN 1f, 2, 0, "[Control Register] [Command Register]"
```

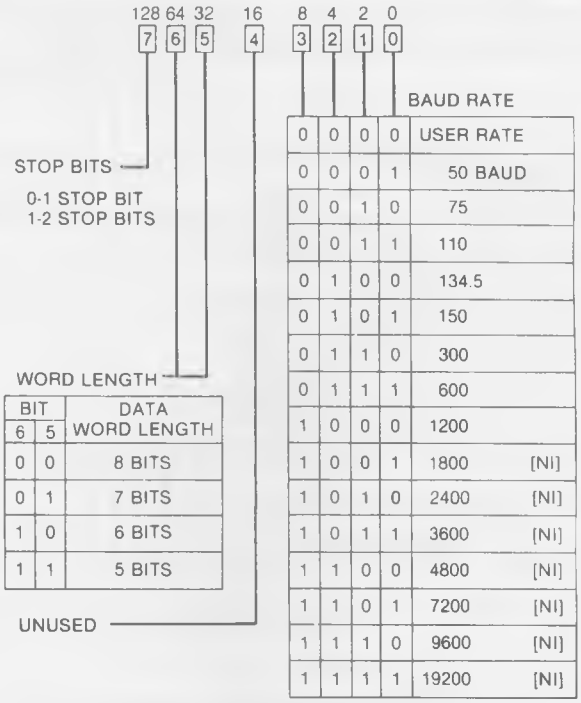
1f refers to the logical file number. This is any number between 1 and 255, and is arbitrarily assigned by the user. If 1f is greater than 127, a line feed will accompany each carriage return. If your device automatically line feeds after a carriage return, then use a number from 1 to 127.

[Control Register] is a single-byte character that configures the format and baud rate of the RS-232 transmission (see Figure 11-6). Note that the quotes indicate that this argument must be a string character.

[Command Register] is a single-byte character controlling parity and handshake configuration (see Figure 11-7). Note that this is also a string character. Here is an example. Let's say your terminal turns out to be 300 baud, has even parity, has an 8-bit word length, and uses 2 stop bits. From Figure 11-6 we see that bits 7, 2, and 1 of the control register must be set. The code

```
10000110
```

FIGURE 11–6
Bit assignments for the C64's control register. Courtesy of Commodore Business Machines.



can be given as a string character by using the CHR$ command in BASIC.

```
CHR$ (128 + 4 + 2)
```

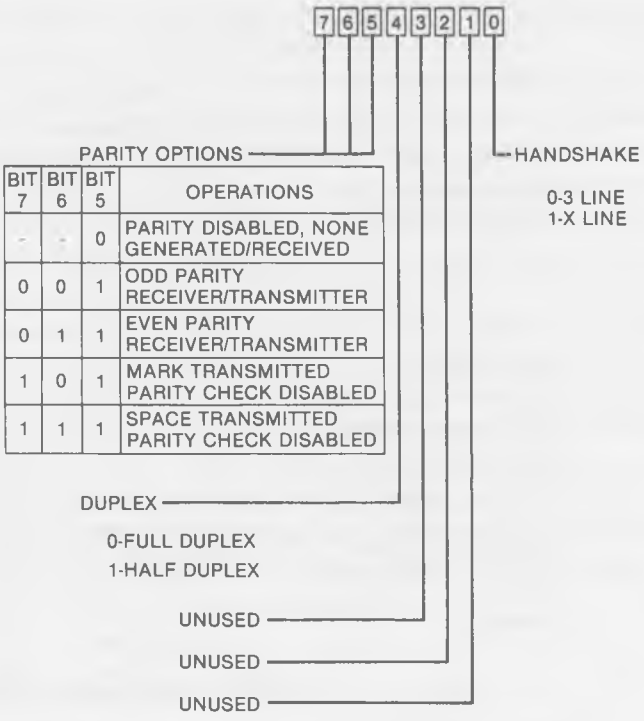
The numbers are the result of setting bit 7, the 128-weight bit, bit 2, the 4-weight bit, and bit 1, the 2-weight bit (see Chapter 1).

For the command register, we will set bits 6, 5, and 4. The duplex option, in this case, refers to whether the Commodore 64 should simultaneously be able to send and receive data. In the half-duplex mode, it can only do one or the other at a time. In general, this bit should be set; otherwise, input data might be lost. Bit 0 refers to whether the user wants to implement a handshake. If this bit is 0, the status of all handshake lines is ignored and data is transmitted at will.

PRINT #1f

Assuming that your terminal does not have automatic line feed (hence the 128 for the logical file number), the following program will send a message to the terminal.

FIGURE 11–7
Bit assignments for the C64's command register. Courtesy of Commodore Business Machines.



BIT 7	BIT 6	BIT 5	OPERATIONS
-	-	0	PARITY DISABLED, NONE GENERATED/RECEIVED
0	0	1	ODD PARITY RECEIVER/TRANSMITTER
0	1	1	EVEN PARITY RECEIVER/TRANSMITTER
1	0	1	MARK TRANSMITTED PARITY CHECK DISABLED
1	1	1	SPACE TRANSMITTED PARITY CHECK DISABLED

```
10 OPEN 128,2,0,CHR$(128+4+2)+CHR$(64+32+16)
20 PRINT#128, "HI THERE"
```

Note that the logical file number, 128, is used in the PRINT# command to identify the RS–232 file that was opened under that number. The 2 in the OPEN statement refers to the permanent device number of the RS–232 channel. It is important that you appreciate the difference between these two numbers.

GET#1f

Similarly, the Commodore 64 can receive input from the RS–232 device. The following program opens the channel and displays input on the screen.

```
10 OPEN 128,2,0,CHR$(134)+CHR$(112)
20 GET#128,B$ : IF B$="" THEN 20
30 PRINT B$;
40 GOTO 20
```

Note that in line 10 we have substituted the sums for the numbers in the control and command registers. This reduces typing. In line 20 we get a character from the buffer. If the buffer is empty, the GET# command will return a null. We test for null with the IF statement in the same line. Valid characters are printed on the screen by line 30. Line 40 loops back to repeat the process.

INPUT#1f

You can also use the INPUT# command to take data from the RS-232 channel, but only *one complete record at a time.* Data is placed in the input buffer and BASIC holds at the INPUT# statement until an end of record is signaled by a "return" ($OD) from the device.

CMD 1f

This is a particularly useful command. It causes the RS-232 channel to become the output device for listing a file. Since one of the most popular uses of the RS-232 channel is for interfacing a serial printer to the Commodore 64, this command gets used often. The following immediate-mode commands will list the program in BASIC's memory on the hypothetical RS-232 terminal described above.

```
OPEN 128,2,0,CHR$(134) + CHR$(112)
CMD 128
LIST
```

Another good use of CMD is when you wish to cause a previously written program to output all of its information to the RS-232 channel rather than to the screen. Simply preface the program with an OPEN statement and CMD. That way, all PRINT commands will direct output to the RS-232 channel.

The output can be redirected to the screen by either closing the channel (see below) or by causing a system reset. The latter can be generated by simply pushing RUN/STOP and RESTORE simultaneously.

Close 1f

The CLOSE 1f command removes the RS-232 device from the system. This is useful because it de-allocates the 512 words of buffer and makes it available to the system again. One word of caution, however: If a CLOSE is executed before the output buffer is empty, all data left in the buffer will be lost. BASIC dumps its output into the buffer and does not wait

for the data to be transmitted. Thus a CLOSE at the end of a program is likely to be executed before the data is transmitted. You can avoid a premature closure of the RS-232 channel by having the program monitor the pointers for the output buffer. As long as the pointers are not equal to each other, there is data still in the buffer. The following code tests the points and stays in a tight loop until the buffer is empty.

```
100 IF PEEK (699) <> PEEK (670) THEN 100
110 CLOSE A
```

Status

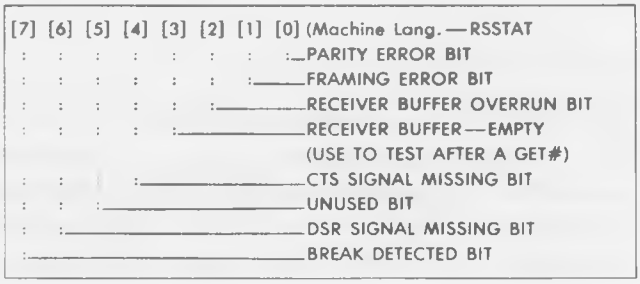
The variable ST in Commodore 64 BASIC is reserved for the value of the status register. This register tell the user the status of the *last* input/output operation. Figure 11-8 shows the meanings of the various bits in the status word for I/O through the RS-232 channel. Each of these can be checked. For example, a receiver buffer overrun condition is signaled by ST=4.

Loading BASIC Programs Through the RS-232 Port

Although the RS-232 channel can be opened as system device 2, you cannot do a LOAD from it. One of the useful applications for the RS-232 channel is to load a BASIC program from another computer, either over a direct wire link or over a telephone link via a modem (see the next chapter). The program in Listing 11-1 configures the RS-232 channel so that it operates in parallel with the Commodore 64's keyboard. Any input on the RS-232 channel will cause the Commodore 64 to operate exactly as if the corresponding key on the Commodore 64 were pressed.

The interrupt vector, locations $0314 and $0315, is changed so as to point to the patch. One of the major functions of the 60 Hz interrupt

FIGURE 11-8
Bit assignments for the C64's status register. Courtesy of Commodore Business Machines.



LISTING 11-1
RS-232 input program

```
LINE# LOC   CODE        LINE

00001  0000             ;*** MAKES THE RS-232 PORT THE INPUT DEVICE ***
00002  0000             *=$C000
00003  C000             ;-----------------------------------
00004  C000             FILNUM=128              ;FILE# 128 OR HIGHER
00005  C000             KEYCNT=$00C6
00006  C000             KEYBUF=$0277            ;START OF KEY BUFFER
00007  C000             IRQVEC=$0314            ;RAM HOOK FOR IRQ
00008  C000             SVCIRQ=$EA31            ;ROM IRQ SERVICEROUTINE
00009  C000             SETIO  =$FFC6           ;KERNAL ROUTINE
00010  C000             ;-----------------------------------
00011  C000             ; TO INITIALLIZE, USE THIS BASIC LINE:
00012  C000             ;    OPEN 128,2,0,CHR$(  ):SYS 49152:GET#128,A$
00013  C000             ;   CHR$(  ) CONTAINS THE PARAMETERS FOR YOUR DEVICE
00014  C000             ;-----------------------------------
00015  C000  08        INIT   PHP              ;SAVE IRQ MASK
00016  C001  78               SEI              ;NO INTERUPTS
00017  C002  A9 13            LDA #$13         ;LOW ORDER ADDR OF PATCH
00018  C004  8D 14 03         STA IRQVEC
00019  C007  A9 C0            LDA #$C0         ;HIGH ORDER ADDR OF PATCH
00020  C009  8D 15 03         STA IRQVEC+1
00021  C00C  A2 80            LDX #FILNUM
00022  C00E  20 C6 FF         JSR SETIO        ;PREPARE RS232
00023  C011  28               PLP
00024  C012  60               RTS
00025  C013                   ;-----------------------------------
00026  C013  48        PATCH  PHA              ;SAVE REGS
00027  C014  98               TYA
00028  C015  48               PHA
00029  C016  AC 9C 02         LDY $029C        ;BUFFER OFFSET
00030  C019  CC 9B 02         CPY $029B        ;BUFFER EMPTY?
00031  C01C  F0 0C            BEQ RTRN
00032  C01E  B1 F7            LDA ($F7),Y;GET CHR
00033  C020  EE 9C 02         INC $029C        ;MOVE POINTER
00034  C023  8D 77 02         STA KEYBUF       ;PUT CHR IN THE KEYBOARDD BUFFER
00035  C026  A9 01            LDA #01
00036  C028  85 C6            STA KEYCNT       ;TELL SYS ITS THERE
00037  C02A  68        RTRN   PLA
00038  C02B  A8               TAY
00039  C02C  68               PLA
00040  C02D  4C 31 EA         JMP SVCIRQ
00041  C030             .END

ERRORS = 00000
```

in the Commodore 64 is to see if a key has been pressed. The patch simply looks to see if a character is in the input buffer and, if so, removes it and puts it into the keyboard buffer. Control is then passed on to the normal interrupt processor. The code is completely relocatable and can be located in a ROM, if you plan to use if often (see Chapter 8). To initialize the program, you must do a SYS command to the address of INIT.

Listing 11-2 is a BASIC program that opens the channel, POKEs the patch into the protected space at $C000, enables the patch, and then erases itself, leaving the RS-232 input enabled. Execute this program before loading a program over the RS-232 channel. One word of caution: When

LISTING 11-2
RS-232 input BASIC

```
READY.

1 REM ***MAKES THE RS-232 PORT THE INPUT DEVICE***
10 DATA 8 , 120 , 169 , 19 , 141 , 20 , 3 , 169
20 DATA 192 , 141 , 21 , 3 , 162 , 128 , 32 , 198
30 DATA 255 , 40 , 96 , 72 , 152 , 72 , 172 , 156
40 DATA 2 , 204 , 155 , 2 , 240 , 12 , 177 , 247
50 DATA 238 , 156 , 2 , 141 , 119 , 2 , 169 , 1
60 DATA 133 , 198 , 104 , 168 , 104 , 76 , 49 , 234
100 FOR I=0 TO 47
110 READ X:POKE 49152+I,X
120 NEXTI
130 REM ---------------------------------------
140 A$="":REM ***PUT YOUR DEVICE CODE HERE***
145 REM ---------------------------------------
150 OPEN 128,2,0,A$:SYS49152:GET#128,A$
160 NEW

READY.
```

you SAVE the loaded program, the Commodore 64 does a warm start and restores the interrupt vector. Thus you must re-enable the patch with the OPEN/SYS/GET # sequence before another program can be downloaded. Be sure to select the arguments of the OPEN command so that they agree with your system. To disable the patch, do a restore.

The Commodore 64 as a Dumb Terminal

A good use of the Commodore 64's RS-232 port is to emulate a dumb terminal, that is, to simply display on the screen all incoming ASCII codes and to send out the keypresses. Such terminals are usually used in communications networks and to access large computer systems. The BASIC programs in Listings 11-3 and 11-4 provide the software to operate in a dumb terminal mode at up to 300 baud. Terminal configurations are of two types, full-duplex and half-duplex. In the former, a keypress is received by the remote device and is immediately echoed back to the Commodore 64. Thus the appearance of your keypresses on the monitor verifies that the remote machine has, indeed, received the transmission properly. In

LISTING 11-3
Dumb terminal FD

```
5 PRINT "␣"
10 OPEN 128,2,0,CHR$(163)+CHR$(160)
20 GET #128,A$
25 GET B$
30 IF A$="" THEN 45
40 PRINT A$;
45 IF B$="" THEN 20
48 IF ASC(B$)=13 THEN PRINT#128,:GOTO20
50 PRINT#128,B$;
60 GOTO 20
```

LISTING 11-4
Dumb terminal HD

```
5 PRINT "■"
10 OPEN 128,2,0,CHR$(163)+CHR$(160)
20 GET #128,A$
25 GET B$
26 PRINT B$;
30 IF A$=""THEN 45
40 PRINT A$;
45 IF B$="" THEN 20
48 IF ASC(B$)=13 THEN PRINT#128,:GOTO20
50 PRINT#128,B$;
60 GOTO 20
```

half-duplex mode, the Commodore 64 is expected to display its own keypresses. Listing 11-3 is for a full-duplex connection and 11-4 is for half-duplex. Remember, the arguments in the OPEN statement must agree with your system's format. Listing 12-1 in the next chapter provides a much more sophisticated, self-configuring terminal emulation that is also suitable.

12

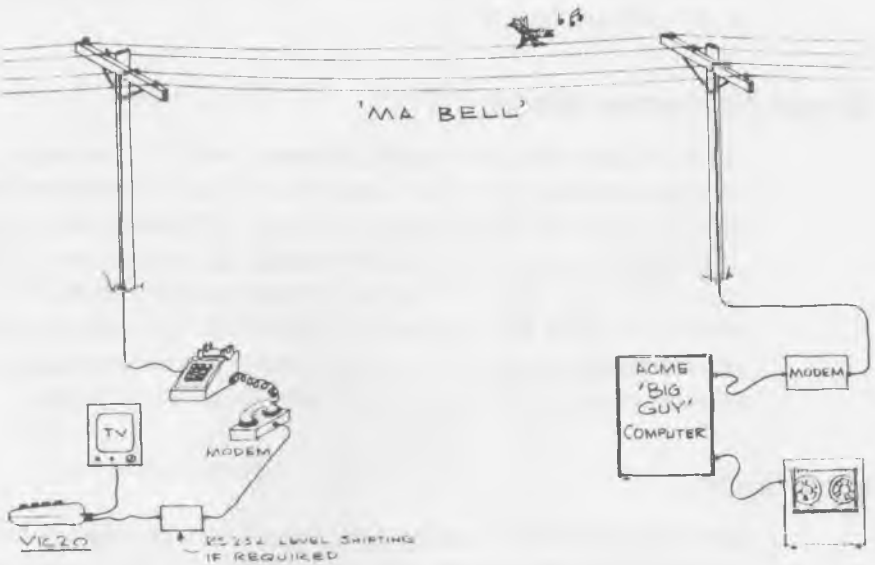
MODEMS AND THE COMMODORE 64

One of the growing uses for the personal computer is communicating with other computers, including large mainframe systems. Communications like this are almost always done over the telephone lines and require a device called the modem. Herein, we will describe the modem and how it can be easily used with the Commodore 64.

The purpose of the modem (modulator/demodulator) between the telephone line and the computer or terminal is to generate and receive audio tone in a range suitable for phone transmission and to assure that neither the terminal nor the phone system is harmed by the connection. The modem may couple to the telephone lines by either a physical wire connection or an acoustic connection; the latter uses a special cradle into which you place the handset. Figure 12–1 shows a typical configuration for telephone line communications.

The Commodore 64, with its built-in RS–232 capability, can easily accommodate a modem. *Virtually any commercial RS–232 modem can be used, but it will require the proper level shifting adaptation described earlier in Chapter 11, because the Commodore 64 signal levels at the connector are TTL.* Commodore also makes a compatible modem specially designed for the Commodore 64 that offers direct connection to the telephone line. It comes with its own software and instruction manual. The Commodore modem will not be described further here.

FIGURE 12–1
A typical modem application in which the C64 communicates with a central computer via a telephone connection.



Protocol and Mode

Communications with big computers normally require that the accessor appear as a terminal. There are certain characteristics of any big system that must be paralleled by your system. For example, the big guy establishes the baud rate and may require one or two stop bits in the character format. Also, there are usually protocol requirements, such as typing a password to access the system. You must know whether you want to talk BASIC, FORTRAN, COBOL, or whatever the system supports, and so on. The central computer may operate in full-duplex, half-duplex, or either (your choice). In full-duplex, the terminal does not put your keypresses on the screen; rather, the central computer "echoes" your characters. Full-duplex is generally preferred. In half-duplex mode, your computer must display the characters as they are typed on the keyboard.

Answer/Originate

True two-way communication is necessary for interacting with the central computer. Two-way communication must provide for two simultaneous conversations. The modem accomplishes this feat by providing two separate tone bands for modulation. The two bands are commonly designated Answer and Originate, and are determined, generally, by who calls whom.

In short, if you dial up the big guy, you have originated the conversation and use the originate tones; it will answer with the answer tones. Most modern modems allow both answer and originate, selected by a switch or by software control.

A Simple Homebrew Modem

A very reliable and easy to build 300-baud, Bell 103–compatible, answer-originate modem can be built using two ICs and a handful of parts. The heart of this project is Texas Instruments' TMS99532 modem chip. The TMS99532 comes in an 18-pin DIP package and, at the time of this writing, costs about $25. The parts for the complete modem shown in Figure 12–2 should cost about $40. Because the TMS99532 is crystal controlled, there is no calibration required. Our modem uses acoustic coupling and therefore requires no registration with your local telephone company.

Acoustic Cradle

Our prototype coupler was designed to work with the Princess-style phone, but dimensioned drawings of a cradle for the standard handset are also included. If you want to try your own cradle design, it will probably work as long as you keep the microphone and speaker close to the handset and shield out ambient noise.

If you choose to copy our design, cut the pieces from ¼-inch plywood pieces according to the plan dimensions (see Figures 12–3 and 12–4). Glue and nail all the pieces together except the speaker and microphone mounts (white glue is recommended). Affix the microphone and speaker to their mountings with a small amount of silicone glue. Drill small holes as needed for the wiring and use shielded leads for both connections. When the microphone and speaker are secure to the base pieces, attach about 30-inch leads to each, thread the leads through the drilled holes, and then glue the mountings into place. Line all surfaces inside the box with $^3/_8$-inch foam rubber (carpet padding works well) using contact cement. Last, attach a terminator for the connection to the modem board.

Circuit

Use only the exact values for the capacitors and resistors given. These were carefully selected for correct operation and are easy to find. The modem is built on a 4½-inch by 5½-inch perforated experimenter card and attaches to the Commodore 64 with the 22/44-to-12/24 adapter plug introduced in Chapter 3. Begin construction of the three power supplies. We suggest that you test for the correct voltages with the card connected

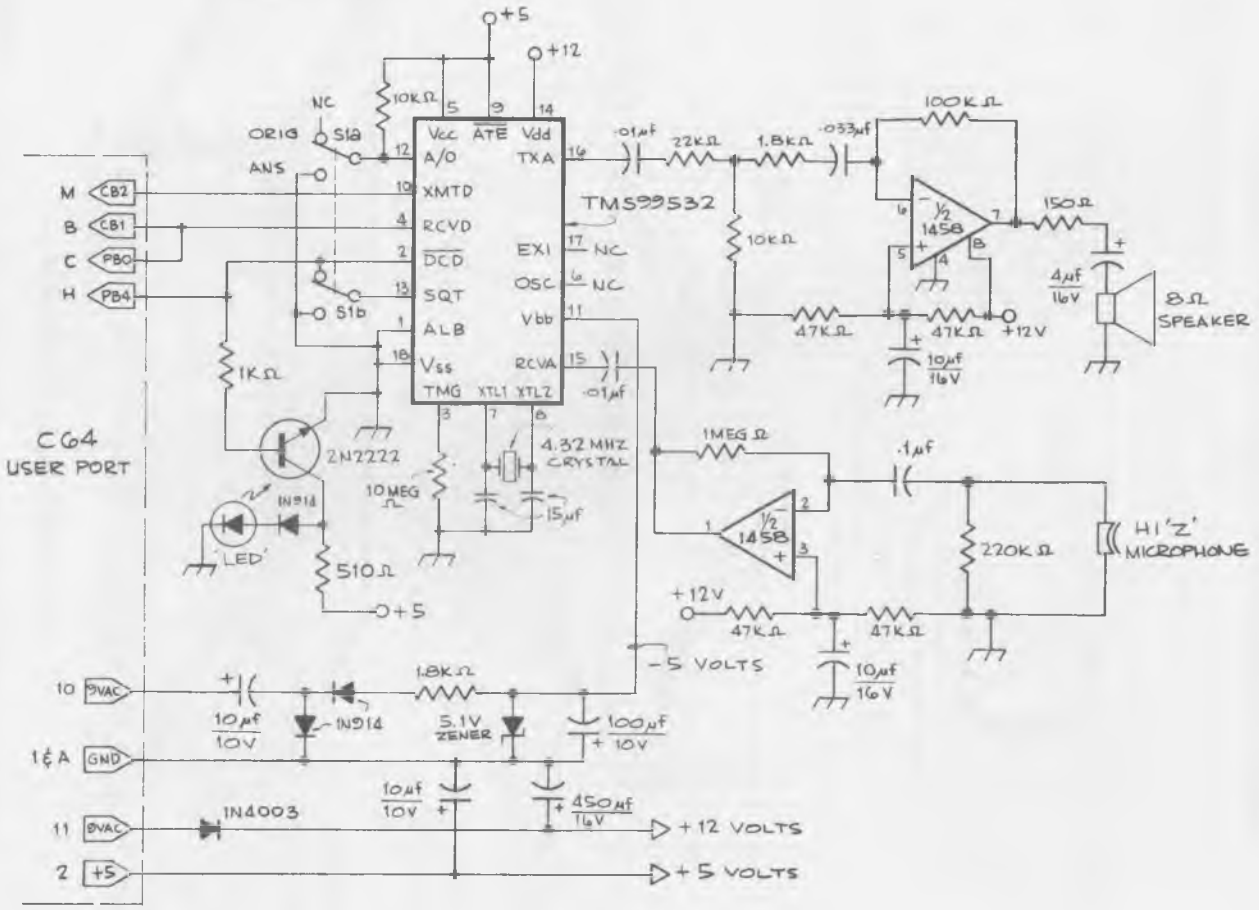
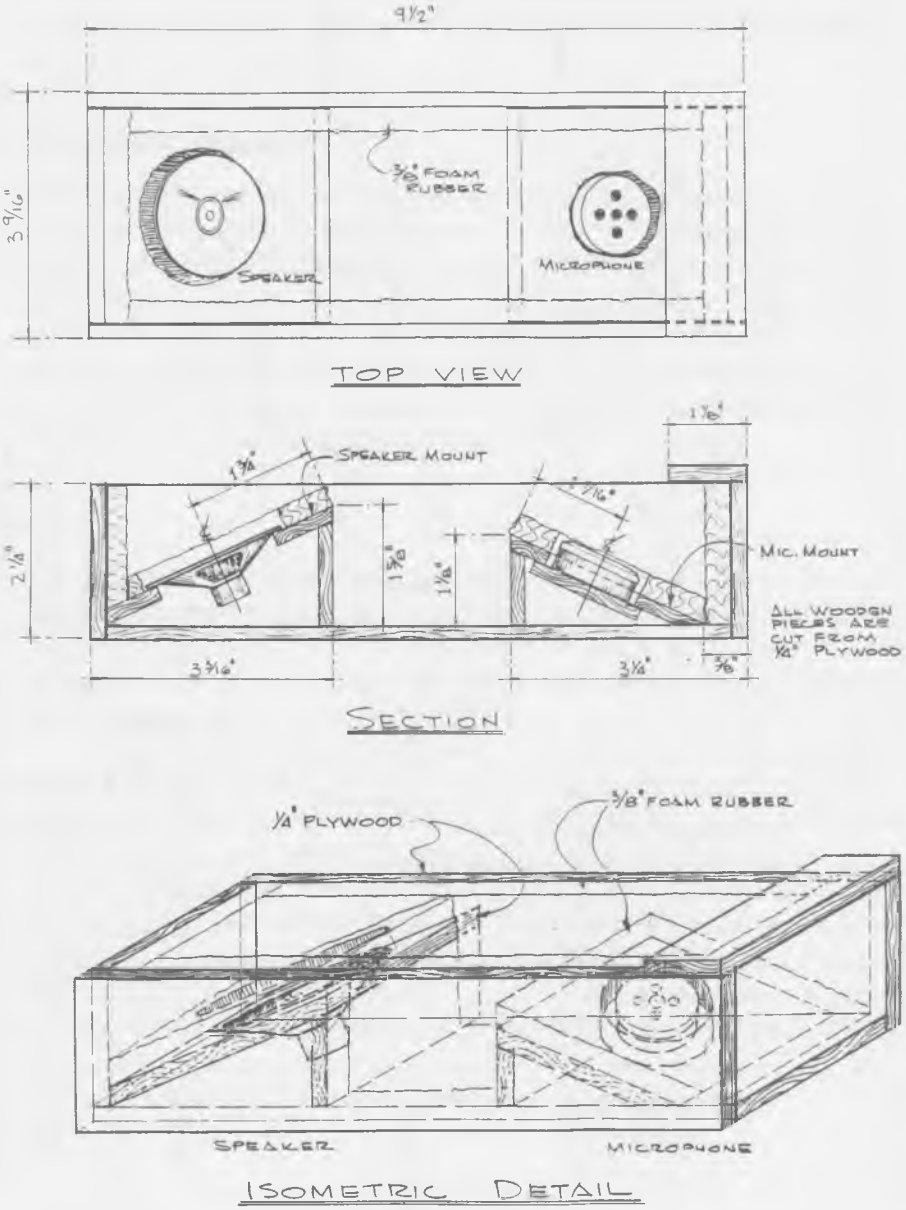


FIGURE 12–2
Schematic for modem.

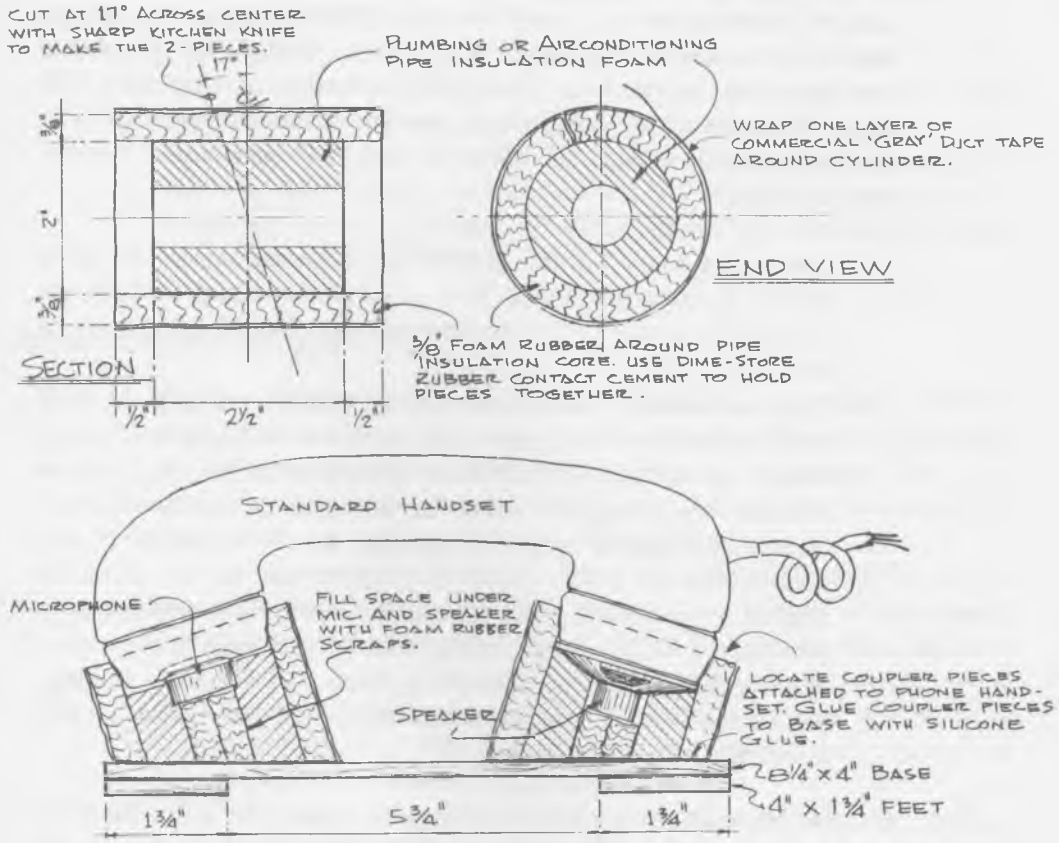
FIGURE 12–3
Dimensions for a cradle for a Princess-type telephone handset.



to the Commodore 64 and not complete the modem circuit until all three supplies are working.

Mount the 8, 16, and 18-pin sockets and the answer/originate switch on the card. Mount the remaining parts as you wish, keeping the amplifier components close to the 1458. Wire the components, but leave the ICs out of their sockets for testing.

FIGURE 12–4
Dimensions for a cradle for a standard-type telephone handset.



An optional LED may be added for indicating when communication has been established. The TMS99532 DCD (Data Carrier Detect) line goes low when the valid carrier signal is detected. This line has sufficient output levels to drive TTL logic or transistors. The 2N2222 NPN transistor we used amplifies the DCD output enough to drive an LED. The DCD may also be connected to pin H of the user port for implementing a handshake for advanced communications software. However, the handshake is not generally needed.

Testing

The simple testing sequence given below should be helpful in bringing up the modem for the first time. If you encounter problems anywhere in the procedure, correct that fault before proceeding.

With the modem chip and the 1458 removed, connect the card to the C64 and turn on the power. Test the voltages at the appropriate socket

pins as listed in Table 12–1. A dressmaker's pin might be helpful for this. Use your logic probe to check ground connections. If the voltages are right, turn off the computer and insert the chips. Connect the cradle, put the select switch to answer, and turn on the computer. Immediately, you should hear the answer carrier tone. (If not, flip the switch, because the switch might be in the wrong position.) If you do not hear the carrier tone, doublecheck your wiring. The following hints may be helpful if you still have problems.

1. Check pins 7 and 8 of the TMS99532 with a logic probe for clock pulses. If none are found, then your crystal could be bad, the TMS99532 is bad, or the capacitors on pins 7 and 8 are missing or bad.
2. Turn the power off and remove the TMS99532, but leave the 1458 in its socket. Use an alligator clip lead or a short piece of 22-gauge wire to connect the TMS99532 socket pin 15 to pin 16. Listen to the speaker as you lightly tap on the microphone. You should clearly hear the tapping amplified through the 1458 circuit. If not, doublecheck the 1458's circuit. If problems still persist, check for shorted or incorrectly wired capacitors, especially the tantalums on pins 3 and 5. Finally, you can check the speaker with an ohmmeter. When you measure the speaker's resistance on the ohms xl scale, you should hear a click, and the meter should read between 5 and 10 ohms.

If the modem has passed the above tests, type in the following short program, which will allow you to send a keypress to the modem to test its modulation. Again, the select switch should be set for the answer mode.

```
10 OPEN 2,2,0,CHR$(161)+          REM Initialize RS-232
   CHR$(160)                      for 50 baud
```

TABLE 12-1: IC Socket Voltage Chart for Modem

Pin #	VDC
TMS99532	
5 & 9	+5
14	+12
11	-5
1 & 18	GND
1458	
8	+12
4	GND

```
20 GET A$ : IF A$ = "" THEN 20   (Get any key pressed)
30 PRINT#2,A;                    (Send it to modem)
40 GOTO 20                       (Loop)
```

With the program running and the answer carrier heard, press any key and you should hear the carrier warble as modulation occurs.

When you have gotten this far, you will need to call a buddy or a dial-up system to make the true test. But before any real communication is possible, you will need some software to make the Commodore 64 behave as a terminal.

Communication Program

At this point, we assume that you have built and successfully tested the above modem or have purchased a commercial unit. This section describes a simple communications program that operates the modem. For quick setups the avid hacker will ultimately want the computer to prompt for parameters and automatically start things rolling.

The program in Listing 12–1 has such an initialization routine. It is written entirely in BASIC and should be fairly easy to understand. Transmitting and receiving are done in lines 20–110 and initialization in lines 120–650. Keeping the communications section near the top and leaving out unnecessary spaces helps with execution speed. This program will work with baud rates up to 300. For higher speeds, lines 20–110 will have to be replaced with a machine language routine.

The user must input information to initialize the baud rate, number of stop bits, mode, and so on. The following list, given in order of input, details the responses expected.

1. Linefeed after carriage return—select yes if the receiving computer expects your system to transmit a linefeed code after every CR. Default is yes. (A default condition is generated by simply hitting the return key in response to the prompt.)
2. Communication rate—enter either 110 or 300. Default is 300 baud.
3. Bits per character—enter 5, 6, 7, or 8. Default is 7-bit ASCII.
4. Stop bits—enter 1 or 2. Default is 1 stop bit.
5. Communication mode—enter 1 for full-duplex or 0 for half-duplex. Default is half-duplex.
6. Parity option—enter option value from table below. Default is 1.

Value	*Option*
1	Parity Disable, None Generated or Received
2	Odd Parity —\ Receiver or Transmitter

LISTING 12-1
Com program BASIC

```
1 REM*COM.PRG
2 REM
3 GOTO 130
4 REM
20 GET#CH,I$
30 IFI$=""THEN70
40 IFI$=RO$THENI$=D$
60 PRINTI$;:IFDU=1THENPRINT#CH,I$;
70 GETO$
80 IFO$=""THEN20
90 IFO$=D$THENPRINT#CH,RO$;:GOTO100
95 PRINT#CH,O$;
100 IFO$=CR$ANDLFTHENPRINT#CH
105 IFDUTHEN20
110 PRINTO$;:GOTO20
120 REM**************
130 REM*** INITILIZE RS232 PARAMETERS
140 RU=127:R1=0:R2=0
145 REM*** DEFAULTS
150 RT=300:CS=7:SB=1:DU=0:PA=1
160 PRINT"■      ■INIT RS232":PRINT
170 REM***GET CHAN #**
180 Y$="Y":INPUT"LF AFTER CR Y/N>";Y$
190 CH=2:IF Y$="Y"THEN CH=128
200 REM***BITS PER SEC*
210 INPUT"RATE              >";RT
220 IF RT=300 THEN R1=6
230 IF RT=110 THEN R1=3
240 IF R1=0 THEN 200
250 REM***BITS PER CHAR*
260 INPUT"BITS / CHAR     >";CS
270 IF CS<5 OR CS>8 THEN 250
280 R1=R1+32*(8-CS)
290 REM***1 OR 2 STOP BITS*
300 INPUT"# STOP BITS     >";SB
310 IF SB<1 OR SB>2 THEN 290
320 R1=R1+128*(SB-1)
330 REM***DUPLEX OR HALF*
340 INPUT"1=DUP  0=.5DUP >";DU
350 IF DU<0 OR DU>1 THEN 330
360 R2=16*ABS(DU-1)
370 REM***PARITY OPTION*
380 INPUT"PARITY (1-5)    >";PA
390 IF PA<1 OR PA>5 THEN 370
400 R2=R2+32*INT((PA-1)*1.8)
410 REM***HANDSHAKE?****
420 Y$="N":INPUT"HANDSHAKE    Y/N>";Y$
430 IF Y$="Y" THEN R2=R2+1
440 REM***PARITY OPTION*
470 REM***RUBOUT CONVERSION*
480 Y$="N":INPUT"CNG RUBOUT   Y/N>";Y$
490 IF Y$="N" THEN 530
500 INPUT" ASCII RUB CODE>";RU
510 IFRU<0 OR RU>255 THEN 470
530 REM***SAVE PARAMS FOR LATER*
540 T=(PEEK(44)+8)*256
550 POKET,DU:POKET+1,CH
560 POKET+2,RU
565 REM***OPEN FILE*
570 OPEN CH,2,0,CHR$(R1)+CHR$(R2)
575 REM***RESTORE PARAMS LOST BY OPEN*
580 T=(PEEK(44)+8)*256
590 DU=PEEK(T):CH=PEEK(T+1)
600 RU=PEEK(T+2)
```

```
610 D$=CHR$(127):CR$=CHR$(18)
615 LF=0 :IF CH>127 THEN LF=1
620 IF RU<>127 THEN RO$=CHR$(RU)
630 PRINT"■      ■INITILIZED■!!"
640 PRINT:PRINT"GO- ";
645 REM***START COMMUNICATIONS ROUTINE*
650 GOTO 20
READY.
```

3 Even Parity — Receiver or Transmitter
4 Mark Transmitted — Parity Check Disabled
5 Space Transmitted — Parity Check Disabled

7. Need handshake?—select yes if a handshake is required. Default is no handshake.
8. Change delete code?—substitute a different code for the delete key. Default is 127. (An ASCII RUB-OUT).

At this point, the parameters are set and the computer will prompt GO- for communication to begin. Happy hacking!

PARTS LIST

1 Vector 4.5″ x 5.5″ circuit card #3662–5 or equivalent
1 TMS 99532 Texas Instruments single-chip Bell 103–compatible modem (Available from Hall-Mark Electronics Corp., 11333 Pagemill Drive, Dallas, TX 75243)
1 1458 dual 741 op-amp
1 4.032-mHz crystal. Specs: (HC18/U holder), (–20 to 70°C), (+/–.005% temp. stable), (+/–.005% freq. tol.), (22-pf loading), (70-ohm series res.), (2-mw drive level), (Fundamental mode) Available for $5.25 postpaid from The Bit Stop, 5959 South Shenandoah Rd., Mobile, AL 36608.
1 18-pin wire-wrap socket
1 16-pin wire-wrap socket
1 8-pin wire-wrap socket
1 16-pin DIP header plug
3 IN914 diodes
1 1N4002 rectifier diode
1 2N2222 NPN transistor
1 5.1V zener diode
1 green LED
1 150-ohm ¼W resistor
1 510-ohm ¼W resistor
1 1000-ohm ¼W resistor

1 1800-ohm ¼W resistor
1 10,000-ohm ¼W resistor
1 22,000-ohm ¼W resistor
4 47,000-ohm ¼W resistors
1 100,000-ohm ¼W resistor
1 220,000-ohm ¼W resistor
1 1 meg-ohm ¼W resistor
1 10 meg-ohm ¼W resistor
2 15-pf 25V ceramic capacitors
3 .01-mfd 50V Mylar capacitors
1 .033-mfd 50V Mylar capacitor
1 .1-mfd 35V Mylar capacitor
1 4-mfd 35V electrolytic capacitor
4 10-mfd 35V tantalum capacitors
1 100-mfd 10V electrolytic capacitor
1 450-mfd 10V electrolytic capacitor
1 DPST toggle switch
1 Microphone element Radio Shack #270–088
1 2.25″ diameter speaker Radio Shack #40–246

Miscellaneous

2 sq. ft. ¼-inch plywood
60″ small-diameter shielded cable
1 sq. ft. $^3/_8$-in.-thick foam rubber

13

OUTPUTTING OVER THE IEEE SERIAL PORT

Another important means of having your Commodore 64 communicate with the outside world is by its IEEE serial bus. The serial bus is bi-directional, that is, it can be used for both output and input. Commodore uses this bus to interface its disk drives and printers. The main advantage of the serial bus is that the Commodore 64's operating system extensively supports this bus with high-level commands that facilitate programming. Communication over the serial bus is accomplished by a complex, serialized data format over 3 wires, CLOCK, ATTENTION, and DATA. The bytes of data are sent one bit after another on the DATA line in a fashion similar to that used by the RS-232 protocol. Although the protocol for input over the serial bus is beyond the scope of this book, it is not too difficult to build an interface that can accommodate output from the serial bus. This chapter describes a relatively simple project that was originally designed to interface the Commodore 64's serial bus to a standard serial or parallel printer.

Although we have shown you how to interface such printers in a much simpler manner in the preceding chapters, interfacing a printer to the serial bus has one distinct advantage over the other methods. When interfaced to the serial bus, the printer emulates Commodore's printers and responds to the same programming commands. Thus, a printer so interfaced will be compatible with virtually all of the commercial software available for

the Commodore 64, such as word processors and spreadsheet programs. Furthermore, this interface is an ideal way to incorporate a letter-quality printer into your system to allow serious word processing. The project, of course, is not just limited to interfacing printers, as you will see later on in this chapter.

A Computerized Controller

Above, we mentioned that the protocol for communication over the serial bus was very complicated. In fact, it is so complicated that we could not think of a straightforward method for decoding the signals using available ICs. We therefore decided to design a "smart" controller that uses a microprocessor and a 2716 EPROM memory chip (see Figure 13–1). The latter contains a program which monitors the bus and picks signals off the bus when ever data is sent to it. By using a microprocessor, we were able to reduce the number of components in the interface to a bare minimum. We chose the 6802 microprocessor because its on-board RAM eliminates the need for memory chips.

We should stress to you that this is a relatively complex project. We do not recommend that you try to build it unless you have already had some previous experience in building electronic projects. Any error in the wiring will cause the interface to misfunction. Since we cannot anticipate all of the wiring errors that may be made, we can only give you a cursory outline of troubleshooting hints in this chapter. Nevertheless, if the instructions are carefully followed, the average hacker should not have too much trouble with this project.

Let's Build It

At this point, you must make a decision as to which configuration of the interface you want to build. We have included two versions, a serial version which outputs data in the standard RS–232 format and a parallel version which would be compatible with a standard parallel-type printer like the Centronics or Epson. Figure 13–2 shows the additional schematics for the serial version while schematics for the parallel version appear in Figure 13–3.

Either version can be built on a 4½-inch by 4-inch experimenter card with holes on .01-inch spacing such as the Radio Shack part #276–152. The layout is not critical. However, to avoid confusion, we recommend that you use the component layout shown in Figure 13–4 or Figure 13–5.

Use wire-wrap IC sockets as indicated in the parts list and arrange the sockets on the board as indicated. Note that the beveled corner on the

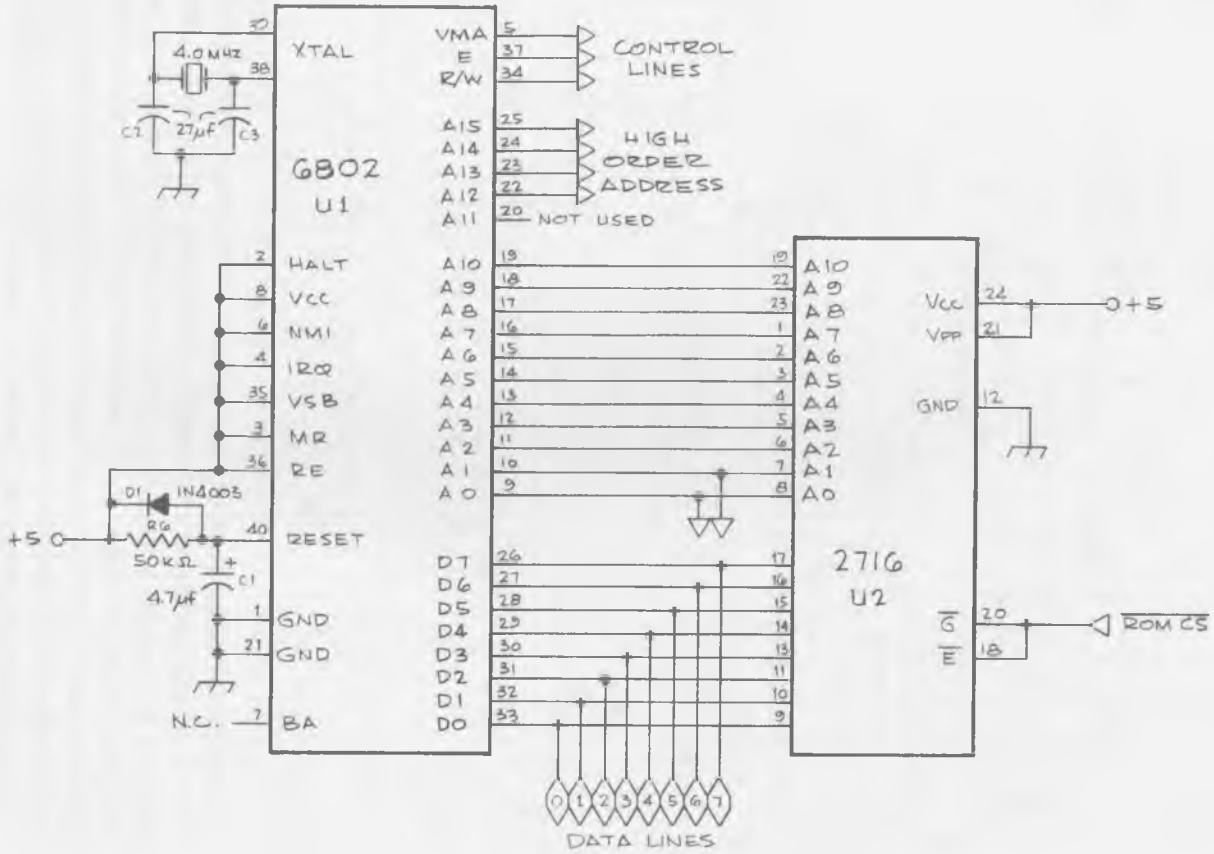
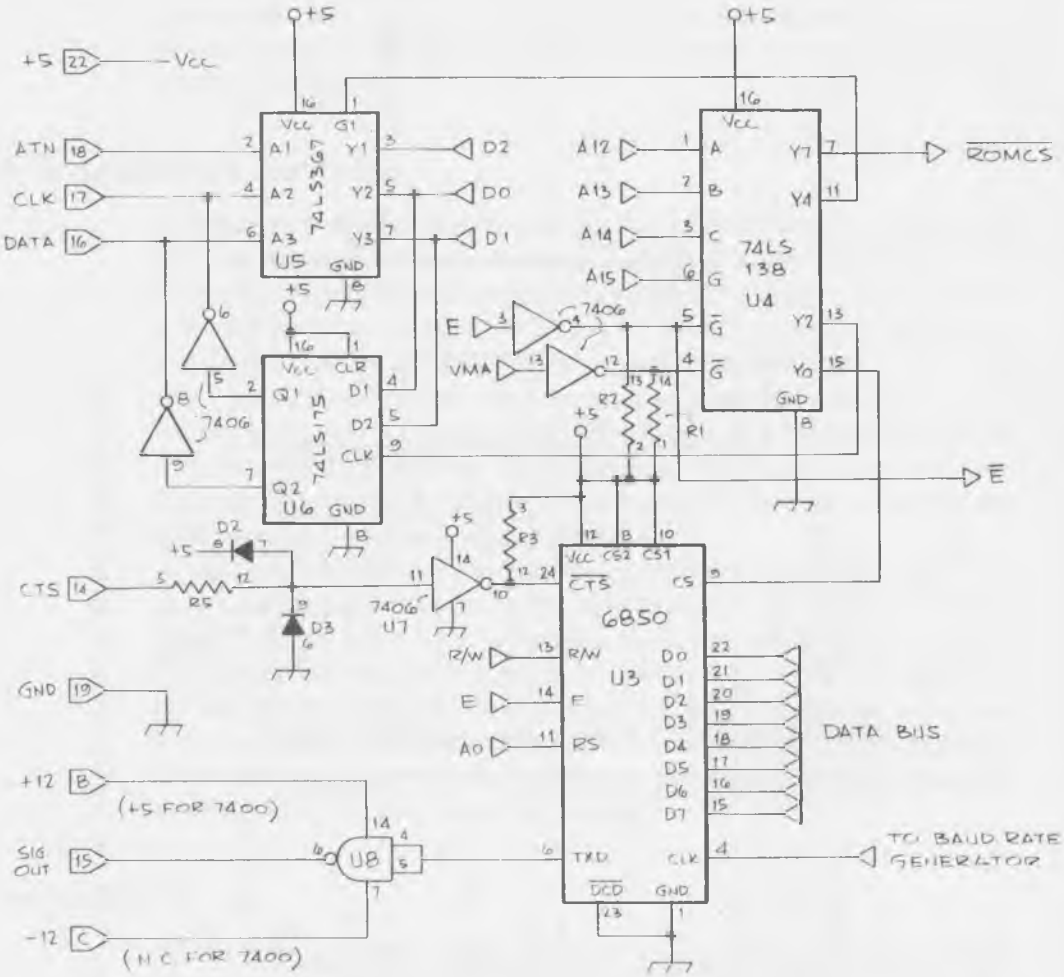


FIGURE 13–1
The 6802 microprocessor FOM and address decoding for the IEEE interface.

FIGURE 13–2
Schematics for the serial version of the IEEE interface.



sockets in the layout indicates pin 1. Fasten each socket in place with a dab of epoxy glue between the bottom of the socket and the board. When the glue has set, put Vector T44 push-in terminals on the connector pads that will be used and solder the terminals in place. Likewise, add terminals for all resistors, the crystal, and capacitors, and solder these components to the top of the terminals. Next, using wire-wrap techniques, make all of the connections shown in Figures 13–1 and either 13–2 and 13–6 or 13–3, whichever are appropriate. We recommend that those figures be duplicated on a copier; then, as each connection is made, cross off that wire on the copy with a colored pen. That should help to keep your errors to a minimum. Remember, the wiring must be perfect! Go slowly and doublecheck each connection.

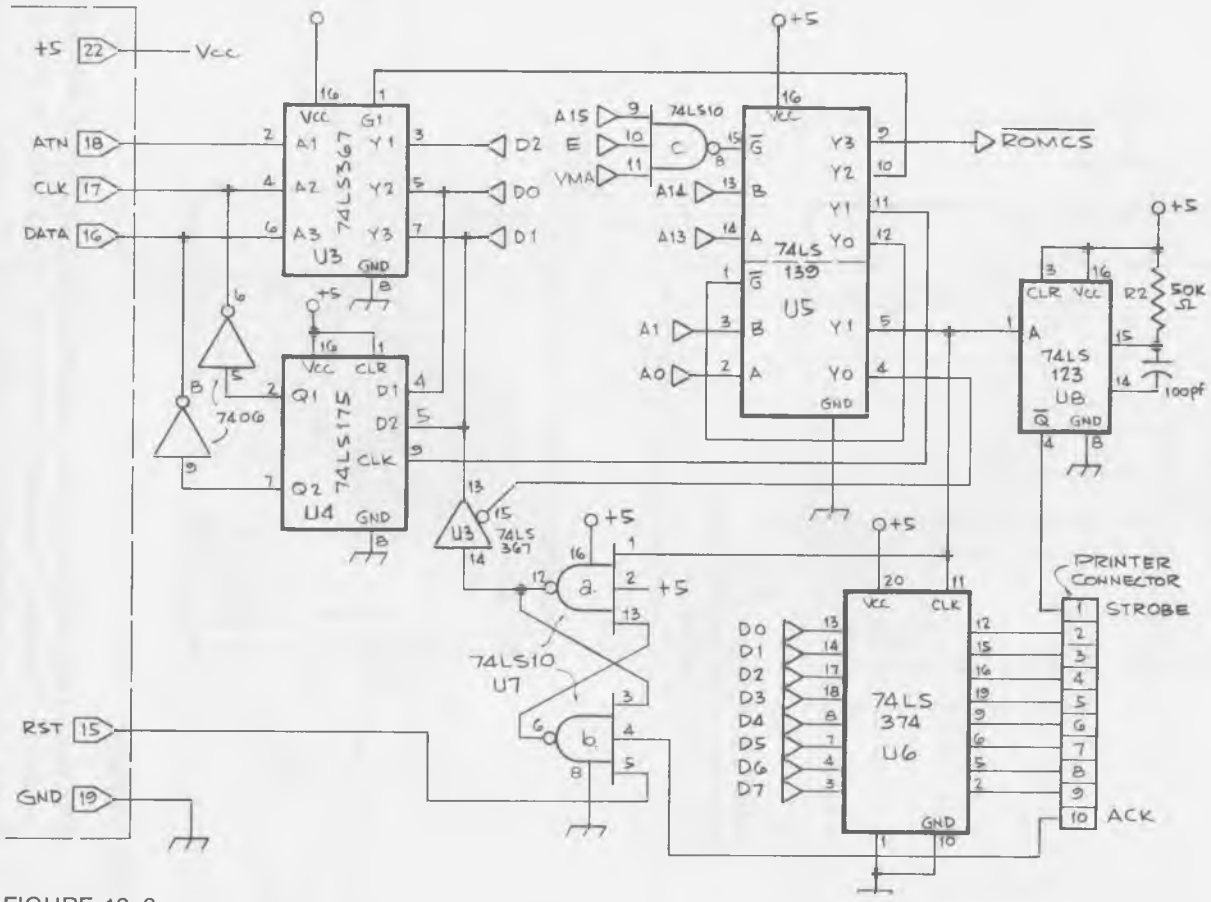


FIGURE 13–3
Connections for the parallel version of the interface.

FIGURE 13–4
Component arrangement for the serial version of the interface.

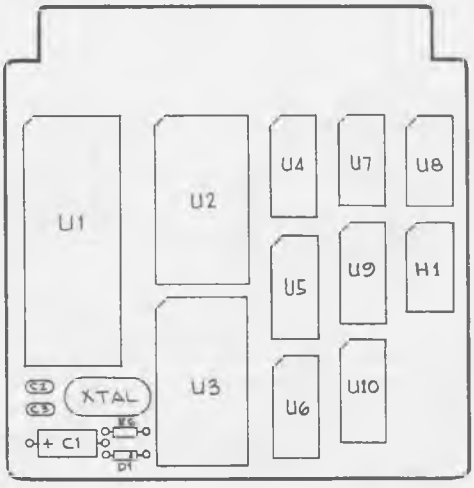
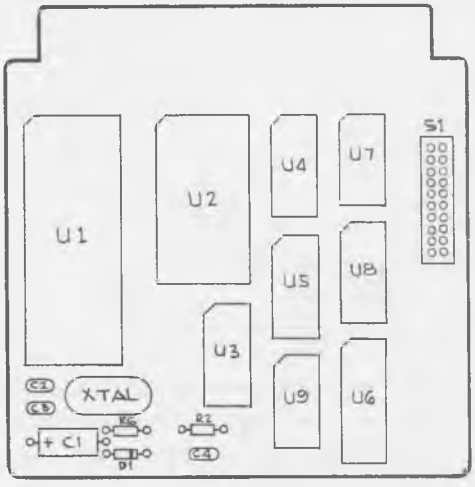


FIGURE 13–5
Component layout for the parallel version of the interface.



The Serial Version

This project also requires a +5 volt power supply capable of delivering one AMP of current. You can either buy one or build the one shown in Figure 13–7 on a separate board. In addition, if you chose to build the RS–232 version, you may need to add a ± 12V source as shown in Figure 13–8.

Hint: Virtually all of the RS–232 devices built in the past 10 years

FIGURE 13–6
The baud-rate generator for the serial version of the interface.

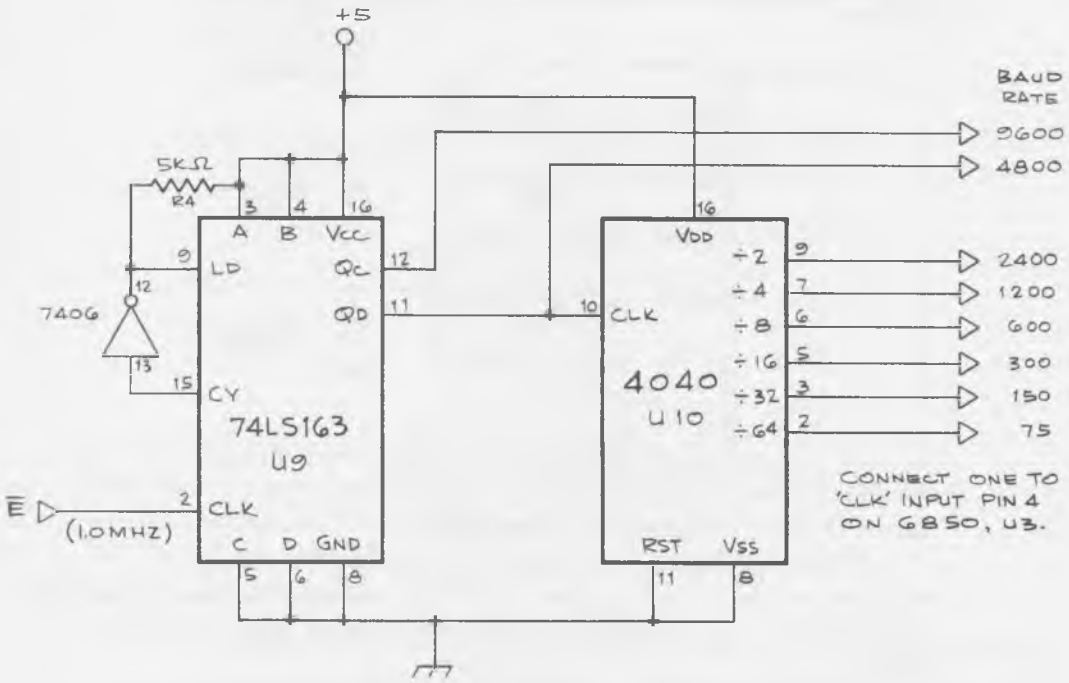
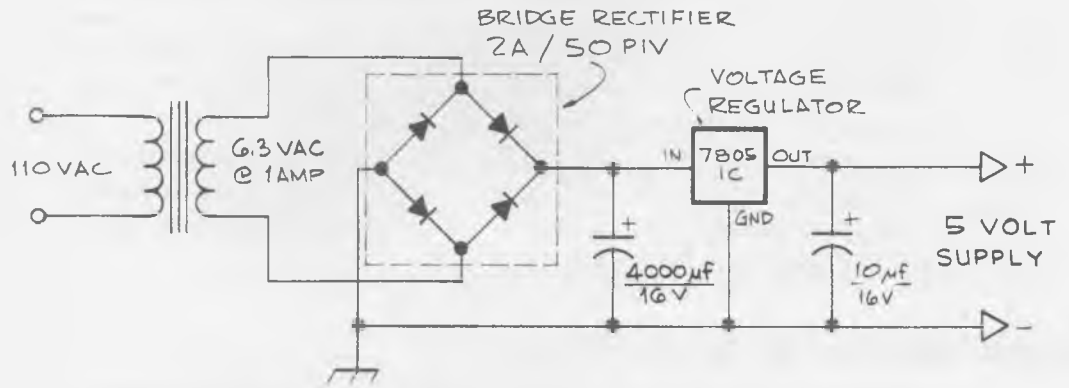
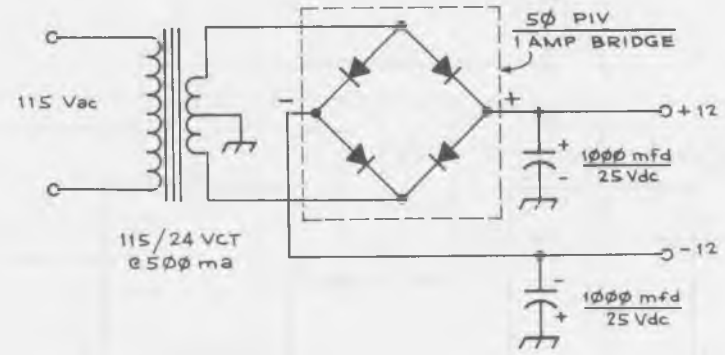


FIGURE 13–7
A 5-volt supply suitable for the printer interface.



use the 1489 level detector chip. The 1489 senses a low when −12 volts appears on the input and a high when +12 volts appears. Fortunately, 0 volts will also be sensed by the 1489 as a low and +5 volts as a high. *Thus, most RS-232 devices can be directly driven by a TTL signal.* If that is the case for your device, then the 1488 and the 12-volt supply can be eliminated—greatly simplifying the construction! If your device will operate

FIGURE 13–8
A ± 12-volt supply suitable for the printer interface.



with the TTL output, substitute a 7400 for the 1488 in Figure 13–2. For the TTL version, connect pad B of the edge connector to +5 volts rather than +12 volts. No connection is required to pad C for the TTL version.

If your device uses 20-ma current loop rather than the RS–232 format, you can substitute the appropriate 20-ma output circuit as shown in Figures 11–4 or 11–5. Be sure to include an inverter ahead of the opto-isolater.

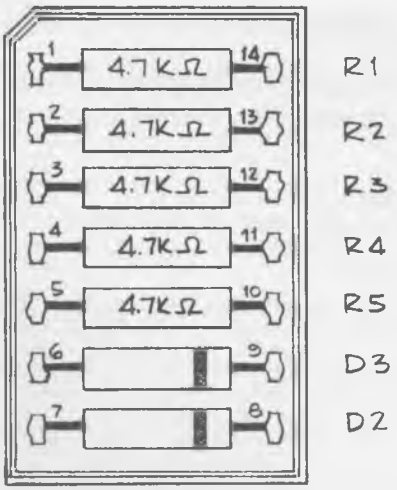
Another choice that must be made in building the serial version is the selection of a baud rate. Figure 13–6 shows a schematic for the baud rate generator. All of the common baud rates (except 110) are generated by this circuit. Choose the baud rate output pin that is correct for your device and connect that pin to pin 4 of the 6850. If your baud rate is not represented, construct a free-running oscillator, perhaps using a 555, with a frequency output equal to 16 times the required baud rate.

Finally, resistors R2–R6 and D2 and D3 have been mounted on a 7-pin header. Solder these components to the header as shown in Figure 13–9. The numbers on the schematic refer to the pin numbers on the 14 pin DIP socket for the header, H1.

Considerations for the Parallel Version

The parallel version is much simpler to build as there are no design choices that must be made prior to construction. Build the circuits as shown in Figure 13–1 and 13–3. The number on the connector to the right of the figure refers to pin numbers for the 36-pin Centronics connector. In our prototype, we glued a 20-pin header to the board as shown in Figure 13–5 and used the cable and plug assembly shown in Figure 9–1. Only the +5V supply is required for the parallel version.

FIGURE 13–9
Component layout for the header. Be sure that the diodes face in the direction shown.



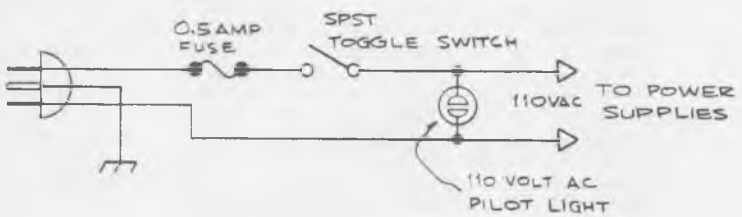
Making the Enclosure

Obtain an enclosure large enough to accommodate all of the components. Securely mount the power supply (or supplies) taking care to avoid shorts to the 110-volt lines. Install a power switch, a ½-amp fuse holder and a pilot lamp as shown in Figure 13–10.

Caution: The 110-volt AC lines are *extremely dangerous* and can yield a painful shock or even electrocution if touched! Carefully solder and insulate all 110VAC connections so that contact with these wires will not be possible while troubleshooting your circuits.

Connect the +5 volt lead from the supply to eyelet 19 of the 22/44-pin edge connector. Likewise, connect the ground lead to eyelet 19. Arrange the leads from the 6-pin DIN connector so that pin 3 (ATN) connects to eyelet 18, pin 4 (CLK) connects to eyelet 17, pin 5 (DATA) connects to eyelet 15, and pin 2 (GND) connects to eyelet 19 of the 22/44-pin edge

FIGURE 13–10
Wiring diagram for the 110VAC portion of the interface.



connector. Complete the wiring of the edge connector as is appropriate for your version of the project.

Programming the EPROM

Listing 13–1 shows the program for the ROM. The program is in 6800 machine language. Note that the format of the 6800 code is similar to the

LISTING 13–1
ROM program for printer interface (printer)

```
:ASM

                1000 * UPPER/LOWER CASE PRINTER ROM
                1010         .OR $F800
                1030         .TF IEEE.PRINT.OBJ5
                1040 *-------------------------------
001F-           1050 STACK   .EQ $001F
0020-           1060 ATNFLG  .EQ $0020
0021-           1070 CNT     .EQ $0021
0022-           1080 ACC     .EQ $0022
0023-           1090 STATUS  .EQ $0023
0024-           1100 TIMER   .EQ $0024
0025-           1110 CASEFG  .EQ $0025
8000-           1120 ACIA    .EQ $8000     ADDR. OF ACIA
A000-           1130 LATCH   .EQ $A000     ADDR. OF OUTPUT LATCH
C000-           1140 INPUT   .EQ $C000     ADDR. OF INPUT PORT
                1150 *-------------------------------
F800- 8E 00 1F  1160 START   LDS #STACK    INIT. STACK
F803- 86 03     1170         LDA #$03
F805- B7 80 00  1180         STA ACIA      RESET ACIA
F808- 86 15     1190         LDA #$15
F80A- B7 80 00  1200         STA ACIA      SET ACIA MODE
F80D- 7F A0 00  1210 RECVCL  CLR LATCH     FLOAT DATA,CLK
F810- 7F 00 25  1220         CLR CASEFG
F813- B6 C0 00  1230 BUSY    LDA INPUT
F816- 84 04     1240         ANA #$04      WAIT4 ATN HIGH
F818- 27 F9     1250         BEQ BUSY
F81A- B6 C0 00  1260 SLEEP   LDA INPUT
F81D- 84 04     1270         ANA #$04      WAIT4 ATN LOW
F81F- 26 F9     1280         BNE SLEEP
F821- 86 02     1290         LDA #$02      DATA LOW
F823- B7 A0 00  1300         STA LATCH
F826- B6 C0 00  1310 LP1     LDA INPUT
F829- 44        1320         LRA           CLOCK TO CARRY
F82A- 25 FA     1330         BCS LP1       WAIT4 LOW
F82C- B6 C0 00  1340 LP2     LDA INPUT
F82F- 44        1350         LRA           CLOCK TO CARRY
F830- 24 FA     1360         BCC LP2       WAIT4 HIGH
F832- 7F A0 00  1370         CLR LATCH     FLOAT CLK/DATA
F835- B6 C0 00  1380 LP3     LDA INPUT
F838- 44        1390         LRA           WAIT4 CLK LOW
F839- 25 FA     1400         BCS LP3
F83B- BD F8 D2  1410 DOGET   JSR GETBYT
F83E- D6 20     1420         LDB ATNFLG    BYTE UNDER ATN?
F840- 27 CB     1430         BEQ RECVCL    BACK IF NOT
F842- 81 24     1440         CPA #$24      DEVICE 4
F844- 26 C7     1450         BNE RECVCL
F846- 86 02     1470 GETEM   LDA #$02      DATA LOW
F848- B7 A0 00  1480         STA LATCH
F84B- 4F        1490         CLA           0=FIRST TIME
F84C- 97 21     1500         STA CNT
F84E- 97 23     1510         STA STATUS
```

```
F850- B6 C0 00 1520 G1      LDA INPUT
F853- 44       1530         LRA
F854- 24 FA    1540         BCC G1         WAIT4 CLK HIGH
F856- 7F A0 00 1550 FLOAT   CLR LATCH      FLOAT EM
F859- 86 10    1560 SETIME  LDA #$10
F85B- 97 24    1570         STA TIMER
F85D- 7A 00 24 1580 DECTIM  DEC TIMER
F860- 27 55    1590         BEQ CKTWO      DO TWICE
F862- B6 C0 00 1600         LDA INPUT
F865- 44       1610         LRA            WAIT4 CLK LOW
F866- 25 F5    1620         BCS DECTIM
F868- 8D 68    1630         BSR GETBYT
F86A- D6 20    1640         LDB ATNFLG
F86C- 26 22    1650         BNE TST3F
F86E- C6 FF    1660         LDB #$FF       ASSUME CASE CHANGE
F870- 81 11    1670         CPA #17
F872- 27 05    1680         BEQ STOREB
F874- 5F       1690         CLB            ASSUME NORMAL
F875- 81 91    1700         CPA #145
F877- 26 04    1710         BNE TSTFLG
F879- D7 25    1720 STOREB  STB CASEFG     STORE FLAG
F87B- 20 23    1730         BRA DATALO
F87D- 7D 00 25 1740 TSTFLG  TST CASEFG     IF 0, PRINT
F880- 27 0A    1750         BEQ GOPRNT
F882- 81 40    1760         CPA #$40       <A, PRINT
F884- 23 06    1770         BLS GOPRNT
F886- 81 5A    1780         CPA #$5A
F888- 22 02    1790         BHI GOPRNT
F88A- 8B 20    1800         ADA #$20       FLIP A-Z
F88C- 8D 71    1810 GOPRNT  BSR PRINT
F88E- 20 10    1820         BRA DATALO
F890- 81 3F    1830 TST3F   CPA #$3F       UNLISTEN?
F892- 26 06    1840         BNE TST67
F894- 7F 00 25 1850         CLR CASEFG
F897- 7E F8 0D 1860 JRECVC  JMP RECVCL     BACK TO SLEEP
F89A- 81 67    1870 TST67   CPA #$67
F89C- 26 02    1880         BNE DATALO
F89E- 97 25    1890         STA CASEFG     NON-ZERO
F8A0- C6 02    1910 DATALO  LDB #$02       DATA LOW
F8A2- F7 A0 00 1920         STB LATCH
F8A5- 96 23    1930         LDA STATUS     TEST EOI
F8A7- 27 08    1940         BEQ SKIP
F8A9- C6 0F    1950         LDB #$0F
F8AB- 5A       1960 DELAY   DCB
F8AC- 26 FD    1970         BNE DELAY
F8AE- F7 A0 00 1980         STB LATCH      FLOAT EM
F8B1- 81 40    1990 SKIP    CPA #$40       EOI ?
F8B3- 26 91    2000         BNE GETEM      MORE
F8B5- 20 E0    2010         BRA JRECVC
F8B7- 96 21    2020 CKTWO   LDA CNT
F8B9- 26 DC    2030         BNE JRECVC     SECOND TIME?
F8BB- 86 02    2040         LDA #$02
F8BD- B7 A0 00 2050         STA LATCH      DATA LOW
F8C0- C6 0B    2060         LDB #$0B
F8C2- 5A       2070 DEL     DCB
F8C3- 26 FD    2080         BNE DEL
F8C5- 7F A0 00 2090         CLR LATCH      FLOAT EM
F8C8- 86 40    2100         LDA #$40
F8CA- 97 23    2110         STA STATUS     FLAG EOI
F8CC- 7C 00 21 2120         INC CNT
F8CF- 7E F8 59 2130         JMP SETIME
               2140 *-------------------------------
F8D2- 86 08    2150 GETBYT  LDA #$08       8 BITS
F8D4- 97 21    2160         STA CNT
F8D6- C6 01    2170         LDB #$01       ASSUME ATN
F8D8- 86 04    2180         LDA #$04
F8DA- B4 C0 00 2190         ANA INPUT      TEST ATN
F8DD- 27 01    2200         BEQ STB
```

```
F8DF- 5A        2210          DCB           MAKE B ZERO
F8E0- D7 20     2220 STB      STB ATNFLG    FLAG ATN
F8E2- B6 C0 00  2230 BITLP    LDA INPUT
F8E5- 44        2240          LRA           WAIT4 CLK HI
F8E6- 24 FA     2250          BCC BITLP
F8E8- 44        2260          LRA           GET BIT
F8E9- 76 00 22  2270          ROR ACC       BIT TO ACC
F8EC- B6 C0 00  2280 L3       LDA INPUT
F8EF- 44        2290          LRA
F8F0- 25 FA     2300          BCS L3        WAIT4 CLK LOW
F8F2- 7A 00 21  2310          DEC CNT
F8F5- 26 EB     2320          BNE BITLP
F8F7- 86 02     2330          LDA #$02      HOLDOFF
F8F9- B7 A0 00  2340          STA LATCH
F8FC- 96 22     2350          LDA ACC       GET CHAR.
F8FE- 39        2360          RTS
                2370 *--------------------------------
F8FF- F6 80 00  2390 PRINT    LDB ACIA
F902- C4 02     2400          ANB #$02      TEST XMT RDY
F904- 27 F9     2410          BEQ PRINT
F906- 84 7F     2420          ANA #$7F      STRIP BIT 7
F908- B7 80 01  2430          STA ACIA+1    SEND CHAR.
F90B- 81 0D     2440          CPA #$0D      CAR.RET. ?
F90D- 27 05     2450          BEQ CRET
F90F- 81 0A     2460          CPA #$0A      LINE FEED ?
F911- 27 13     2470          BEQ LFDLY
F913- 39        2480          RTS
F914- CE FF FF  2490 CRET     LDX #$FFFF
F917- 09        2500 DECX     DEX
F918- 26 FD     2510          BNE DECX      DELAY FOR CRET
F91A- F6 80 00  2520 STAT     LDB ACIA
F91D- C4 02     2530          ANB #$02      LOOP UNTIL READY
F91F- 27 F9     2540          BEQ STAT
F921- 86 0A     2550          LDA #$0A      LINE FEED
F923- B7 80 01  2560          STA ACIA+1    SEND IT
F926- CE 40 00  2570 LFDLY    LDX #$4000
F929- 09        2580 DEX      DEX
F92A- 26 FD     2590          BNE DEX       DELAY
F92C- 39        2600          RTS
                2610 *--------------------------------
F92D-           2620          .BS $FFF8-*   FILLIN SPACE TO VECTORS
FFF8- F8 00     2630 IRQ      .DA START
FFFA- F8 00     2640 SWI      .DA START
FFFC- F8 00     2650 NMI      .DA START
FFFE- F8 00     2660 RST      .DA START
```

6502 code that the Commodore 64 uses, but that the instruction codes have different names. If you have built the EPROM programmer in Chapter 8 or have access to a programmer, you can copy the op-codes from our assembled listing. Don't forget to put the interrupt vectors at the top of EPROM's memory space ($FFF8 to $FFFF). If you do not have a programmer, you can purchase a preprogrammed EPROM from The Bit Stop, 5958 South Shenandoah Rd., Mobile, AL 36608, for $30 postpaid.

Notice that the EPROM is programmed the same for either the parallel or the serial version. We have, however, included two versions of the ROM program. Listing 13-1 shows a version that emulates the Commodore 1525 printer. When a CHR $(17) is sent to it, it enables the lowercase/uppercase character set. A CHR $(145) puts it back into uppercase only. This version should be used for a printer interface. We recognize that many

users may want to use the interface for other things besides just printing. For that reason, we have included a second program in Listing 13–2 which sends a straight binary representation of the code sent to it. This version

LISTING 13–2
ROM program for printer interface (binary version)

```
:ASM

                  1000 * SERIAL IEEE PRINTER ROUTINE
                  1010         .OR $F800
                  1020         .TA $0800
                  1030 *       .TF IEEE.PRINT.OBJ3
001F-             1040 STACK   .EQ $001F
0020-             1050 ATNFLG  .EQ $0020
0021-             1060 CNT     .EQ $0021
0022-             1070 ACC     .EQ $0022
0023-             1080 STATUS  .EQ $0023
0024-             1090 TIMER   .EQ $0024
8000-             1100 ACIA    .EQ $8000     ADDR. OF ACIA
A000-             1110 LATCH   .EQ $A000     ADDR. OF OUTPUT LATCH
C000-             1120 INPUT   .EQ $C000     ADDR. OF INPUT PORT
F800- 8E 00 1F    1130 START   LDS #STACK    INIT. STACK
F803- 86 03       1140         LDA ##03
F805- B7 80 00    1150         STA ACIA      RESET ACIA
F808- 86 15       1160         LDA ##$15
F80A- B7 80 00    1170         STA ACIA      SET ACIA MODE
F80D- 7F A0 00    1180 RECVCL  CLR LATCH     FLOAT DATA,CLK
F810- B6 C0 00    1200 BUSY    LDA INPUT
F813- 84 04       1210         ANA ##04      WAIT4 ATN HIGH
F815- 27 F9       1220         BEQ BUSY
F817- B6 C0 00    1230 SLEEP   LDA INPUT
F81A- 84 04       1240         ANA ##$04     WAIT4 ATN LOW
F81C- 26 F9       1250         BNE SLEEP
F81E- 86 02       1260         LDA ##$02     DATA LOW
F820- B7 A0 00    1270         STA LATCH
F823- B6 C0 00    1280 LP1     LDA INPUT
F826- 44          1290         LRA           CLOCK TO CARRY
F827- 25 FA       1300         BCS LP1       WAIT4 LOW
F829- B6 C0 00    1310 LP2     LDA INPUT
F82C- 44          1320         LRA           CLOCK TO CARRY
F82D- 24 FA       1330         BCC LP2       WAIT4 HIGH
F82F- 7F A0 00    1350         CLR LATCH     FLOAT CLK/DATA
F832- B6 C0 00    1360 LP3     LDA INPUT
F835- 44          1370         LRA           WAIT4 CLK LOW
F836- 25 FA       1380         BCS LP3
F838- 8D 69       1390 DOGET   BSR GETBYT
F83A- D6 20       1400         LDB ATNFLG    BYTE UNDER ATN?
F83C- 27 CF       1410         BEQ RECVCL    BACK IF NOT
F83E- 81 24       1420         CPA ##$24     DEVICE 4
F840- 26 CB       1430         BNE RECVCL
F842- 86 02       1440 GETEM   LDA ##$02     DATA LOW
F844- B7 A0 00    1450         STA LATCH
F847- 4F          1460         CLA           0=FIRST TIME
F848- 97 21       1470         STA CNT
F84A- 97 23       1480         STA STATUS
F84C- B6 C0 00    1490 G1      LDA INPUT
F84F- 44          1500         LRA
F850- 24 FA       1510         BCC G1        WAIT4 CLK HIGH
F852- 7F A0 00    1520 FLOAT   CLR LATCH     FLOAT EM
F855- 86 10       1540 SETIME  LDA ##$10
F857- 97 24       1550         STA TIMER
F859- 7A 00 24    1560 DECTIM  DEC TIMER
F85C- 27 2B       1570         BEQ CKTWO     DO TWICE
```

```
F861- 44       1590         LRA          WAIT4 CLK LOW
F862- 25 F5    1600         BCS DECTIM
F864- 8D 3D    1610         BSR GETBYT
F866- D6 20    1620         LDB ATNFLG
F868- 26 04    1630         BNE TST3F
F86A- 8D 64    1640         BSR PRINT
F86C- 20 04    1645         BRA LDB02
F86E- 81 3F    1650 TST3F   CPA ##3F      UNLISTEN?
F870- 27 9B    1660         BEQ RECVCL
F872- C6 02    1670 LDB02   LDB ##02      DATA LOW
F874- F7 A0 00 1680         STB LATCH
F877- 96 23    1690         LDA STATUS    TEST EOI
F879- 27 08    1700         BEQ SKIP
F87B- C6 0F    1710         LDB ##0F
F87D- 5A       1720 DELAY   DCB
F87E- 26 FD    1730         BNE DELAY
F880- F7 A0 00 1740         STB LATCH     FLOAT EM
F883- 81 40    1750 SKIP    CPA ##40      EOI ?
F885- 26 BB    1760         BNE GETEM     MORE
F887- 20 84    1770         BRA RECVCL
F889- 96 21    1780 CKTWO   LDA CNT
F88B- 26 80    1790         BNE RECVCL    SECOND TIME?
F88D- 86 02    1800         LDA ##02
F88F- B7 A0 00 1810         STA LATCH     DATA LOW
F892- C6 0B    1820         LDB ##0B
F894- 5A       1830 DEL     DCB
F895- 26 FD    1840         BNE DEL
F897- 7F A0 00 1850         CLR LATCH     FLOAT EM
F89A- 86 40    1860         LDA ##40
F89C- 97 23    1870         STA STATUS    FLAG EOI
F89E- 7C 00 21 1880         INC CNT
F8A1- 20 B2    1890         BRA SETIME
F8A3- 86 08    1910 GETBYT  LDA ##08      8 BITS
F8A5- 97 21    1920         STA CNT
F8A7- C6 01    1930         LDB ##01      ASSUME ATN
F8A9- 86 04    1940         LDA ##04
F8AB- B4 C0 00 1950         ANA INPUT     TEST ATN
F8AE- 27 01    1960         BEQ STB
F8B0- 5A       1970         DCB           MAKE B ZERO
F8B1- D7 20    1980 STB     STB ATNFLG    FLAG ATN
F8B3- B6 C0 00 1990 BITLP   LDA INPUT
F8B6- 44       2000         LRA           WAIT4 CLK HI
F8B7- 24 FA    2010         BCC BITLP
F8B9- 44       2020         LRA           GET BIT
F8BA- 76 00 22 2030         ROR ACC       BIT TO ACC
F8BD- B6 C0 00 2040 L3      LDA INPUT
F8C0- 44       2050         LRA
F8C1- 25 FA    2060         BCS L3        WAIT4 CLK LOW
F8C3- 7A 00 21 2070         DEC CNT
F8C6- 26 EB    2080         BNE BITLP
F8C8- 86 02    2082         LDA ##02
F8CA- B7 A0 00 2084         STA LATCH
F8CD- 96 22    2090         LDA ACC       GET CHAR.
F8CF- 39       2100         RTS
F8D0- F6 80 00 2110 PRINT   LDB ACIA
F8D3- C4 02    2120         ANB ##02      TEST XMT RDY
F8D5- 27 F9    2130         BEQ PRINT
F8D7- B7 80 01 2140         STA ACIA+1    SEND CHAR.
F8DA- 81 0D    2150         CPA ##0D      CAR.RET. ?
F8DC- 27 05    2160         BEQ CRET
F8DE- 81 0A    2170         CPA ##0A      LINE FEED ?
F8E0- 27 13    2180         BEQ LFDLY
F8E2- 39       2190         RTS
F8E3- CE FF FF 2200 CRET    LDX ##FFFF
F8E6- 09       2210 DECX    DEX
F8E7- 26 FD    2220         BNE DECX      DELAY FOR CRET
```

```
F8E9- F6 80 00 2230 STAT   LDB ACIA
F8EC- C4 02    2240        ANB #$02      LOOP UNTIL READY
F8EE- 27 F9    2250        BEQ STAT
F8F0- 86 0A    2260        LDA #$0A      LINE FEED
F8F2- B7 80 01 2270        STA ACIA+1    SEND IT
F8F5- CE 40 00 2280 LFDLY  LDX #$4000
F8F8- 09       2290 DEX    DEX
F8F9- 26 FD    2300        BNE DEX       DELAY
F8FB- 39       2310        RTS
F8FC-          2320        .BS $FFF8-*   FILLIN SPACE TO VECTORS
FFF8- F8 00    2330 IRQ    .DA START
FFFA- F8 00    2340 SWI    .DA START
FFFC- F8 00    2350 NMI    .DA START
FFFE- F8 00    2360 RST    .DA START
```

also works fine as an uppercase-only printer driver. In addition, it could be used to operate a serial x-y plotter or to enable a general-purpose parallel output port with handshake. All 256 possible binary codes can be output with the binary version but cannot be output with the printer version. When ordering a ROM from The Bit Stop, specify whether you want the printer version—C64PR—or the binary version—C64BR.

The Initial Smoke Test

When all of the wiring is complete, you are ready to begin the checkout. Do not insert any ICs at this time. Attach the common side of a volt/ohm meter to a ground connection and put the scale on ohm x 1. Touch the other lead to a +5V point on the board such as pin 22 on the edge connector. An open circuit should be seen. If not, a short exists between the +5V and the GND on your board. Don't go any further until that problem has been corrected. Next, check for continuity to all of the grounded pins shown in the schematics. If these all checkout, place the common lead on a 5-volt pin and check for continuity to all of the pins that were to be wired to +5 volts. Correct any problems before proceeding.

When the above tests have been made, turn on the power supply and verify that there are +5 volts on the board. Having made that test, turn off the power and insert the ICs in their sockets. When you are sure all is well, turn on the power and look for smoke. If none is seen, you can be encouraged that things may turn out for the best.

Is It Working?

Before you connect the interface to the computer, we recommend the following tests be made with a logic probe. Touch the logic probe to pin 37 of the 6802. This is E (analogous to phase 2 on the 6510) and should show continuous high-frequency pulses. If none are seen, check the crystal and its wiring. Next, touch the probe to pin 18 of the EPROM. Again,

a continuous train of pulses should be seen. If no pulses are seen, the EPROM is not being accessed. The most likely cause would be an error in the address decoder: the 74LS139 in the parallel version, or the 74LS138 in the serial version.

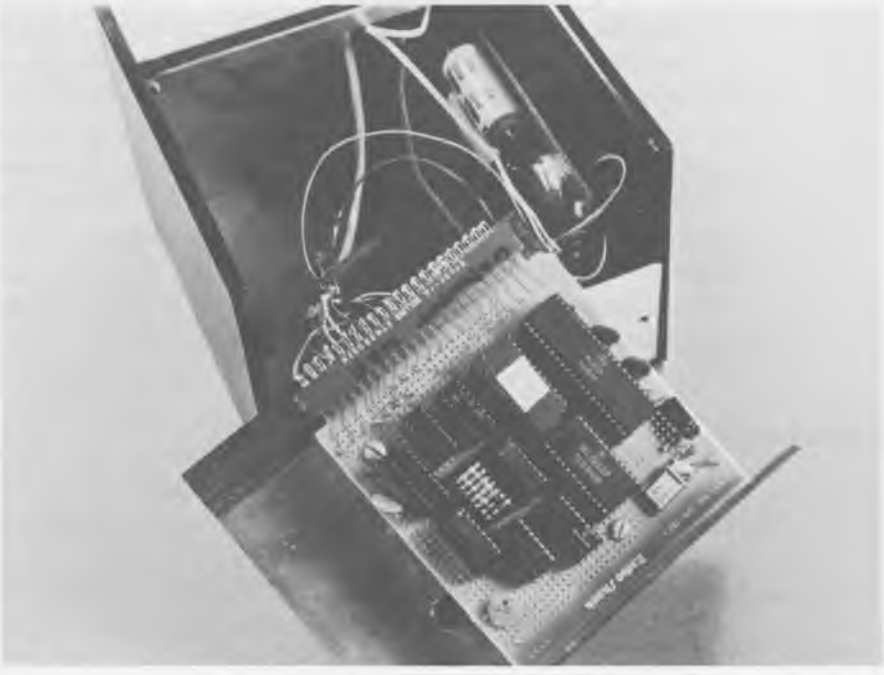
Specific Checkout Instructions for the Serial Version

The following steps refer only to the serial version of the interface. Touch the probe to pin 4 of the 6850. If a continuous pulse train is not seen here, the baud-rate generator is not working. Check the wiring of the 74163 and the 4040. Next, check for pulses on pin 1 of the 74LS367. If none are found, check the wiring on the 74LS138 address decoder. Finally, put the probe on pin 4 of U8. This should be in a logic 1 state.

If all of the above checks out, turn off the interface and connect it to the Commodore 64's serial port and connect it to the printer. Turn the computer, the printer, and the interface on and enter the following commands.

```
OPEN 4,4
CMD 4
```

FIGURE 13–11
A photo of the completed interface. This is the serial version.



The printer should have responded with READY. If the printer responded, but with gibberish, you probably have not matched the baud rates properly. Double check the settings on the printer and the wiring between the 4040 and the 6850. If the computer responded with a DEVICE NOT PRESENT error, the interface is still not working properly. Don't forget to connect the computer's ground (pin 2 of the serial socket) to pin 19 of the edge connector. (Figure 13–11 shows the completed serial version of the interface.)

If You Have a Dead System

If none, or very few, of the preceding tests were achieved, then one of two possibilities exists: 1) you have a wiring error, or 2) you have one or more bad components. Look for errors in the wiring of the data lines and the address lines. All of the data lines should have pulses. Before you turn off the power touch each IC to see if any are hot. Those that are may be defective—probably as a result of a wiring error—and will likely need to be replaced after the error is found. The 6850 and the 6802 normally run warm. Remove all of the ICs and check every connection with an ohm meter. Also look at the bottom of the board and check all pins which have no connection against the schematics to see if a wire was omitted.

Checking the Parallel Version

Put the logic probe on pin 1 of the 74LS367. A continuous train of high frequency pulses should be present. If none are found, check the 74LS139 for errors in wiring. Pin 4 of the 74LS123 should be in a high state, as should pin 1 of the 7410. Pin 12 and pin 3 of the 7410 should be in a logic low. If all of the above tests have been successful or corrected, turn off the interface and connect the computer and the printer to it. Turn on the power and enter the following commands.

```
OPEN 4,4
CMD 4
```

The printer should have printed the word READY. If it printed gibberish, then a wire has been either omitted or exchanged on the 74LS374. If the computer responded with a DEVICE NOT PRESENT error, then the communication link has not been established over the serial bus. Check the wiring on the DIN plug. Be sure that pin 2 (ground) is connected to pin 19 of the edge connector. Check the wiring on the 7406 and verify that both CLK and ATN are in a logic high state.

If no error code was returned by the Commodore 64 but the printer failed to respond, check to see if pin 13 of U3 is at a low. This is the

handshake line for the printer. No output will be sent until this line goes low. If it is not low, verify that pin 1 of the 7410 is high. If that proves to be the case, an error probably exists in the wiring of the 7410. Finally, check to see if a pulse appears on pin 11 of the 74LS374 when the CMD 4 is entered at the keyboard. If not, look for errors in U5.

Programming the Interface

If all of the above tests have been passed, you are ready to program the interface. The printer version of the interface actually contains two character sets. As it wakes up, a CHR$(65) will result in an uppercase A being printed. The character set can be changed by simply sending the control code CHR$(17). Thereafter, a CHR$(65) will cause a lowercase a to be printed. This latter mode allows you to use the printer with a word processor. The uppercase character set can be restored by sending a CHR$(145). The interface will try to send the graphic characters but since it is unlikely that they are defined in your printer's character set, spaces and meaningless characters will be substituted for them.

Listing a Program

A commonly employed application of the interface is to list a program. This can easily be done by the following commands:

```
OPEN 4,4
CMD 4
LIST
```

The CMD 4 makes the printer the system output device, as opposed to the screen. To cancel the command, issue the following:

```
PRINT #4
CLOSE 4
```

Sending Output to the Printer from a Program

You can cause a program to output the printer by opening the printer file and using the PRINT# command. The following program prints HI THERE on the printer.

```
10 OPEN 4,4
20 PRINT#4, "HI THERE"
30 CLOSE 4
```

The first 4 in the OPEN statement is the file # and should agree with the arguments in the PRINT# and CLOSE statements. The second 4 corresponds to device number four, the printer. Traditionally, Commodore has used 4 as its printer's device number, and we have done the same to maintain software compatibility. If for some reason you would like to change the device number, it can easily be done by changing line 1440 in Listing 13–1. The file # can be any number, but we recommend a 4 to avoid accidently mixing up the two numbers in your programming. The PRINT# works exactly like the PRINT command except that output goes to the printer rather than the screen.

Using the Interface with Other Devices

The interface is not just limited to printers. The interface can serve as an intelligent, general-purpose output device. For example, the serial version might be used to interface to an x-y plotter such as that offered by Houston Instruments. In that case, you should use the binary version of the ROM program so that all possible codes could be sent. The interface is quite fast and data rates of 3000 baud can be achieved over it.

The parallel version can serve as a general-purpose, 8-bit wide, parallel-output port with a handshake. A data-available pulse appears on pin 4 of U8. After the byte has been processed, the next byte will be sent when pin 1 of U7 is pulled low. By assigning different device numbers (line 1420 in Listing 13–2) several of these interfaces can share the same bus.

PARTIAL PARTS LIST—BOTH VERSIONS

6802 MPU
2716 EPROM*
7406
74LS175
74LS367
4.0-mHz Xtal (don't substitute)
2–27-pf ceramic capacitors
4.7-μf electrolytic/16V
50K/¼W resistor
4"x4½" experimenter's board Radio Shack #276-152
40-pin wire-wrap socket
24-pin wire-wrap socket
5 16-pin wire-wrap sockets
44-position-card edge connector
5-volt/1-amp regulated power supply
14-position header

6-pin DIN plug on 3 feet of cable for serial port
IN4003 silicon diode
110V/1-amp SPST switch
110V pilot lamp
110V/½-amp fuse and holder
Suitable enclosure to fit components
Vector T44 push in terminals

*EPROM must contain the program shown in Listing 13-1. A preprogrammed ROM can be ordered from The Bit Stop, 5930 South Shenandoah, Mobile, AL 36608, for $30.

ADDITIONAL PARTS FOR SERIAL VERSION

74LS138
74LS163
4040
1488 or 7400 (see text)
6850 ACIA
5 4.7K/¼W resistors
24-pin wire-wrap DIP socket
3 14-pin wire-wrap DIP sockets
±12V/50 ma power supply (may not be needed, see text)
25-pin D-female connector

ADDITIONAL PARTS FOR PARALLEL VERSION

74LS123
7410
74LS374
74LS139
20-pin wire-wrap DIP socket
14-pin wire-wrap DIP socket
.001-μf capacitor
10K/¼W resistor
20-pin "Centronics" connector on 3 feet of cable

INDEX

COMPUTERS

Make your Commodore 64 do more for you!

This clearly written guide provides dozens of simple, useful interfacing projects specifically designed to maximize the big power inside your Commodore 64. Each project comes with a complete listing of BASIC programs to run so that you can quickly tap into the Commodore 64's built-in hardware without knowing machine code. As for the minimal extra hardware required, you'll find a complete list of what you'll need inside.

Here are just some of the projects you can build and things you can do with your Commodore 64 and this handbook:

- a speech synthesizer
- a light sensor suitable for a burglar alarm
- mechanical actuators
- a telephone modem
- analog-to-digital converters
- printer interfaces
- use an audio cassette recorder for data and program storage
- and more.

James M. Downey is a professor in the Department of Physiology, College of Medicine, at the University of South Alabama.

Donald Rindsberg is president of The Bit Stop, a computer consulting firm located in Mobile, Alabama.

William Isherwood is a civil engineer with Poly Engineering, located in Mobile, Alabama.

Cover design by Hal Siegel

PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey 07632

3

ISBN 0-13-223553-6