Easy

INTERFACING PROJECTS FOR THE VIC 20

A Spectrum Book

Jim Downey, Don Rindsberg, and William Isherwood

JAMES M. DOWNEY, DON RINDSBERG,
AND WILLIAM ISHERWOOD

EASY INTERFACING PROJECTS FOR THE VIC 20

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

Library of Congress Cataloging in Publication Data

Downey, James M.
Easy interfacing projects for the VIC 20.

"A Spectrum Book."
Includes index.
1. Computer interfaces. 2. VIC 20 (Computer)
I. Rindsberg, Don. II. Isherwood, William. III. Title.
TK7887.5.D69 1984 001.64'4 84-6770
ISBN 0-13-223421-1

A Spectrum Book. Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-223421-1

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

Figure 3-3 is reprinted by permission of Heath Company.
Figures 5-3 and 5-4 are reprinted by permission of Airpax Corporation, Cheshire, CT
Figure 8-4 is reprinted by permission of Motorola.
Figure 11-2 is reprinted by permission of Commodore.

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*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

CONTENTS

PREFACE

The Commodore VIC 20 computer represents a unique entry into the personal-computer market. It combines two special features: very sophisticated internal input–output hardware and extremely low cost. The internal hardware makes the VIC 20 very easy to connect to a wide variety of devices because usually only a minimum of additional components are needed to effect the interface. Another feature of the VIC 20 is its extremely low cost. At the time of this writing, the VIC 20 is available for well under $100, which makes it ideally suited to dedicated applications. This book attempts to show the reader how to design an interface for a particular application. It then takes the reader step by step through the software development required to operate that interface. Whenever possible, we implement the software from BASIC using PEEK and POKE commands. Often, however, it is necessary to perform a given task with a machine language routine. The text shows how to implement a machine language program in the VIC 20 and how to combine machine language routines with a BASIC program.

An important consideration for any dedicated application, such as a security system or a process controller in an industrial plant, is that you must provide a convenient and inexpensive way of loading the program each time the system is to be used. The cassette drive is inexpensive, but

its slow rate of data transfer makes it unacceptable for any serious installation. The solution is to put your software in a read-only memory so that the VIC 20 automatically executes the program whenever it is turned on. This is easily done by using the built-in auto-start feature that the game cartridges use. (When a game cartridge is in the VIC 20, the computer starts the game automatically whenever the power is turned on.) We show you how to put your BASIC software in an auto-start read-only memory so that it will be instantly available at the flick of a switch. With auto-start software, neither a cassette nor even a TV monitor will be required in most applications.

Our impression of the VIC 20 is that it is extremely sophisticated. Although it is clearly not the data processor that its big brother the Commodore 64 is, it is actually an easier machine to interface. Several hardware features were either dropped in the 64's design or were made so sophisticated that programming became cumbersome. Thus, even if the price of the 64 approaches that of the VIC 20 in the future, the latter will still be the machine of choice for many applications.

Finally, we have not tried to make this book an exhaustive work on interface hardware design. Rather, it is written as an introduction for the inexperienced reader. We start with elementary concepts and build from there. Therefore, we strongly recommend this text as a teaching guide for those who want to take their first plunge into the computer hardware world. As your skills increase, you may want to attempt more ambitious projects, such as designing an interface on the memory expansion bus. An in-depth presentation of the design rules for such a project can be found in our previous publication *PET Interfacing* by Howard W. Sams & Co., Indianapolis, 1981. The organization of the PET and the VIC 20 are very similar, and most of the information in that volume is applicable to the VIC 20 as well.

EASY INTERFACING PROJECTS FOR THE VIC 20

1

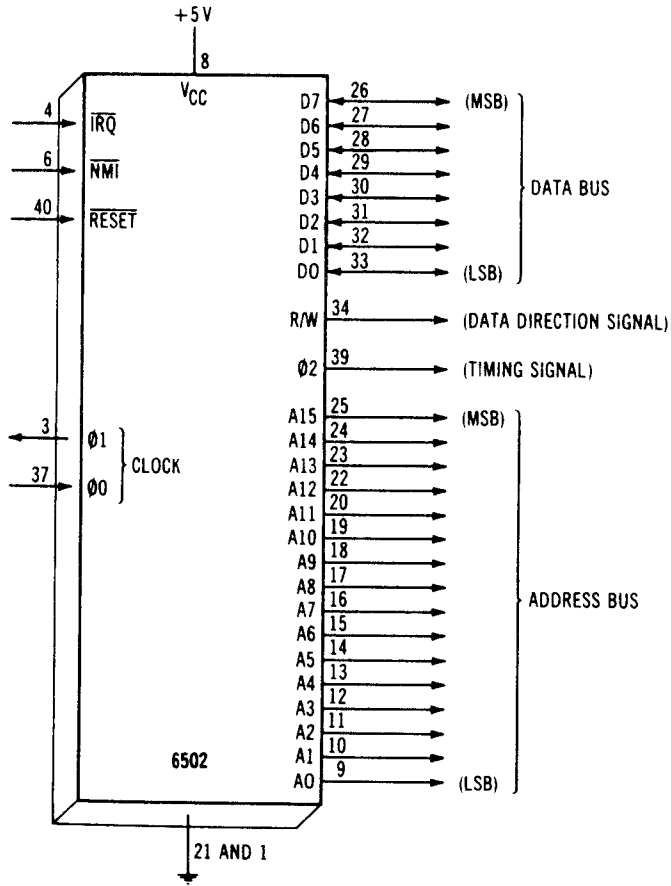
COMPUTER FUNDAMENTALS

As the major computer manufacturers have battled it out to win a share of the low-end computer market, you the consumer have become the real winner. The Commodore VIC 20 is an extremely sophisticated small computer, and yet at the time of this writing it can be bought for under $100. The low price is, in large part, a direct result of several very sophisticated, large-scale integrated circuits that were specially designed for the VIC 20 and greatly reduce the number of components inside the box. Like most of the bargain-basement computers on the market, the VIC 20 has BASIC language capabilities and a provision to store programs on cassette tapes. In addition, the VIC 20 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 VIC 20 already that most of the projects described in this book will require only a bare minimum of components and will rely heavily on optimal use of the components residing in the VIC 20's deceptively small case.

The 6502 Microprocessor

The heart of the VIC 20 is the MOS Technology 6502 microprocessor (see Figure 1-1). This is a single integrated circuit, about the size of a stick of chewing gum, which contains all the circuitry required for a comput-

FIGURE 1-1
Signals on the 6502 microprocessor.



er's central processing unit. The 6502 is a very popular microprocessor and is extremely powerful. The 6502 does several important functions in the VIC 20. First, it can perform arithmetic and logical operations internally. It can also direct data to and from memory that is external to the 6502. 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 6502 microprocessor does all of 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 6502, 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 6502 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 table below shows how bit 0 represents 2 to the zero 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	16	8	4	2	1
Bit # in the 6502	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 1's. 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 VIC 20 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 the BASIC programming software to 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. Recent microprocessors have a 16-bit word length but are still rather expensive.

The 8-bit word length is often referred to as a byte. IBM introduced the term byte to refer to one-half of their 16-bit word because they 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 represents a full word in the 6502, those 8 bits are usually called a byte rather than a word in microprocessor jargon.

Computer Anatomy

The VIC 20 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, of course, the 6502 chip, which we have already discussed. Now let's examine the other two components.

Memory

Memory in your VIC 20 computer is important because it is the place where the program is stored and it provides locations for information to be temporarily stored until the computer can process it. Memory in the VIC 20 is organized into a series of 8-bit bytes into which the computer can either deposit or retrieve 8 bits of information at a time. The 6502 selects a specific memory location by placing its binary code on the 16 address lines that emanate from the 6502 chip. Since there are 16 such lines, 2^{16} or 65,536 individual locations can be uniquely addressed. Although the 6502 can accommodate 65,536 words of memory, only a fraction of this space is actually filled in the VIC 20 computer.

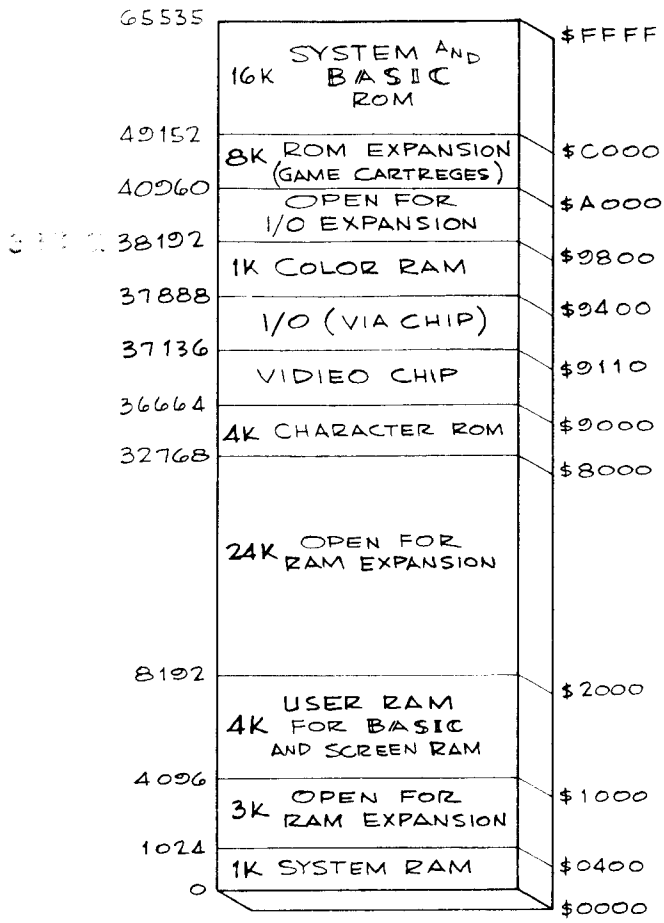
Memory is divided into two types Read Write (often called RAM, which stands for Random Access Memory) and Read-Only Memory, or ROM. The former can either be written to or read from. RAM is used in your computer to store temporary data. When you enter a BASIC program, it is held in RAM. As long as you 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 a 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 microcomputer systems consists of a single off on switch. Your VIC 20 computer comes with 16,384

words of ROM containing a bootstrap system and the BASIC language interpreter. When the 6502 is first started, it automatically starts executing the program beginning at the highest address of memory. This is where the ROM is located. The ROM 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 (short 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 VIC 20's 4K RAM, they mean that it has 1024×4, or 4096, words of Random Access Memory. Figure 1-2 shows how the memory space is used by your VIC 20.

FIGURE 1-2
Memory map for the VIC 20.



I/O Devices

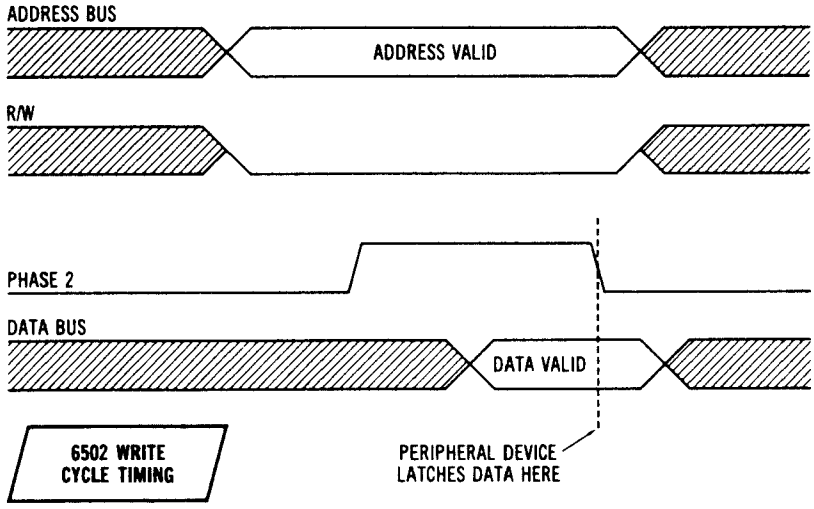
The third part of the computer that you should be aware of is the I/O (input/output) section. Regardless of how powerful a computer might be, it is only useful if it has some means of communicating with the external world. The VIC 20 can communicate with us via the TV screen and the keyboard. The TV screen is an output device. Similarly, the keyboard is an input device. Your VIC 20 is not limited, however, to only 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 VIC 20 computer as well. It could turn lights off and on in your house, receive or send Morse code over a ham radio set, measure the temperature in your garden, or a variety of other tasks if the proper I/O devices were incorporated into the computer. The techniques by which such devices are incorporated are called interfacing. These principles are actually quite simple for the VIC 20 and easily grasped.

The way an I/O device is interfaced to the 6502 is deceptively simple. All I/O devices are interfaced as if they were memory. The actual sequence of events that occur when the 6502 accesses memory involves four sets of control lines that emanate from the 6502 chip—the read-write line (R/W), the phase two clock, the 16 address lines (A0-A15), and the eight data lines (D0-D7). If the 6502 is to write data to memory location $A000, the following sequence of events takes place (see Figure 1-3). The bit pattern for $A000, 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 logic 0 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 eight 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 6502. 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 6502 must read and hold the data as it briefly appears. See Figure 1-4 for a description of the read cycle.

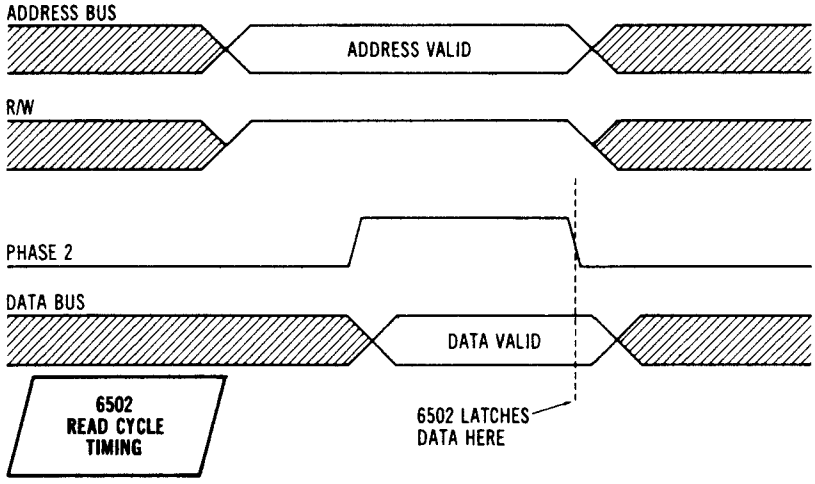
To interface an I/O device to the 6502, you must find a memory address that is not occupied by RAM or ROM and provide the logic circuitry to recognize when the address appears. Memory addresses 38912 to 40959 are reserved for I/O devices in the VIC 20. If the device is

FIGURE 1-3
A write cycle for the 6502. Note that R/W is low to indicate a write. The 6502 places the data on the data bus.



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 VIC 20's BASIC has two important commands, PEEK and POKE, which can access memory. Thus, if an I/O device is interfaced into the VIC 20 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.

FIGURE 1-4
A read cycle for the 6502. The timing is similar to the write cycle except that R/W is high to signal a read. For a read cycle, the external device must put the data on the data bus.



Although you could add an almost infinite variety of I/O devices to the VIC 20's memory space (over 2000 locations are vacant), two devices already present in the VIC 20 will meet most of your interfacing needs. These are the 6522 VIA chip, which is connected to the VIC 20's user port, and the 6560 video chip, which contains the game paddle, sound, and light pen circuitry. This book will show you how to fully use these two devices.

Data Formats

All data in the VIC 20 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 joystick to read a binary number, which is proportional to the angle of the stick. 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). Because of this standard, there are thousands of peripheral devices available that communicate through this code. Because the VIC 20 uses this code as well, all of those devices are compatible with your machine.

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 that makes up a complete electrical circuit. Today's technology permits thousands of individual components to be

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	—

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 VIC 20 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

FIGURE 1-5
Commonly used logic symbols.

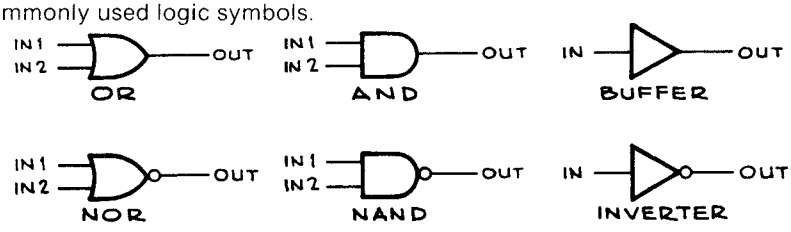
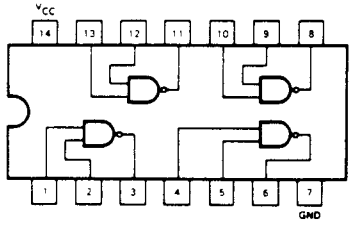
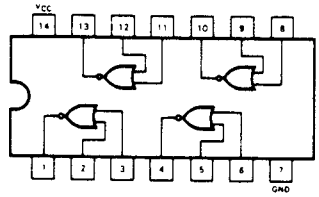


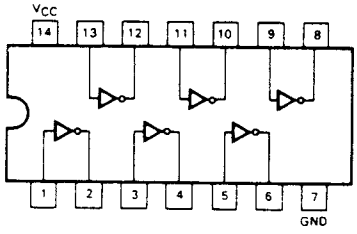
FIGURE 1-6
Chip data for the 7400, 7402, and 7404.



7400



7402

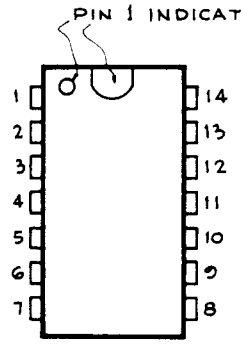


7404

quad 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 of the 7400 series ICs come in the familiar DIP (Dual Inline 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.

FIGURE 1-7
The convention for numbering the pins on an IC. This is a top view with the pins pointing away from the viewer.



INTEGRATED CIRCUIT WITH PINS POINTING AWAY FROM VIEWER.

2

USING MACHINE LANGUAGE IN THE VIC 20

In Chapter 1 we learned that all I/O devices in the VIC 20 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 it is 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 6502 microprocessor is incapable of executing the BASIC language commands directly. These commands are much too complex. Rather, the 6502 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; another may cause a byte of data to be stored in memory. There are no commands as complicated as multiplication or division in the 6502. How, then, does the VIC 20 perform the intricate commands available in BASIC? The answer is that each BASIC command causes the 6502 to perform a number of simple machine language commands in sequence until the required function is completed. A multiplication will actually involve complicated combinations of addi-

tions and shifts to arrive at the answer. This is all accomplished by a large machine language program in the VIC 20 called the BASIC interpreter. As the 6502 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 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 6502 machine language instruction executes in seven microseconds or less. Also, machine language is much better suited for manipulating individual bits than BASIC.

Many of the projects in this book rely on some machine language software. It is not really necessary to become proficient in 6502 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 6502 machine code and to demonstrate how it can be implemented in the VIC 20.

Registers Within the 6502

The 6502 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 6502. It is 8 bits wide, the same as a memory location, which 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 6502—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 6502. 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 6502 must keep a first-in last-out file of numbers. This file is kept in memory, and the stack pointer points to the next available space in memory for the file.

Fundamentals of Addressing: Hexadecimal Notation

The 6502 microprocessor has a group of 16 output lines called address lines A0 through A15. The connections from these pins to all other devices make up the "address bus." The 6502, 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:

LHLLLLLLLLHHHHH

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 0s, and all the "H" values to 1s, as follows:

0100000000011111

Since this is a number consisting of all 1s and 0s, 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 their 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

(Continued)

DECIMAL ORDER	*BINARY NUMBER*	*SINGLE-CHARACTER REPRESENTATION*
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

which can be written quickly and, more importantly, can be communicated with much less chance of error than the L/H or 0/1 notation.

Likewise, groups of eight 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

could 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 (0–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 our 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 VIC 20's BASIC must be given numbers in decimal for the PEEK and POKE commands. If you anticipate any serious machine language programming in the VIC 20, we strongly urge you to get Commodore's VICMON cartridge, which contains an excellent machine language monitor 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 6502 is represented by a 1-byte hexadecimal operation code and by a three-letter mnemonic name. Table 2-1 presents a list of the 6502 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 6502 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 6502 sees the $69 instruction, it knows that this is a 2-byte instruction and increments the program counter register by 2 before it

TABLE 2-1A: Alphabetic Listing of Op Codes

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
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

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

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

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, 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 6502 commands. A detailed description of all of the 6502 commands and addressing modes is obviously beyond the scope of this book. However, if you wish to learn more about 6502 machine language, we recommend *Apple II-6502 Assembly Language Tutor* by Richard Haskell (Prentice-Hall).

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 first column gives the address where the instruction is stored, the second 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 6502 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.

TABLE 2-1B: Hex Listing of Opcodes

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

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 6502 to start executing machine code at the decimal address following the SYS command. The 6502 will continue to execute machine code until a RTS

(return from subroutine) instruction is encountered. At that point, the 6502 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
1C0A   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 VIC 20, 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 512 bytes below the start of the screen RAM, which is high in BASIC's RAM. 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.

By far the safest place to store a machine language program is to redefine the top of BASIC's RAM. When the VIC 20 is first turned on, it does a memory test to see how much RAM is present. It then indicates to the system where the top of BASIC's memory is by putting the address of that byte in addresses $0037 and $0038 (55 and 56 decimal). Protected space can be created by your BASIC program by reducing the top of memory address by the number of bytes required to accommodate your code. Note that the top of memory address is held as the low-order byte in $37 and the high-order byte in $38. The following BASIC code reserves 20 locations at the top of memory.

```
10  TOP = PEEK (55) + 256 * PEEK (56)
20  TOP = TOP-20
30  HI = INT (TOP/256)
40  LO = TOP-HI
50  POKE 55, LO
60  POKE 56, HI
70  REM INSERT CODE AT TOP + 1
```

There are two advantages to this approach. First, if memory space is exceeded, an out of memory error will occur with a return to the system

rather than the spectacular system crash that occurs as machine code is overwritten with data. The second advantage is that this program will work in all VIC 20 system configurations because it dynamically determines memory size each time it is executed.

Using the 60 Hz Interrupt Vector

One particularly useful feature of the VIC 20 is the way it senses key presses. Every 60th of a second the 6502 is interrupted. On an interrupt, BASIC calculations are stopped, and the 6502 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 which 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, which after completion can pass control to the keyboard scan routine at $EABF. 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 VIC 20'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 $EABF, 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 an invalid address, and a system crash will result.

An example of this approach appears in Chapter 5, where a refresh pulse for a mechanical servo is continuously generated. Generation of pulses is completely transparent to the operation of BASIC, so it appears that the VIC 20 is doing two jobs at once.

3

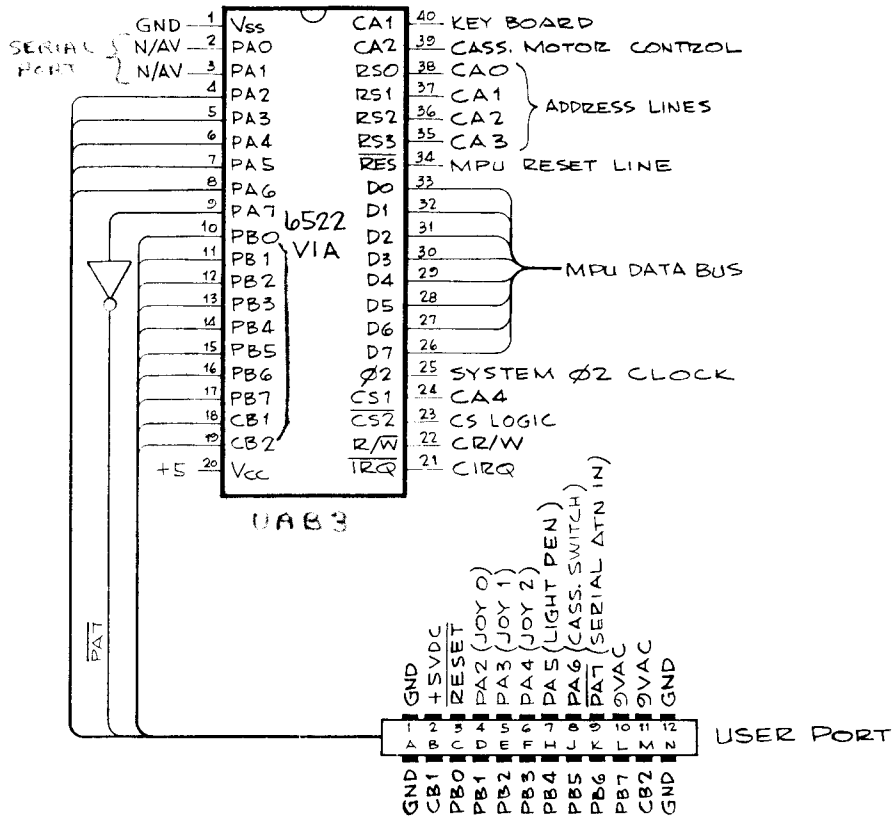
THE 6522 VIA

The 6522 Versatile Interface Adapter (VIA) is a very flexible chip. Within the 6522 IC there are two parallel, 8-bit, latching input/output ports, two 16-bit interval timers, and a serial shift register. The VIC 20 has two 6522s and uses them to control and/or communicate with the keyboard, cassette, joystick, printer, and diskette. The 6522 comes as a 40-pin integrated circuit that is primarily intended as a multifunction companion chip for the 6502 microprocessor. The VIAs greatly reduce the parts count and cost of the VIC 20 computer. It is an inexpensive building block, and, as it turns out, some of its functions can even operate simultaneously.

A generous portion of one of these 6522s is connected to the user port. 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, 3, 4, 5, 6, and 7 of port A are brought out through the general purpose user port. Port A bits 2, 3, 4, and 5, though designated by Commodore for joystick and light pen use, are available to the user for general applications as well. Bit 6 and inverted bit 7 of port A are also brought out, but if a cassette, printer, or diskette are connected to your VIC, using bits 6 and 7 can create conflicts with those devices.

FIGURE 3-1
Signals on the 6522 VIA chip and their pin assignments on the VIC 20's user port



Each 6522 occupies 16 adjacent address locations in the VIC 20's memory. These registers allow the programmer to input and output data in either parallel or serialized form, set and read the timers, read various 6522 status flags, and control various MPU interrupting features. The remainder of this chapter will explain and present examples, with experiments, of the various 6522 functions. The interface projects in the following chapters use these functions in specific applications.

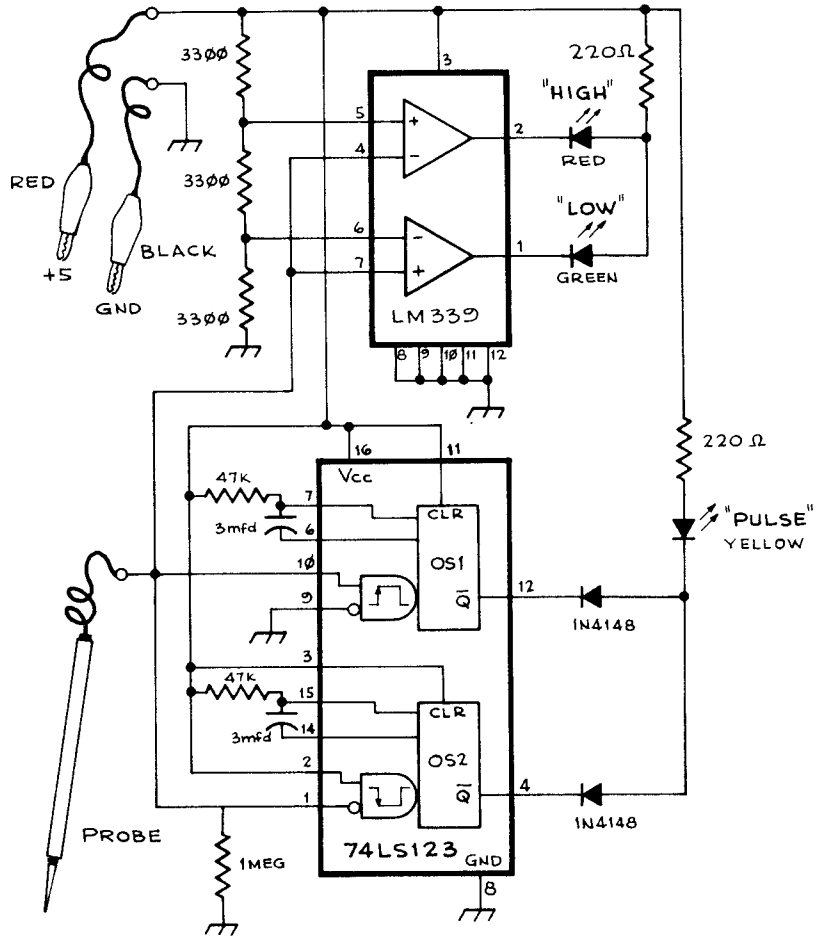
The Logic Probe

Two basic tools are required before you attempt an interfacing project. First, you should have a basic understanding of the computer fundamentals given in the first two chapters. Second, you should be equipped with a logic probe to help you debug your circuits. 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 indicators HIGH, LOW, or PULSE light when the line is in a logic high, a logic low, or a transition from one state to another, respectively. A fourth state is realized when none of the indicators are lit. This state results when the probe is at an intermediate voltage and either indicates unconnected power or ground wires 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.

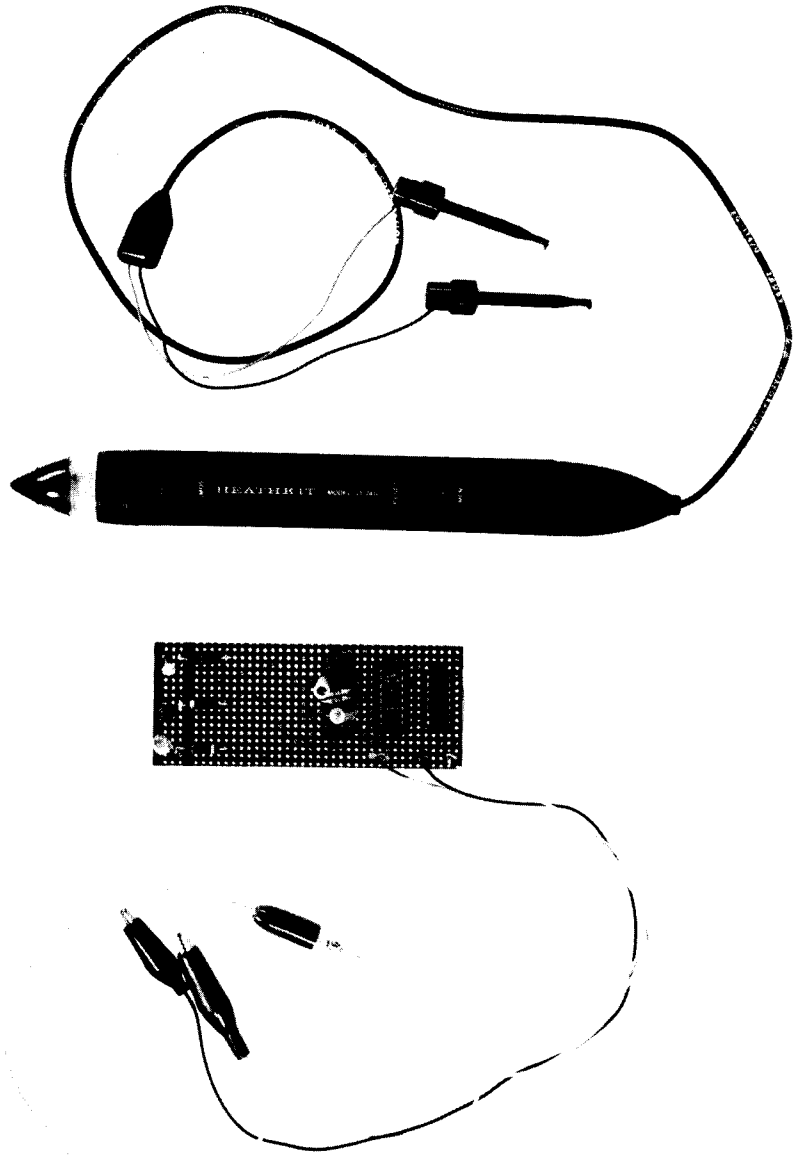
Figure 3-2 shows the schematic of a simple logic probe. It can easily be built on a 2" x 4" piece of perfboard. Wire-wrap construction, as

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



shown in Figure 3-3, should be used. Attach a 12" 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" wire should be used for

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. Copyright © 1978.



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 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.

The 8-Bit-Wide I/O Ports

One of the most useful features of the 6522 is its 8-bit-wide input/output ports. The 6522 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 eight lines of the port, P0-P7, corresponds to a bit in that port's register. Port B's register, one of the 16 address locations occupied by the 6522, is found at 37136 decimal. When all 8 bits of the port are 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 because 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 6522's circuitry. Another concept, equally useful in interfacing and nicely provided by the 6522, 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 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 uses this

technique. An interface may require only one or two of the 6522'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 use 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 6522. Port B's direction register is at 37138 decimal. Table 3-2 illustrates the concept of bit weights. These decimal values can be assigned to each bit of the 8-bit word. From 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 255 decimal at address 37138. Likewise, if we wanted to configure Port B as an 8-bit-wide parallel input port, we would store a 0 at 37138. Figure 3-4 gives an example of a bidirectional configuration for the port B direction register, which we will use in our experiments later on. This pattern will allow combined input and output via port B. The

TABLE 3-1: VIA #1 Memory Map

	MEMORY ASSIGNMENTS FOR THE 6522 AVAILABLE TO THE USER.			
Relative Address	Decimal Absolute Address	Hexidecimal Absolute Address	6522 Register Description	Register Mnemonic
0	37136	9110	Port B I O	(PB)
1	37137	9111	Port A I O (Handshake)	(AH)
2	37138	9112	Port B Direction	(BD)
3	37139	9113	Port A Direction	(AD)
4	37140	9114	Timer 1 Low Byte	(TL1)
5	37141	9115	Timer 1 High Byte	(TH1)
6	37142	9116	Timer 1 Low Load	(LC1)
7	37143	9117	Timer 1 High Load	(HC1)
8	37144	9118	Timer 2 Low Byte/Load	(L2)
9	37145	9119	Timer 2 High Byte/Load	(H2)
10	37146	911A	Shift Register I O	(SR)
11	37147	911B	Auxiliary Control Register	(ACR)
12	37148	911C	Peripheral Control Register	(PCR)
13	37149	911D	Interrupt Flag Register	(IFR)
14	37150	911E	Interrupt Flag Enable	(FE)
15	37151	911F	Port A I O (No Handshake)	(PA)

TABLE 3-2: Bit Weights

DECIMAL WEIGHT	128	64	32	16	8	4	2	1
BIT	7	6	5	4	3	2	1	0

decimal value of 240 was POKEd to location 37138 to produce the resulting I/O paths for Port B register at location 37136.

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 four LEDs for indicating output status and four pairs of input lines connected to switches for input control. This is a simple interface yet very useful in learning to use VIC's user port and some basic 6522 programming. The I/O header will be connected through the user port by mounting the header on a 4 ¼" x 5 ½" perforated experimenter card. The header is plugged into a wire-wrap socket that was 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 6522's I/O lines. Solder the four LEDs on to a 16-pin DIP header as shown, 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

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.

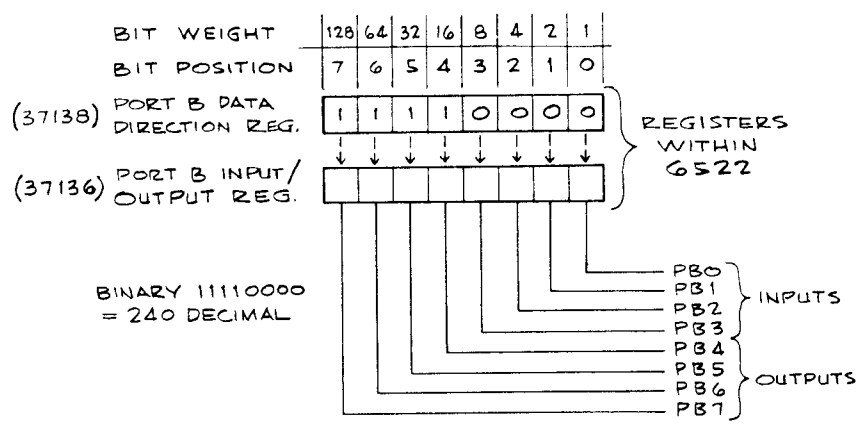
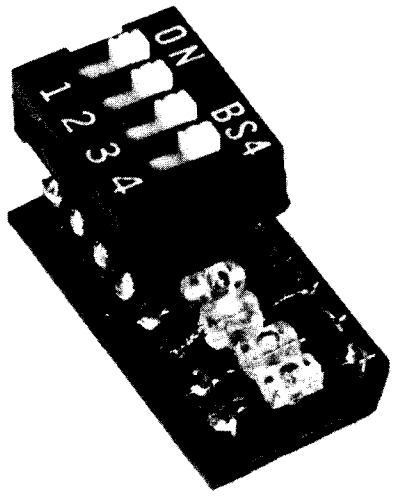


FIGURE 3-5
Photo of the header used in these experiments.



connected. Next, solder a four-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 4 ½" x 5 ½" 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 using 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 four LEDs should dimly light. Although the 6522 is initialized by the VIC 20 in the input mode, the 6522 puts out a small current because the I/O lines are pulled up to +5 volts through 10,000 ohm resistors inside the 6522. This small current is enough to dimly light the LEDs. If the LEDs do not light as indicated, turn the VIC 20 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.

Special Note: Should you accidentally short Vcc, pin 2 on the user port connector, to ground, pins 1 and 12, you will blow the 3-amp fuse on the VIC 20's main circuit board. Remove the three screws on the bottom

(old VIC-20 only!)
VERSION

FIGURE 3-6
Wiring diagram of the header.

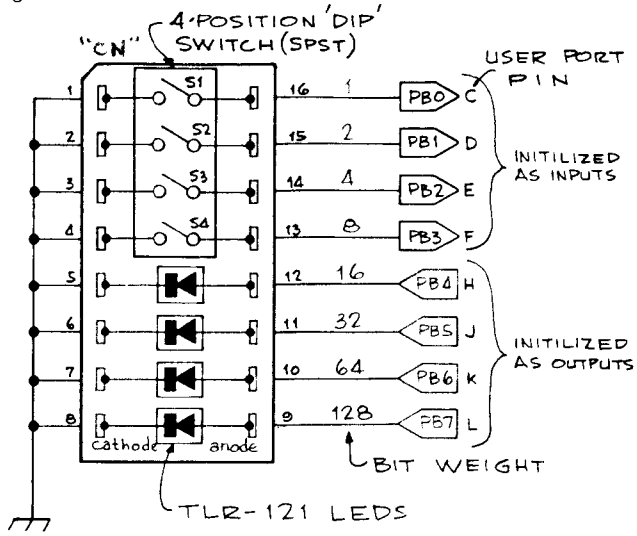
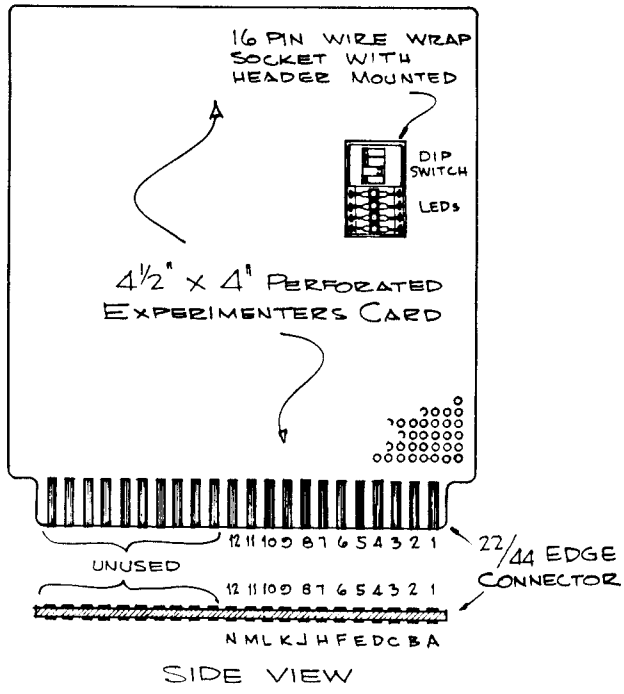


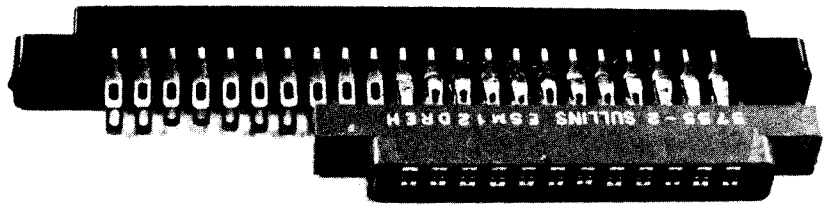
FIGURE 3-7
Details of the header card.



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 adaptor that allows the standard 22/44 experimenter's board to be plugged into the Pet's user port. This adaptor can be used with all of the projects in this book that are interfaced at the user port.



of the case and open the cover. The fuse is an odd size, but we find that an AGA-3 automotive fuse can be substituted.

Once you have everything working, enter:

```
POKE 37138, 240 (and RETURN).
```

All four 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. Since the port B register is automatically set to 0 on power-up, the value being output on the 4 high-order bits through port B is binary 0000.

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

```
POKE 37136, 16
```

The LED connected to PB4 will light. Now enter:

```
POKE 37136, (16 + 32)
```

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

```
POKE 37136, 32
```

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

```
POKE 37136, (PEEK(37136) 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 (37136)
```

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 to the zero state. For example, enter:

```
POKE 37136, 240
```

and all four LEDs will turn on. Now enter:

```
POKE 37136, (PEEK(37136) 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 VIA. You will need to understand the intent of these two logical operations to follow the programs associated with the projects.

We have 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 37138, 240                    (Bits 0-3 input; bits
                                       4-7 output)
20 INPUT "ENTER BIT WEIGHT";W         (Get output value)
30 POKE 37136, 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 four 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 VIC 20 and the 6522.

```
 1 REM BINARY COUNTER
 2 REM LEDs REPEAT COUNT 0 TO 15
10 PB = 37136 : POKE PB +2, 240      (Initialize port B direction reg.)
20 FOR A = O TO 15                   (count modulo 16)
30 POKE PB,A * 16                    (Shift 4 bits left and output 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 = 37136 : 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 6522's timers.

Many other variations can be devised to control the four 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 6522 in your VIC 20's user port can be initialized for up to 12 input bits, using both ports A and B. The four 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 bits we chose will be adequate to demonstrate how inputs may be received by the 6522.

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

```
POKE 37138, 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 four switches to the on position. Now enter:

```
PRINT PEEK (37136) AND 15
```

A 0 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 three inputs should still be low. Enter:

```
PRINT PEEK (37136) 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 that 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. This is because the pull-up resistors in the 6522 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 = 37136 : POKE PB + 2, 240   (Half input, half output)
20 IN = PEEK(PB) AND 15            (Input and strip 4 MSBs)
30 POKE PB, IN * 16                (Shift to MSBs and output)
40 GOTO 20                         (Go get next input)
```

With the program running, all four LEDs should be off. Turn OFF any one 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 an 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 controls output PB4, input PB1 controls output PB5, and so on. Additional lines of code may be inserted into the above program to allow screen acknowledgement 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 6522 Timers

There are two 16-bit interval timers in the 6522 designated as Timer #1 and Timer #2. Each has its own particular characteristics and functions. One obvious use of the 6522 in the VIC 20 is, of course, the time of day clock. Whenever the VIC 20 is powered up or reset, 6522 #1 is initialized not only to control the keyboard, but one of the timers is configured to issue an MPU (microprocessor unit) interrupt 60 times each second. The interrupt, in turn, forces the VIC 20 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.

Interval Timer

One useful application for the 6522 is to time real-world events. Although these timers are best suited for relatively short periods applicable to digital circuits and machine language programming, we will present an experiment to illustrate how to initialize and use timer #1 as a precision interval timer using BASIC. This experiment will turn the PB4 LED on and off 10 times per second. Table 3–3 shows that timer #1 can be configured to free-run by setting bits 7 and 6 of the auxiliary control register (ACR) to either 11 or 01. In the former configuration, output line PB7 will change its state each time the clock times out. In the latter configuration, PB7 is disabled.

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 flag register (IFR) and the countdown restarts from the original user-set value. Since the MPU clock is quartz crystal controlled, precision timing is guaranteed.

Our program will WAIT in line 100 for the flag bit to be set, at which time the program clears the flag and flip-flops the value of PB4. The program then returns to the WAIT condition for the next clock

TABLE 3-3: Auxiliary Control Register Bit Functions

BIT #								
7	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*FUNCTION*
0	0							One-shot mode (Timer # 1) Output to PB7 disabled
0	1							Free-running mode (Timer # 1) Output to PB7 disabled
1	0							One-shot mode (Timer # 1) Output to PB7 enabled
1	1							Free-running mode (Timer # 1) Output to PB7 enabled
		0						One-shot mode (Timer # 2)
		1						External input mode (Timer # 2)
			0	0	0			Shift register disabled
			0	0	1			Shift in from CB1 (Timer # 2)
			0	1	0			Shift in from CB1 (MPU clock)
			0	1	1			Shift in from CB1 (External clock)
			1	0	0			Free-run at rate set by timer # 2
			1	0	1			Shift out over CB1 (Timer # 2)
			1	1	0			Shift out over CB1 (MPU clock)
			1	1	1			Shift out over CB1 (External clock)
						0		No latch on port B input
						1		Latch port B input on CB1 flag
							0	No port A latch
							1	Latch port A input on CA1 flag

timeout to occur. Our program does 100 iterations and then stops. Enter and execute the following program:

```
 10 PB = 37136 : AC = PB + 11      (Set up easy access to
                                    various 6522 registers)
 20 FR = PB + 13 : FE = FR + 1
 30 TLI = PB + 4 : TH1 = PB + 5    ( " )
 40 POKE PB + 2, 16                (Make PB4-PB7
                                    outputs)
 50 POKE AC, 64                    (T1 set for free
                                    running)
 60 POKE FE, 64                    (Disable IRQ)
 70 POKE TL1, 100                  (Load low latch)
 80 POKE TH1, 199                  (Load high latch and
                                    start)
 90 FOR J = 1 TO 100               (Do task 100 times)
100 WAIT FR, 64                    (Wait for timeout)
110 POKE FR, 64                    (Reset FR flag)
120 POKE PB, ABS(PEEK(PB)-16)      (Flip-flop PB4)
130 NEXT J                         (More?)
140 END
```

The program will execute for exactly five seconds with the LED on PB4 flickering on and off, then stop. The on and off periods are equal and exactly timed to 1/20th of a second, governed by timer #1. Line 50 in the program configures the ACR for free-running with PB7 disabled. The 16-bit starting value for timer #1's counter is actually POKEd in as two 8-bit numbers designated as the low byte and the high byte. The two bytes in lines 70 and 80 were derived as follows:

```
1/20th second      = 50,000 microseconds
50,000*MPU clock   = 51,000 MPU cycles where MPU
                     clock = 1.02 MHz
TH1 (high byte)    = INT(51,000/256) = 199
TL1 (low byte)     = 51,000-(TH1 * 256) = 56
```

These numbers need only to be put into the 6522 once, and the moment the high byte is set, the timer function begins. Therefore, you should always set up the low byte first. The WAIT statement in line 100 causes the program to halt until the timeout occurs, when bit 6 in the IFR will be set (see Table 3-4). When the timeout occurs, you must immediately

TABLE 3-4: IFR and IER Bit Functions

THE INTERRUPT FLAG REGISTER

Bit #	Function
7	IRQ Status
6	Timer #1 timeout
5	Timer #2 timeout
4	CB1 pin
3	CB2 pin
2	Completion of 8 shifts
1	CA1 pin
0	CA2 pin

INTERRUPT ENABLE REGISTER

Bit #	Function
7	Enable Control
6	Timer #1
5	Timer #2
4	CB1
3	CB2
2	Shifts
1	CA1
0	CA2

clear the IFR bit 6 by POKEing a 64 back into the flag register. (Note that we clear the timer#1 flag by poking a 64 into the IFR. This is because this register does not work like the other registers. If the status bit 7 is a 0 during a write to this register, then all bits set to a 1 are cleared. If bit 7 is a 1, then all bits that are ones in the write are set. This is done in line 110. The address following the IFR is the interrupt enable register, IER.) The purpose of the interrupt enable register, as its name implies, is to enable or disable MPU interrupts. In the program above, we do not want an MPU interrupt, so the timer #1 is disabled in line 60 by clearing bit 6 to a 0. Again bit 6 is cleared by poking bit 7 as 0 and bit 6 as 1. Disabling the interrupt has no effect on the behavior of the flag register.

Special Note: Only machine language programs can take advantage of true MPU interrupts. Always disable interrupts when using 6522 functions that can cause interrupts (especially when programming in BASIC) unless you know how to handle the interrupt condition that will result. Uncontrolled interrupts will usually result in a system crash. Whenever the VIC 20 is reset, all 6522 interrupts are disabled. In other words, line 50 in the previous program could have been left out without any problems. However, we took the opportunity with the previous example to introduce the concept of the interrupt register.

Timer #1 as a Frequency Generator

You can shorten the program by having the 6522 automatically control the state of the LED by enabling PB7 in the timer control. Change line 50 so that 192 rather than 64 is POKEd to the ACR. Run the program and note that both the LED on PB4 and the LED on PB7 now flicker. Now, remove line 120 and run the program again. This time only the PB7 LED will flicker because the instruction controlling PB4 was deleted. This mode is very useful for generating a square wave signal at a very accurate frequency. The frequency in hertz can be determined by this formula:

$$\text{Frequency} = \frac{1}{(2 * \text{initial count})/1022730}$$

Frequencies between 7.8 hertz and 1022730 hertz are possible. The following is a general purpose frequency generator program.

```
 1 REM FREQUENCY GENERATOR
10 A = 1.95556E-6                          (2 X (1/MPU clock frequency)
20 PB = 37136 : BD = PB + 2                (Variables to access 6522)
30 AC = PB + 11 : TL1 = PB + 4             ( " )
40 THL = PB + 5                            ( " )
50 INPUT "ENTER FREQ"; FQ                  (Get desired frequency)
60 IF FQ<8 OR FQ>51362 THEN 50             (Test legal range)
70 CO = 1/(FQ * A) : CO = INT(.5 + CO)     (Compute time period = 1/frequency)
80 PRINT "CAN DO"; INT(1/(CD*A)+.1)        (Print closest legal frequency)
90 CH = INT(CO/256)                        (Compute high-order half)
100 CL = CO - CH * 256                     (Compute low-order half)
110 POKE AC, (192 OR PEEK (AC))            (Set up ACR for application)
```

```
120 POKE TL1, CL : POKE TH1, CH    (Set desired values)
130 GOTO 50                        (Go get next frequency)
```

Special Note: If TL1 and TH1 are initialized to 0, the output frequency will be the same as the MPU clock.

One-Shot Mode

Another interesting ability of the 6522 timer is the one-shot mode. With this mode, we can enable PB7 to go low for a period controlled by timer #1. PB7 returns to the high state when T1 times out. In Chapter 5 we will use the one-shot mode to control a servo actuator that requires a varied pulse width to control its position. We can demonstrate the one-shot mode with the following experiment.

Turn the VIC 20 off, then back on, to reset things and enter the following direct commands.

```
POKE 37138, 128   (Set PB7 to output)
POKE 37147, 128   (Select one-shot mode with PB7 enabled)
```

The LED connected to PB7 should be glowing brightly. Watch LED 7 and enter:

```
POKE 37140, 255   (Set timer #1 low byte)
POKE 37141, 255   (Set timer #1 high byte)
```

As you press RETURN, you should see LED 7 go off for about 1/15th of a second. If you want to look at this again, you need only to enter POKE 37141, 255 because the other information previously set is retained. One-fifteenth second is as long a pulse as the 6522 one-shot mode can produce. Smaller values will produce shorter width pulses, which, of course, may not be detected by the eye, but your logic probe will indicate when the pulse occurred. Put the logic probe on PB7, pin 16 of the header, and POKE a 0 to 37141.

Timer #2

Timer #2 works somewhat differently than timer #1. If bit 5 of the ACR is a 0, the timer acts as an interval timer just as timer #1 did in the one-shot

mode. The initial count comes from the timer #2 low and high byte registers. Bit 5 in the IFR is set by the timeout condition on timer #2. An interrupt can be enabled, but no provision has been made to have timer #2 control any of the output lines.

If bit 5 of the ACR is a 1, timer #2 is controlled by TTL pulses on PB6 rather than by the MPU clock. For timing long intervals, the output of timer #1 can be used as the input to timer #2 by simply connecting PB7 to PB6. Intervals of up to 150 minutes are then possible.

Shift Register

The 6522 has the provision to do input/output operations in a serial fashion via its shift register. Bits 2–4 of the ACR control the operation of the shift register. Data can be shifted out of the shift register, 1 bit at a time, over the CB2 pin. Similarly, data can be shifted into the shift register via the same pin. The timing can be controlled by either the MPU clock, timer #2, or by external clock pulses. The IEEE bus uses this shift register to send the serial data to the disk and the printer.

The Control Lines: CB1 and CB2

Finally, the user port supports two control lines, CB1 and CB2. CB2 is controlled by bits 5–7 of the peripheral control register (PCR). CB2 can be either an input or an output, as shown in Table 3–5. In the interrupt input mode, the CB2 flag of the IFR (bit 3) is set when a high-to-low transition occurs on CB2. In the input mode, the flag is set on a low-to-high transition. The flag is cleared automatically by a read to port B unless the independent mode is chosen. In the latter case, the flag must be cleared by setting the bit low with a POKE to the IFR.

CB2 has four possible output modes. In the handshake mode, CB2 falls low on a write to port B and is reset when there is an active transition on CB1. In this mode, CB2 acts as a "data available" output, and CB1 acts as a "data received" input. In the pulse output mode, CB2 goes low for one MPU cycle after a write to port B. In the two manual modes, CB2 is simply held high or low.

Bit 4 of the PCR controls the input line, CB1. If this bit is a 0, the CB1 flag (bit 4 of the IFR) is set on a high-to-low transition. If the bit is a 1, the flag is set on a low-to-high transition. Bits 0–3 control CA2 and CA1, which are not available on the user port. Their operation is identical, however, to that of CB1 and CB2, except that they refer to Port A rather than Port B.

TABLE 3-5: Peripheral Control Register Bit Functions

BIT #								
7	6	5	4	3	2	1	0	FUNCTION
0	0	0						CB2 interrupt input mode
0	0	1						CB2 independent interrupt input mode
0	1	0						CB2 input mode
0	1	1						CB2 independent input mode
1	0	0						CB2 handshake output mode
1	0	1						CB2 pulse output mode
1	1	0						CB2 manual output low
1	1	1						CB2 manual output high
			0					CB1 high-to-low
			1					CB1 low-to-high
				0	0	0		CA2 interrupt input mode
				0	0	1		CA2 independent interrupt input mode
				0	1	0		CA2 input mode
				0	1	1		CA2 independent input mode
				1	0	0		CA2 handshake output mode
				1	0	1		CA2 pulse output mode
				1	1	0		CA2 manual output mode
				1	1	1		CA2 manual output high
							0	CA1 high-to-low
							1	CA1 low-to-high

I/O Conditioning

Many computer control applications using 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, and so on. 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 6522 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 6522. We have, therefore, provided the following series of simple interface circuits that will help solve such problems.

DC Loads

Figure 3-9 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 voltage 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 add the 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 wish 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

FIGURE 3-9a.
An output from the user port can switch DC loads of up to one-half ampere and 40 volts with this circuit.

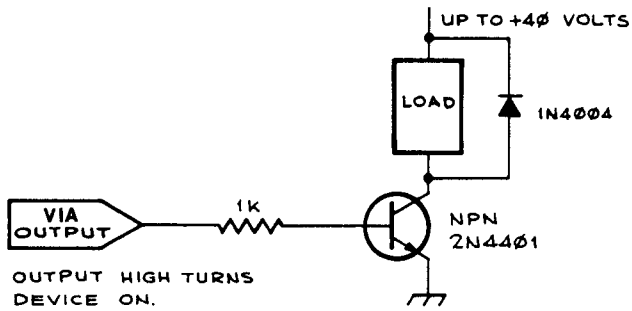
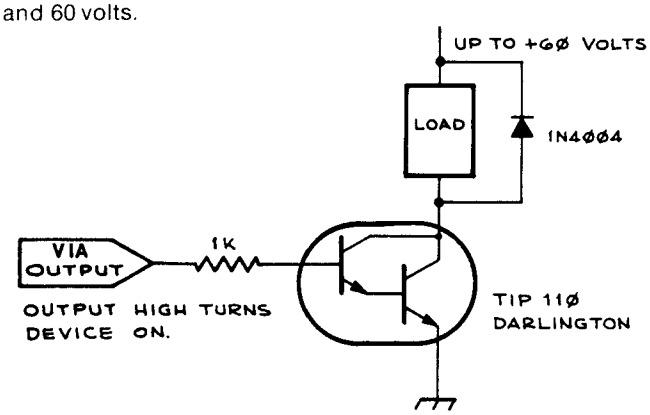


FIGURE 3-9b.
This circuit will handle loads up to 1 ampere and 60 volts.



you could reduce 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 3–10 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 voltages found on the logic boards, the 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 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 light-emitting diode 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 because there are no moving parts. The top panel of Figure 3–11 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 costs 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

FIGURE 3–10
Sensitive relay controlled by 1 output bit 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 the 110 VAC.

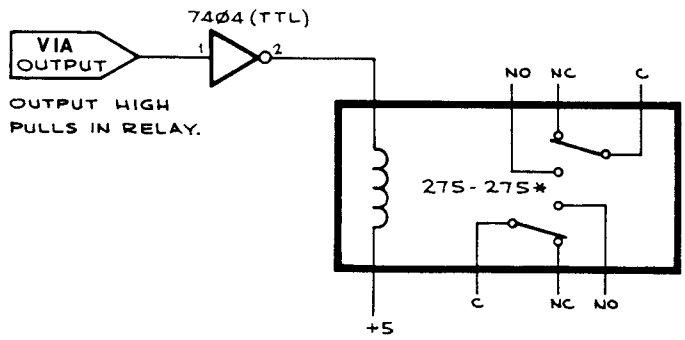
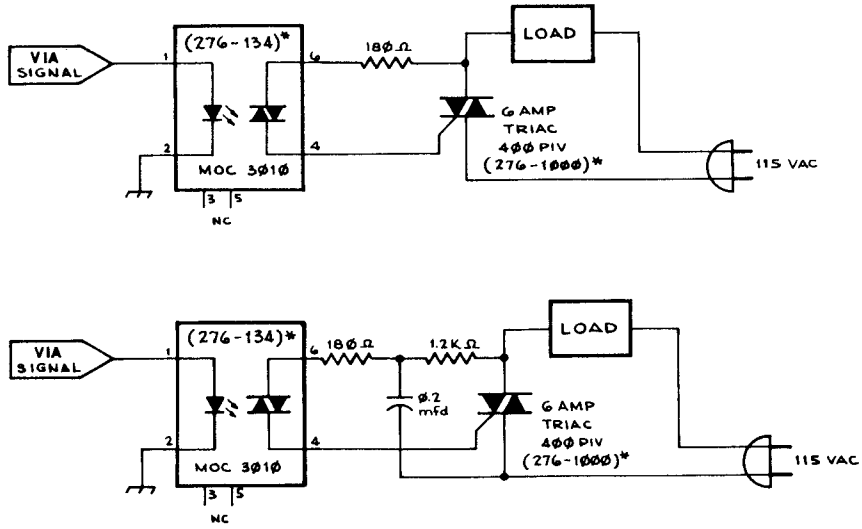


FIGURE 3-11
One output bit from the user port can switch up to 6 amperes of 110 VAC. The numbers with asterisks refer to Radio Shack part numbers. The upper circuit is for resistive loads while the lower circuit is designed to switch inductive loads, such as transformers and motors.



and cost between $10 and $20. They are sold at most industrial electronics supply houses.

Sensing Low-Level and Bipolar Inputs

Many devices we wish to interface do not output TTL logic levels. If we wish 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 6522 input, a voltage comparator IC will usually solve the problem. Figure 3-12 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 only 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 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.

FIGURE 3-12
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.

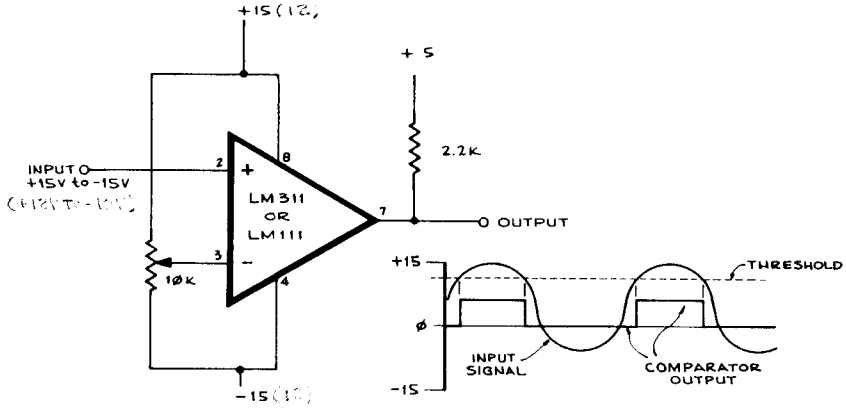
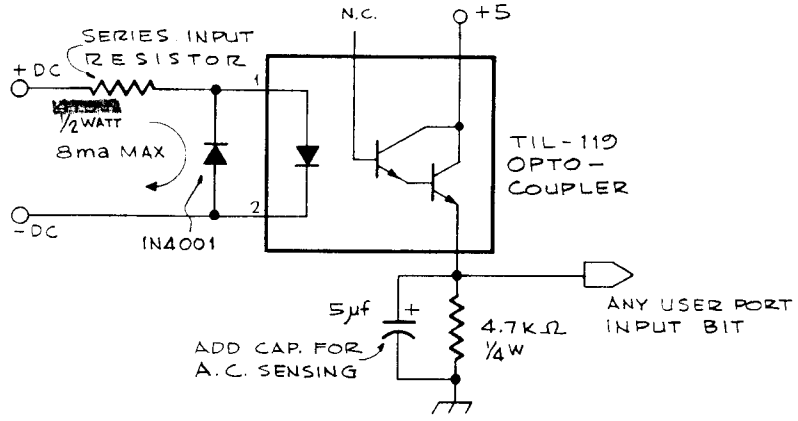


FIGURE 3-13
A coupler for sensing high voltages. The series input resistor should be adjusted to limit the current (see text). The 470-ohm resistor will handle voltages from 5 to 40 VDC. The opto-isolator gives excellent isolation between the source and the computer.



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 6522 as well as the entire VIC 20 must be adequately protected. To accomplish this, we recommend the use of an optocoupler. Figure 3-13 shows a circuit that will handle inputs between +5 to +40 volts DC. Several hundred volts can be accommodated so long as the series input resistor is chosen to limit the maximum current to 8ma (resistance = maximum voltage expected/.008). AC signals can be sensed if the 5 mfd capacitor is connected as shown.

4

SPEECH SYNTHESIS

The VIC 20 has one of the more sophisticated sound generators around built right into it. Although the VIC 20's generator is capable of four-part harmony, it still is not capable of producing human speech. Speech capability can be added to the VIC 20, however, with the simple project described below, which plugs into the VIC 20's user port. In this project, National Semiconductor's DIGITALKER* is interfaced to the VIC 20 to make an inexpensive and easily programmed speech synthesizer.

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 really the building blocks of the spoken word. By linking the proper sequence of phonemes together, 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.

*DIGITALKER is a registered trademark of National Semiconductor.

The shortcoming is that speech quality is usually poor and programming is very 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 developed a simple and inexpensive speech synthesizer system called the DIGITALKER. Although it uses the digitized recording approach, they have solved the problem of programming the speech information by offering a three-chip set that includes the speech processor chip (SPC) and two 8K read-only memory (ROM) chips that are preprogrammed with a 143-word standard vocabulary (listed in Figure 4-1). At the time of this writing, the DT 1050 chip set is available for $34.95*. Because of these features, we chose the DIGITALKER for this project.

How It Works

Figure 4-2 shows a block diagram of the DIGITALKER. The code for the word to be spoken must be determined from the master word list in Figure 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 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 where it 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 pinouts of the two ROMs and the speech processor chip appear at the bottom of the figure.

The VIC 20 Interface

Figures 4-3 to 4-5 show the schematic for the DIGITALKER project. We built the circuit on a 4" x 4" experimenter card with holes spaced on 0.1" 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

*Jameco Electronics, 1355 Shoreway Rd., Belmont, CA 74002. Also, Time Domain Systems, 840 Sandy Cove, Rodeo, CA 94572 (For the speech ROMs only).

FIGURE 4-1
Master word list for the Digitalker. When the binary number at the right is POKEd to the Digitalker at 37136 decimal, the word at the left is spoken (National Semiconductor Corp.).

Word	8-Bit Binary Address SW8 SW1		8-Bit Binary Address SW8 SW1		8-Bit Binary Address SW8 SW1
THIS IS DIGITALKER	00000000	Q	00110000	IS	01100000
ONE	00000001	R	00110001	IT	01100001
TWO	00000010	S	00110010	KILO	01100010
THREE	00000011	T	00110011	LEFT	01100011
FOUR	00000100	U	W0110100	LESS	01100100
FIVE	00000101	V	00110101	LESSER	01100101
SIX	00000110	W	00110110	LIMIT	01100110
SEVEN	00000111	X	00110111	LOW	01100111
EIGHT	00001000	Y	00111000	LOWER	01101000
NINE	00001001	Z	00111001	MARK	01101001
TEN	00001010	AGAIN	00111010	METER	01101010
ELEVEN	00001011	AMPERE	00111011	MILE	01101011
TWELVE	00001100	AND	00111100	MILLI	01101100
THIRTEEN	00001101	AT	00111101	MINUS	01101101
FOURTEEN	00001110	CANCEL	00111110	MINUTE	01101110
FIFTEEN	00001111	CASE	00111111	NEAR	01101111
SIXTEEN	00010000	CENT	01000000	NUMBER	01110000
SEVENTEEN	00010001	400HERTZ TONE	01000001	OF	01110001
EIGHTEEN	00010010	80HERTZ TONE	01000010	OFF	01110010
NINETEEN	00010011	20MS SILENCE	01000011	ON	01110011
TWENTY	00010100	40MS SILENCE	01000100	OUT	01110100
THIRTY	00010101	80MS SILENCE	01000101	OVER	01110101
FORTY	00010110	160MS SILENCE	01000110	PARENTHESIS	01110110
FIFTY	00010111	320MS SILENCE	01000111	PERCENT	01110111
SIXTY	00011000	CENTI	01001000	PLEASE	01111000
SEVENTY	00011001	CHECK	01001001	PLUS	01111001
EIGHTY	00011010	COMMA	01001010	POINT	01111010
NINETY	00011011	CONTROL	01001011	POUND	01111011
HUNDRED	00011100	DANGER	01001100	PULSES	01111100
THOUSAND	00011101	DEGREE	01001101	RATE	01111101
MILLION	00011110	DOLLAR	01001110	RE	01111110
ZERO	00011111	DOWN	01001111	READY	01111111
A	00100000	EQUAL	01010000	RIGHT	10000000
B	00100001	ERROR	01010001	SS (Note 1)	10000001
C	00100010	FEET	01010010	SECOND	10000010
D	00100011	FLOW	01010011	SET	10000011
E	00100100	FUEL	01010100	SPACE	10000100
F	00100101	GALLON	01010101	SPEED	10000101
G	00100110	GO	01010110	STAR	10000110
H	00100111	GRAM	01010111	START	10000111
I	00101000	GREAT	01011000	STOP	10001000
J	00101001	GREATER	01011001	THAN	10001001
K	00101010	HAVE	01011010	THE	10001010
L	00101011	HIGH	01011011	TIME	10001011
M	00101100	HIGHER	01011100	TRY	10001100
N	00101101	HOUR	01011101	UP	10001101
O	00101110	IN	01011110	VOLT	10001110
P	00101111	INCHES	01011111	WEIGHT (Note 2)	10001111

Note 1: "SS" makes any singular word plural
Note 2: Address 143 is the last legal address in this particular word list. Exceeding address 143 will produce pieces of unintelligible invalid speech data.

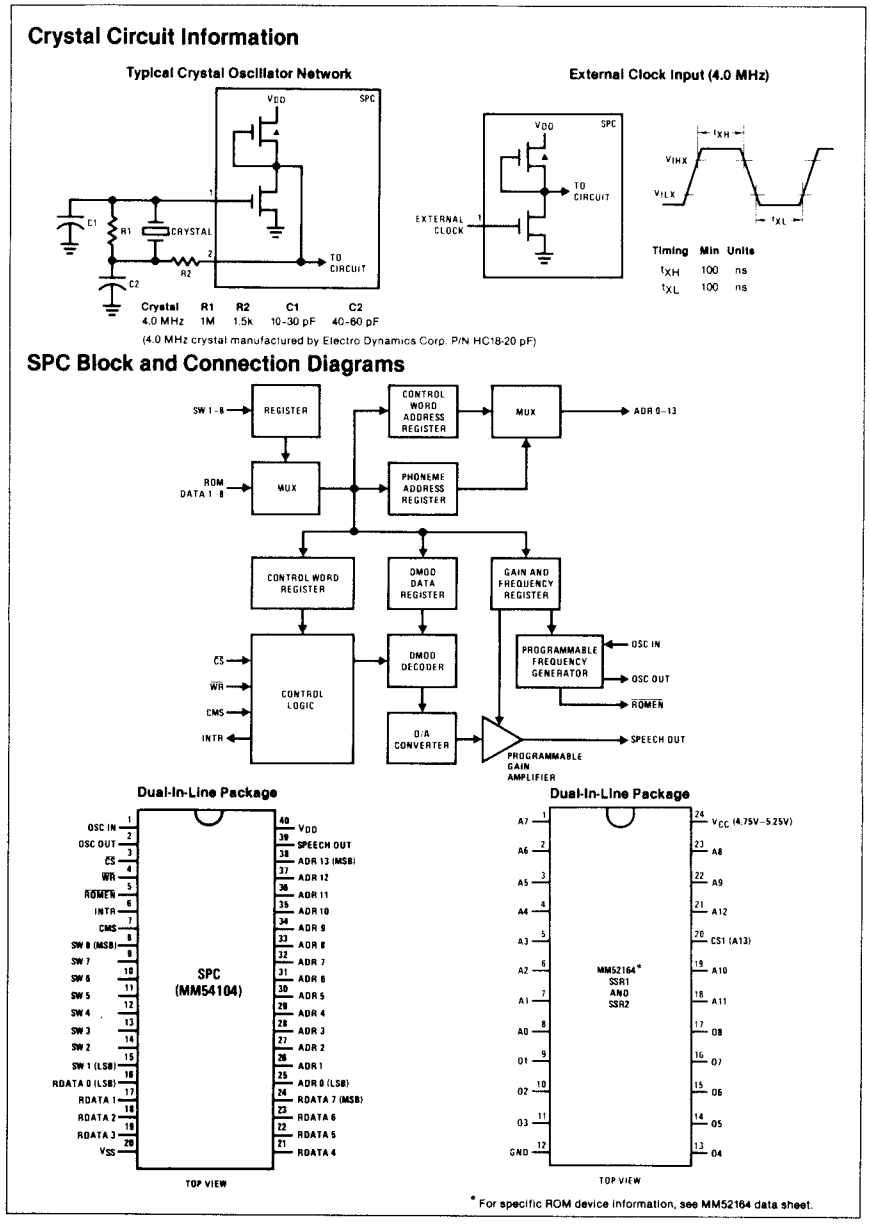
below the board. The board was cut as described in Chapter 8 and a 12/24 pin edge connector was soldered to the card.

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

Checkout

When all of the connections are completed and your wiring has been double-checked, plug the board into the user port. *Do not insert any ICs*

FIGURE 4-2
Block diagram and pinouts for the Speech Processor Chip (SPC) and the speech ROMS (National Semiconductor Corp.).



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 VIC 20's power and confirm that the VIC 20 is operating properly. If not, look for shorts on either the +5 volt or the nine VAC connections. Next, verify that you have +5 volts on pin 24 of both ROM sockets. You should also

FIGURE 4-3
Power supply (top) and audio amplifier (bottom) for the speech synthesizer.

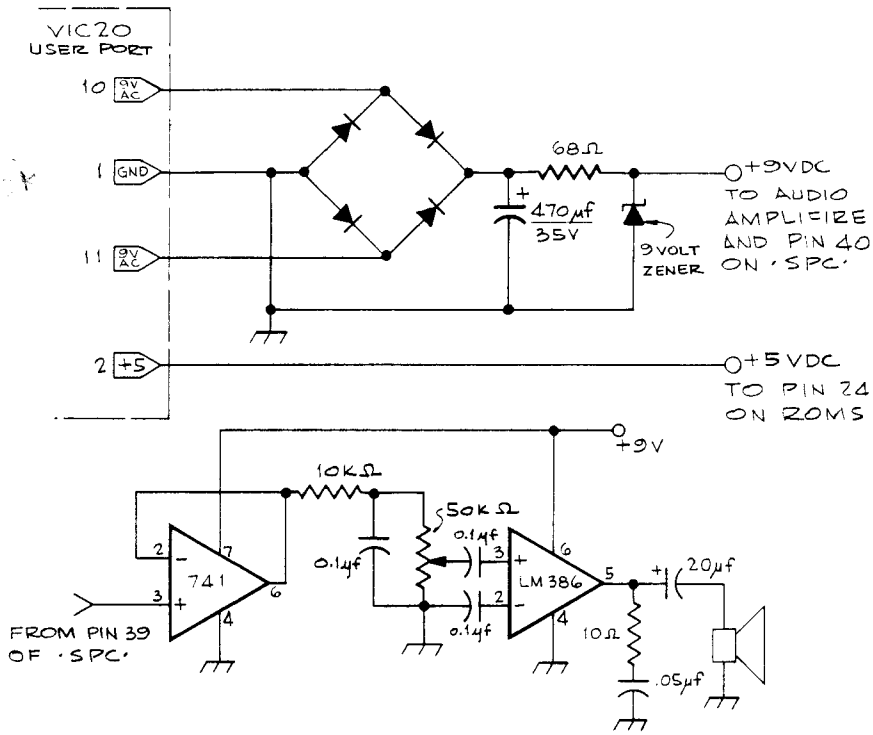


FIGURE 4-4
Connections between the SPC and the VIC 20's user port.

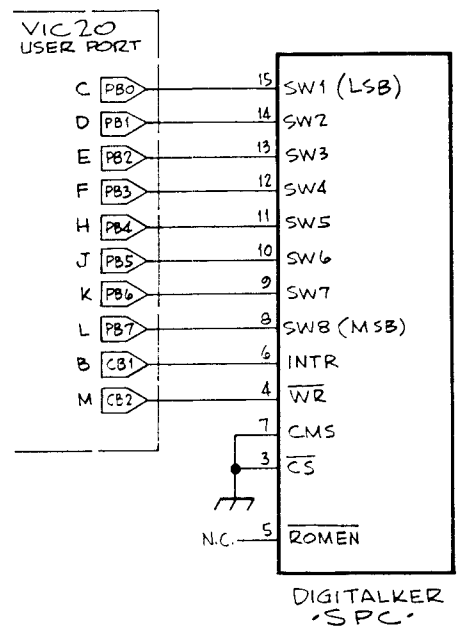


FIGURE 4-5
Interconnections between the SPC, the ROMs, and the crystal.

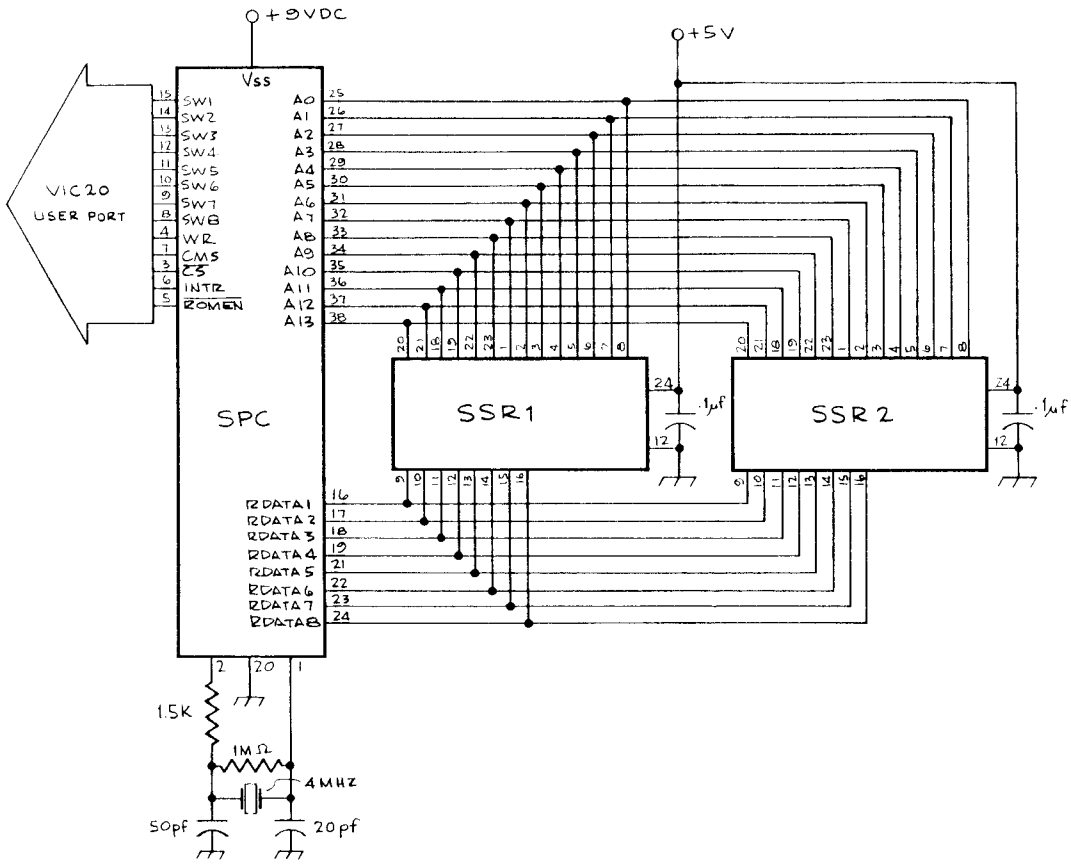
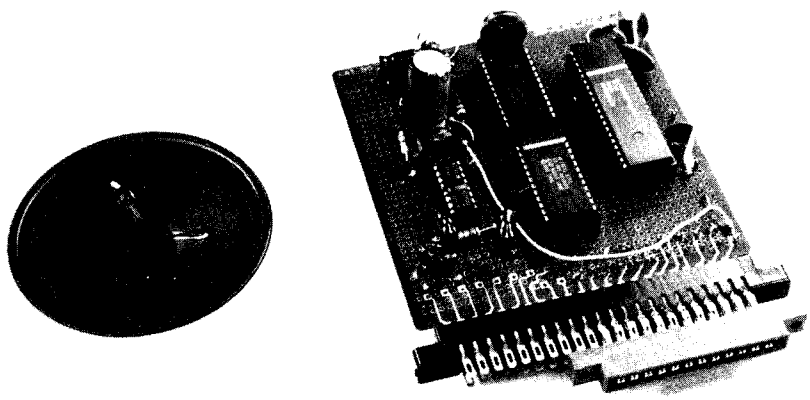


FIGURE 4-6
Photo of the finished Digitalker board.



PARTS LIST – DIGITALKER

Quantity	
1	4" x 4" experimenters board (Radio Shack #276-156)
1	DT 1050 DIGITALKER chip set
1	LM 741 OPAMP
1	LM 386 power amp
1	50 PIV-lamp bridge rectifier
1	1000 μF/35V electrolytic
1	20 μ/35V electrolytic
1	9v 1 watt zenner diode
6	.1 μF/35V Capacitors
1	.05 μF/35V Capacitors
2	24-Pin Wire-Wrap DIP Sockets
1	40-Pin Wire-Wrap DIP Sockets
2	8-Pin Wire-Wrap DIP Sockets
1	4 mHz Crystal
1	50 Pf Capacitor
1	20 Pf Capacitor
1	1 Meg ¼ Watt Resistor
1	68 ohm ¼ Watt Resistor
1	10 K ¼ Watt Resistor
1	10 ohm ¼ Watt Resistor
1	8 ohm 3" speaker
1	50 K Trimpot

see +9 volts on pin 40 of the SPC socket, pin 7 of the 741 socket, and pin 6 of the LM386 socket. Now turn off the VIC 20. Set the volt-ohm meter to ohms x 1. Zero ohms should be observed when the free probe is touched to pin 12 of the ROM sockets; pins 3, 7, and 20 of the SPC socket; and pin 4 of both the 741 socket and the LM 386 socket.

When all these voltages have been verified, 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 trim pot to a middle position and turn the VIC 20 back on. You probably heard a word or phrase spoken at this time. Now enter the following program.

```
10  POKE 37148, 254
20  POKE 37138, 255
30  POKE 37136, J
40  POKE 37148, 192
50  POKE 37148, 254
```

```
60  FOR I = 1 TO 2000: NEXT I
70  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 VIC 20 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 60-cycle hum in the speaker. If that does not occur, look for bad wiring or components in the audio amplifier section.

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 sets bits 5–7 of the peripheral control register high to raise CB2, the WR signal to the SPC. Line 20 sets the data direction register for port B to all outputs. In line 30 we output a 0 through port B to the SW lines. Since J has a default value of 0 in VIC 20's BASIC, all of the SW lines will be held low. In line 40 we lower CB2 to latch the SW data, and in 50 we raise CB2 to start the speech sequence. In line 60 we delay for several seconds, and in 70 the program goes back to repeat the sequence.

It is possible to have the VIC 20 actually sense when the word is finished rather than to simply delay. This will allow you to send words of varying length with proper timing between phrases. To accomplish this, replace line 60 in the program with the following command:

```
60 IF (16 AND PEEK (37149)) = 0 THEN 60
```

Now bit 4 of the interrupt flag register will be tested to see if CB1 is low. If it is low, the word is not finished and the program will loop back and test it again and again until CB1 is returned to the high state on completion of the phrase. Run the program and verify that the words now repeat themselves with no pause between the repetitions. Lines 30–60 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 list. 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 have said "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
70 NEXT J
```

Making Intelligent Phrases

The usefulness of DIGITALKER lies in its programmability so that the 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 representing the ones and requiring no suffix, the next three digits representing the thousands and using the suffix "thousand." 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 1-9, followed by the word "hundred." The middle digit is a little more complicated. If it is 2-9, 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 4-1 codes the flow chart in Figure 4-7 into a BASIC subroutine starting at lines 1000. The subroutine at 1100 arranges the words for a three-digit number and the subroutine at 1200 sends the word code 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. 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

LISTING 4-1
Speak a number.

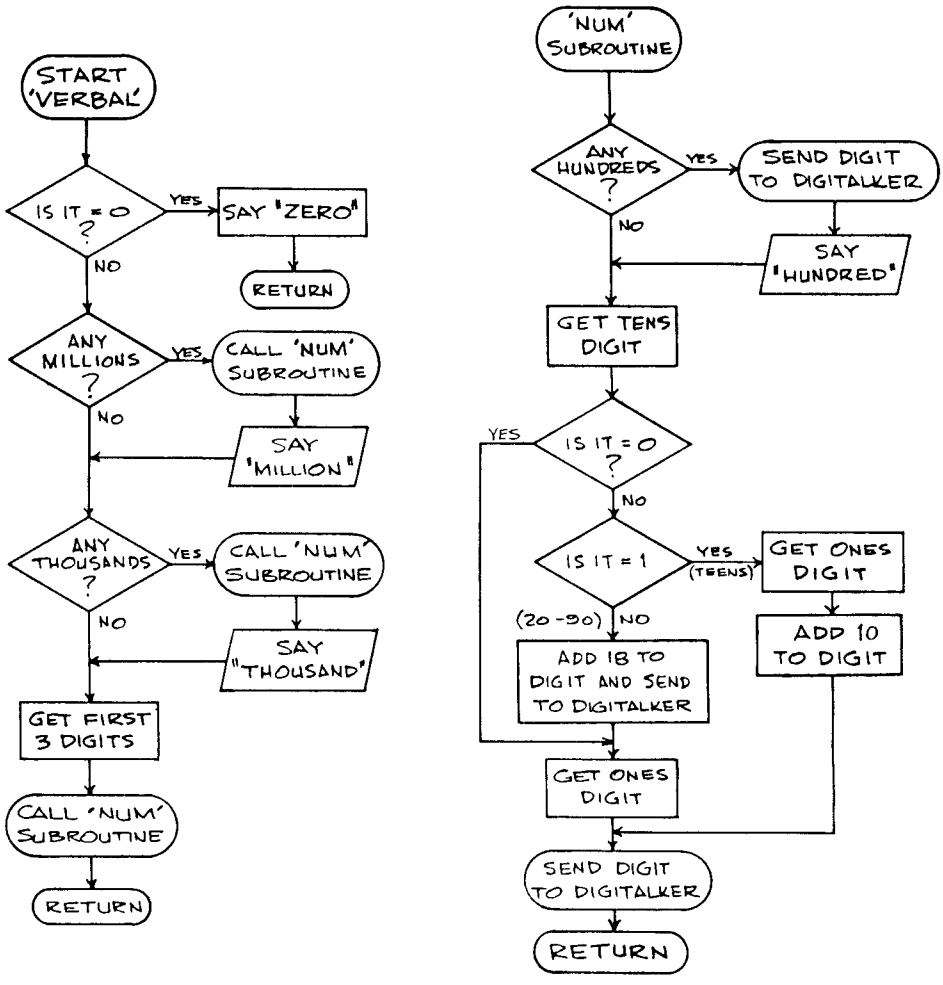
```
1 REM SPEAK A NUMBER UP TO 9 DIGITS
2 POKE 37138,255
3 POKE 37148, 254
10 INPUT Y
20 GOSUB 1000
30 GOTO 10
1000 IF Y=0 THEN Z7=31:GOSUB 1200:RETURN
1005 Z6=INT(Y/1000000)
1010 IF Z6>999 THEN RETURN
1020 IF Z6<>0THEN Z8=Z6:GOSUB 1100:Z7=30:GOSUB1200
1030 Z5=Y-Z6*1000000:Z5=INT(Z5/1000)
1040 IF Z5<>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+18:GOSUB 1200
1170 IF Z1<>0 THEN Z7=Z1:GOSUB1200
1180 RETURN
1190 Z7=Z1+10:GOSUB 1200:RETURN
1200 POKE 37136,Z7
1210 POKE 37148,192
1220 POKE 37148,254
1230 IF INT( PEEK(37149)AND 16) =0 THEN 1220
1240 RETURN
```

LISTING 4-2
Talking lunar lander.

```
1 REM TALKING LUNAR LANDER
2 PRINT"␣ENTER ROCKET BURNS SO THAT YOU LAND "
3 PRINT"AT 5 F/S OR LESS."
4 PRINT"HIT RETURN TO START":INPUT A$
9 POKE 37138,255
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=107:GOSUB1200:Z7=32:GOSUB1200:Z7=93
430 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+18:GOSUB 1200
1170 IF Z1<>0 THEN Z7=Z1:GOSUB1200
1180 RETURN
1190 Z7=Z1+10:GOSUB 1200:RETURN
1200 POKE 37136,Z7
1210 POKE 37148,192
1220 POKE 37148,254
1230 IF INT( PEEK(37149)AND 16) =0 THEN 1220
1240 RETURN
```

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. NUN is a subroutine that says a three-digit number and is called by VERBAL.



from National Semiconductor with specialized vocabularies. Furthermore, you can program your own ROMs using National Semiconductor's development kit DTSW500. This consists of a CPM 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 CPM based system having an 8" disk drive and a ROM burner will allow you to create your own custom vocabulary of up to 256 words for a specific DIGITALKER application.

5

MECHANICAL ACTUATORS

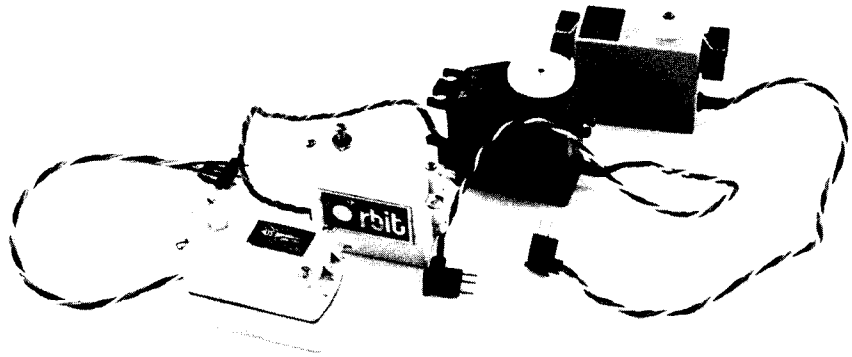
In certain applications, you will want to computer control a mechanical device. Computer controlled mechanical actuators have been around almost as long as computers themselves. Digital x-y plotters and the head positioner on a disk drive are only two examples of computer controlled mechanical devices. More recently, the growing field of robotics has sparked a new interest in mechanical actuators among industrial designers and hobbyists alike. In this chapter, we would like to show you how the VIC 20 can be interfaced and programmed to control both an 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, and a high degree of resolution with 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

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



controlled by a pulse width detector inside the case. The circuit determines the duration of a positive-going TTL pulse. It then determines the position of the shaft by examining the voltage on a potentiometer, which 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 VIC 20 is ideally suited for a servo because you can use the 6522 VIA chip's built-in timer, which is available on the user port. In the one-shot mode, the timer causes output PB7 to go low for a preset period of time and then return to the high state. The only hardware required is

one inverter to convert the VIC 20's low-going pulse to a high-going pulse as shown in Figure 5-2.

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 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 the output of the inverter. The hobby shop will be able to supply you with a miniature socket to match the plug on the servo.

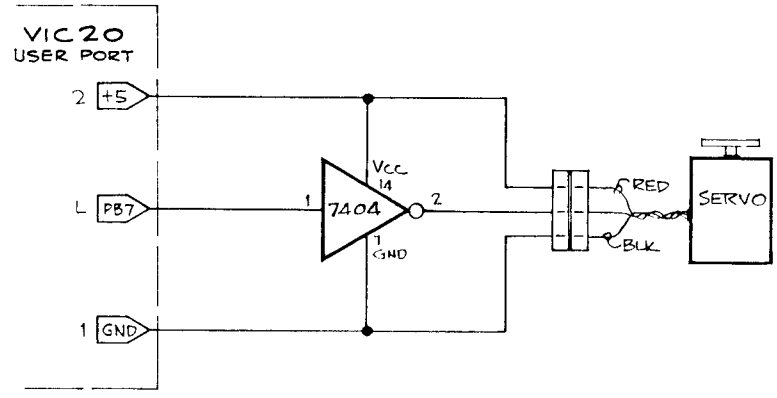
The Software

You can control the servo entirely from BASIC. Remember, however, that the servo 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 game paddle.

```
10 POKE 37147,128
20 P=PEEK (36872)*4 + 900
30 H=INT(P/256)
40 L=P-H*256
50 POKE 37140,L
60 POKE 37141,H
70 GOTO 20
```

Line 10 of the program puts 128 into the auxiliary control register of the 6522 VIA chip. This enables the one-shot mode. Line 20 gets the position of the game paddle and scales this value into the appropriate number of microseconds required for the pulse duration. Lines 30 and 40 break this

FIGURE 5-2
Inverter circuit for the single servo program.



number down into two bytes, H and L. H is the high-order 8 bits of this number, obtained simply by dividing by 256 and converting to an integer. The remainder, L, is the low-order 8 bits. These are put into the timer #1 low byte and timer #1 high byte and counter register by lines 50 and 60. Insertion into the high byte and counter register should be done last because that step starts the timer. PB7 will then go low for as many microseconds as were indicated by the 2-byte number that was placed in the timer register. Line 70 directs the program to repeat the loop so that the entire process will be repeated over and over again.

Using the Servo Under Interrupt Control

The BASIC program above 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 VIC 20. Every 60th of a second, the 6502 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 address $00FB (251 decimal). $FB is an unused location on page zero in the VIC 20. 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 BASIC. The pulse-generator program appears in Listing 5-1.

LISTING 5-1
Pulse generator program.

```
:ASM
                1000 * PULSE-GENERATOR PROGRAM
                1010         .OR $1800     PROGRAM START ADDRESS
                1020         .TA $0800
                1030 *--------------------------------
                1040 * EQUATES:
0314-           1050 IRQVEC .EQ $0314 ← ADDRESS OF IRQ SERVICE ROUTINE
9114-           1060 TMR1.L .EQ $9114     TIMER 1 LOW BYTE
9115-           1070 TMR1.H .EQ $9115     TIMER 1 HIGH BYTE
EABF-           1080 OLDIRQ .EQ $EABF ← OLD IRQ ADDRESS
                1090 *--------------------------------
1800- 08        1100 SINT   PHP            SAVE PROCESSOR STATUS
1801- 78        1110        SEI            NO INTERRUPTS FOR A WHILE
1802- A9 0F     1120        LDA #START     LOW BYTE OF OUR START
1804- 8D 14 03  1130        STA IRQVEC     NEW ADDRESS
1807- A9 18     1140        LDA /START     HIGH BYTE
```

```
1809- 8D 15 03 1150        STA IRQVEC+1
180C- 28       1160        PLP
180D- 58       1170        CLI            ALLOW INTERRUPTS
180E- 60       1180        RTS
               1190 *-------------------------------
180F- 48       1200 START  PHA            SAVE ACC.
1810- A5 FD    1210        LDA $FD        GET A ZERO
1812- 85 FC    1220        STA $FC        PUT INTO $FC
1814- A5 FB    1230        LDA $FB
1816- 18       1240        CLC
1817- 0A       1250        ASL            GET BIT 7
1818- 26 FC    1260        ROL $FC        ROTATE INTO $FC
181A- 0A       1270        ASL            ANOTHER BIT
181B- 26 FC    1280        ROL $FC
181D- 0A       1290        ASL            ANOTHER
181E- 26 FC    1300        ROL $FC
1820- 8D 14 91 1310        STA TMR1.L     START TIMER
1823- A5 FC    1320        LDA $FC
1825- 8D 15 91 1330        STA TMR1.H     TIMER HIGH BYTE
1828- 68       1340        PLA            RESTORE ACC.
1829- 4C BF EA 1350        JMP OLDIRQ     DO NORMAL IRQ SERVICE
```

A BASIC program that locates the pulse-generating code in high memory appears in Listing 5-2. That program, on startup, finds the top of memory and moves the top of memory pointer down to make room for the routine. It then does a SYS to the routine, which changes the interrupt vector. Thereafter, the servo is automatically positioned according to the byte stored at address 251, even if BASIC is not running.

LISTING 5-2
BASIC version of pulse program.

```
1 REM SET UP 6522
5 POKE 37138,255
10 POKE 37147,128
15 POKE 253,0
18 REM FIND MEMTOP
20 H=PEEK(56)
30 L=PEEK(55)
40 MT=256*H+L
50 PRINT MT
55 REM MOVE MEMTOP DOWN
60 MT=MT-45
70 H=INT (MT/256)
80 L=MT-256*H
90 POKE 55,L
100 POKE 56,H
110 MT=MT+1
115 REM POKE IN PROGRAM
120 FOR I=MT TO MT+43
130 READ X
140 POKE I,X
150 NEXT I
160 DATA 8,120,169,15,141,20,3,169,24,141,21
165 DATA 3,40,88,96,72,165,253,133,252,165
170 DATA 251,24,10,38,252,10,38,252,10,38
175 DATA 252,141,20,145,165,252,141,21,145,104
180 DATA 76,191,234
190 REM CALCULATE NEW IR VECTOR
200 IR=MT+15
210 H=INT (IR/256)
220 L=IR-H*256
230 POKE MT+3,L
240 POKE MT+8,H
245 REM CHANGE IR
250 SYS MT
```

This mode of operation can be cancelled at any time by simply resetting the VIC 20, which restores the interrupt vector but does not delete the BASIC program.

Multiple Servos

Unfortunately, only one timer is available to the user in the VIC 20. Thus, another approach must be taken if more than one servo is to be interfaced. To get around that problem, Listing 5-3 presents a machine language program that generates four independently controlled pulses on PB0 through PB3 (pins C-F on the user port connector). Furthermore, these pulses are positive-going so that the pulse inverter is not required as was the case in the previous example. Simply connect one servo input directly to each of these outputs. Be sure not to forget the +5 volt and the ground leads from each of the servos. This program runs in the interrupt mode so that its operation is again transparent to BASIC.

LISTING 5-3 4-servo program.

```
:ASM
                1000 * SERVO 4
                1010         .OR $0340
                1020         .TA $0840
                1030 *---------------------------------
                1040 * EQUATES:
0314-           1050 IRQVEC .EQ $0314 ---- ADDRESS OF IRQ SERVICE ROUTINE
9110-           1060 PORTB  .EQ $9110     USER PORT
EABF-           1070 OLDIRQ .EQ $EABF ---- OLD IRQ ADDRESS
                1080 *---------------------------------
0340- 08        1090 START  PHP            SAVE IRQ STATUS
0341- 78        1100        SEI
0342- A9 50     1110        LDA #ENTER     ADL OF ENTER
0344- 8D 14 03  1120        STA IRQVEC     CHANGE IRQ VECTOR
0347- A9 03     1130   *    LDA /ENTER     ADH OF ENTER
0349- 8D 15 03  1140        STA IRQVEC+1
034C- 28        1150        PLP            RESTORE IRQ STATUS
034D- 58        1160        CLI            ENABLE IRQ'S
034E- 60        1170        RTS
034F- EA        1180        NOP
                1190 *---------------------------------
0350- 08        1200 ENTER  PHP            SAVE ALL REGISTERS
0351- 48        1210        PHA
0352- 8A        1220        TXA
0353- 48        1230   *    PHA
0354- 98        1240        TYA
0355- 48        1250        PHA
0356- A9 FF     1260        LDA #$FF
0358- 8D 93 03  1270        STA MASK       START VALUE FOR MASK
035B- 8D 10 91  1280        STA PORTB      ALL OUTPUTS HIGH
035E- A0 10     1290        LDY #$10       USE Y AS COUNTER
0360- A2 04     1300 LOOP3  LDX #$04       USE X AS CHANNEL#
0362- 98        1310 LOOP2  TYA
0363- DD 97 03  1320        CMP CONT-1,X
0366- B0 20     1330        BCS WASTE      TIME OUT ?
0368- AD 93 03  1340        LDA MASK
036B- 3D 93 03  1350        AND TABLE-1,X
036E- 8D 10 91  1360        STA PORTB      CLEAR A BIT OF PORT
0371- 8D 93 03  1370        STA MASK       SAVE MASK
```

```
0374- CA       1390 MORE   DEX
0375- D0 EB    1400        BNE LOOP2     ALL CHANNELS CHECKED ?
0377- 88       1410        DEY           FALL THRU IF YES
0378- D0 E6    1420        BNE LOOP3     COUNTDOWN OVER ?
037A- A9 00    1430        LDA #$00
037C- 8D 10 91 1440        STA PORTB     ALL PORT BITS LOW
037F- 68       1450        PLA           RESTORE ALL REGS.
0380- A8       1460        TAY
0381- 68       1470        PLA
0382- AA       1480        TAX
0383- 68       1490        PLA
0384- 28       1500        PLP
0385- 4C BF EA 1510        JMP OLDIRQ    DO NORMAL IRQ SERVICE
0388- C6 FB    1520 WASTE  DEC $FB       WASTE 13 CYCLES
038A- AD 50 03 1530        LDA ENTER     TIME WASTERS
038D- AD 50 03 1540        LDA ENTER
0390- 4C 74 03 1550        JMP MORE      GO CHECK AGAIN
0393- 00       1560 MASK   .HS 00        TEMP. SPACE FOR MASK
0394- FE FD FB
0397- F7       1570 TABLE  .HS FEFDFBF7 4 BYTES, EACH WITH DIFFERENT
               1580 *                    BIT LOW
0398- 00 00 00
039B- 00       1590 CONT   .HS 00000000 4 TEMPORARY BYTES
```

The four control registers are $0398-$039B (920-923 decimal). Because this code is not relocatable as the previous example was, we have located it in the cassette buffer. Therefore, be sure to reset the system with a RUN STOP, Control, and Reset all depressed simultaneously before attempting any cassette operations. Otherwise, a system crash will result. There is one important compromise in this approach. The 6502 is simply not fast enough to check all four channels very many times before 1.5 milliseconds has 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 155 increments achieved using the 6522's one-shot timer. For most applications, however, this will probably be a sufficient resolution. Listing 5-4 shows a BASIC program that puts the machine language program into the cassette buffer and resets the interrupt vector.

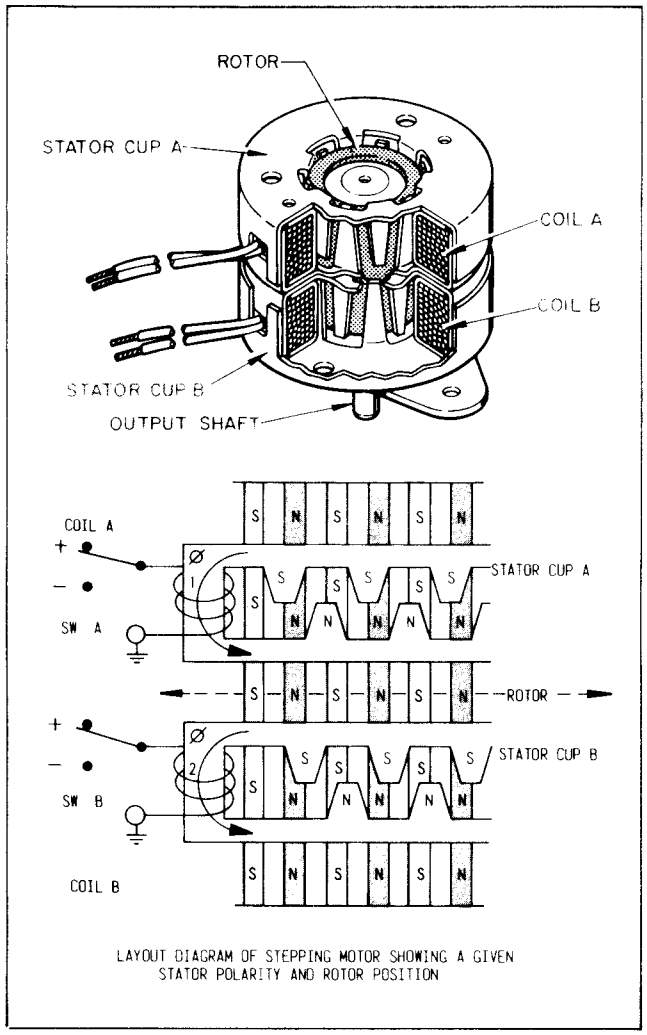
LISTING 5-4 BASIC version of 4-servo program.

```
10 DATA 8,120,169,80,141,20,3,169,3,141,21
20 DATA 3,40,88,96,234,8,72,138,72,152
30 DATA 72,169,255,141,147,3,141,16,145,160
40 DATA 16,162,4,152,205,151,3,176,32,173
50 DATA 147,3,45,147,3,141,16,145,141,147
60 DATA  3,202,208,235,136,208,230,169,0,141
70 DATA 16,145,104,168,104,170,104,40,76,191
80 DATA 234,198,251,173,80,3,173,80,3,76,116
90 DATA 3,234,254,253,251,247
95 POKE37138,255
100 FOR J=832 TO 919
110 READ K
120 POKE J,K
130 NEXT J
140 REM PROGRAM IN PLACE
150 SYS 832
160 REM SET INT VECTOR
170 REM START CODE HERE
```

Stepper Motors

By far the most common computer controlled mechanical actuator is the stepper motor. These low-cost motors can easily be interfaced to the VIC 20 and lend themselves to a variety of applications, such as robotics, digital plotting, and disk drive technology. The stepper 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 stepper motor. Permanent magnets on the moving rotor are surrounded by two stators, A and B. By controlling the direction of current through the

FIGURE 5-3
Cutaway view of a two-phase permanent magnet stepper motor (Airpax Corporation, Cheshire, CT.).



windings of the two stators, the rotor can be caused to move ¼ 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 VIC 20 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 VIC 20 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 stepper motor.

A Small Step for a Man

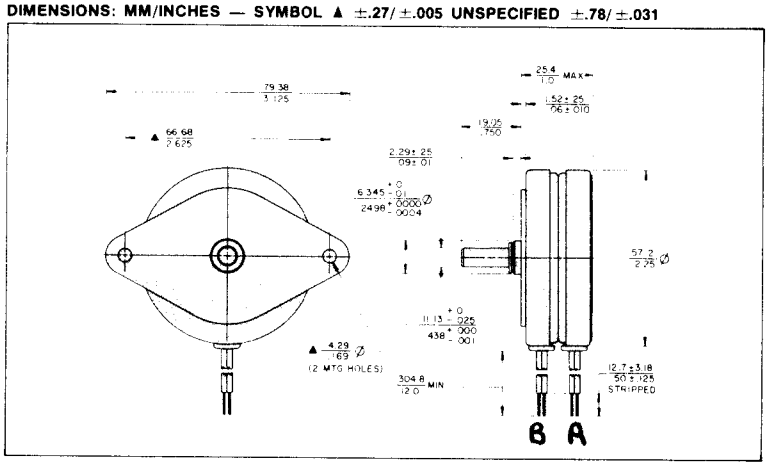
Stepper motors come in two varieties—unipolar and bipolar. In the unipolar models, the stator coils are centertapped. The polarity of each coil is switched by applying positive power to the centertap and closing a switch to ground at just one end of the coil. The polarity is reversed by opening that switch and closing the switch on the other end of the coil. A bipolar motor has no center tap, so a single-pole, double-throw 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, Connecticut 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

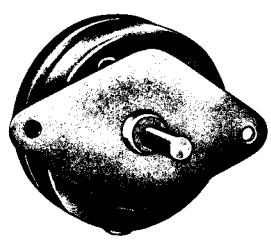
Figure 5-5 shows the interface for the motor. The 4 low-order bits of port B of the user port are connected to the four stator switches. A TTL inverting buffer is used for the connection. This serves two functions. First, it provides additional current drive to the VIC 20's control lines so that the transistor switches can be fully turned on. Second, since these lines assume a high state when the VIC 20 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.

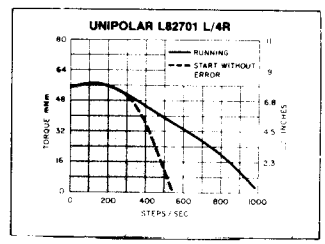
FIGURE 5-4
Technical data for the airpax K82701 unipolar stepper motor (Airpax Corporation, Cheshire, CT.).

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	3.1×10^{-3}				3.7×10^{-3}				3.1×10^{-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											

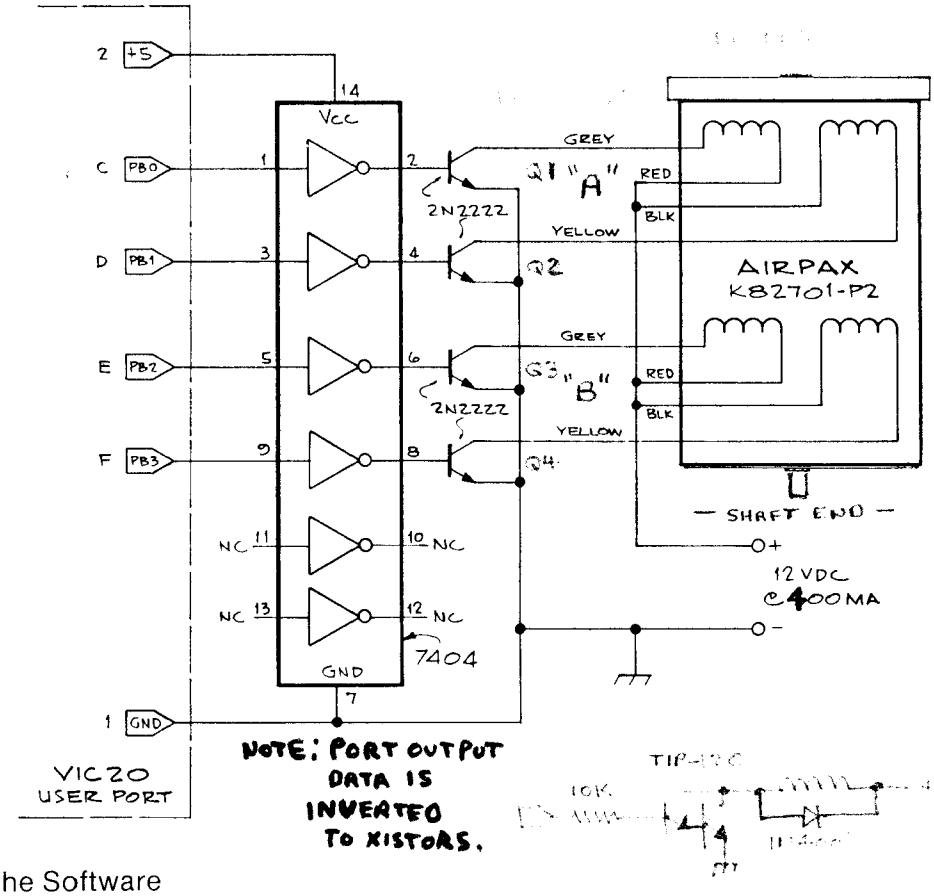




12VDC Power Supply

Since the 400 ma required by the stepping motor is a little more than can be stolen from the VIC 20'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.

FIGURE 5-5
Interface for the stepper motor.



The Software

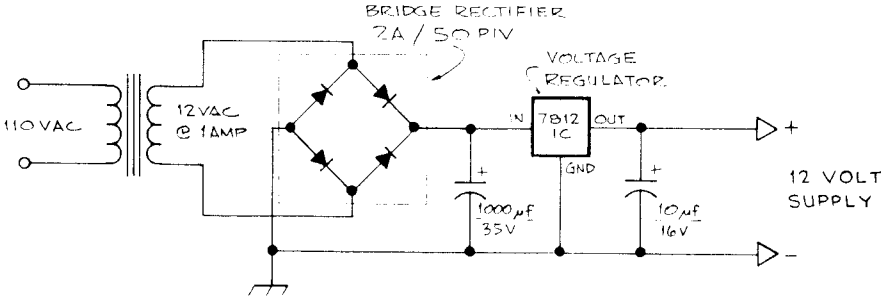
Rotation of the motor is accomplished by turning the four transistors off and on in the proper sequence.

TABLE 5-1

	DATA	STEP	Q1 (1)	Q2 (2)	Q3 (4)	Q4 (8)	
	10	1	ON	OFF	ON	OFF	
CW	6	2	ON	OFF	OFF	ON	CCW
Rotation	5	3	OFF	ON	OFF	ON	Rotation
↓	9	4	OFF	ON	ON	OFF	↑
	10	5	ON	OFF	ON	OFF	

Table 5-1 shows the switching sequence to move 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

FIGURE 5-6
A power supply to provide the high current required by the stepper motor.



generate those bit patterns when successively POKED to address 37136, the address of the interface.

Listing 5-5 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 6522 VIA'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.

Checkout

When you have constructed the interface, you will first want to test it. Plug the interface into the VIC 20 user port, apply the +12 volt power, and turn on the VIC 20. Verify that the VIC 20 has come up properly and that you have a cursor. If not, turn the VIC 20 off immediately and locate the problem. (Hint: Shorts between +5 and ground will blow the 3-

LISTING 5-5
Stepper controller program.

```
1 REM STEPPER MOTOR CONTROLER
10 DATA 10,6,5,9
20 FOR I=1 TO 4:READ Z(I):NEXTI
30 Z1=37136:Z2=1:Z3=5:Z4=0:Z5=4
40 POKE 37138,255
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
999 REM MOVE ONE STEP CW
1000 I=Z2+I:IF I=Z3 THEN I=Z2
1010 POKE Z1,Z(I):RETURN
1999 REM MOVE ONE STEP CCW
2000 I=I-Z2:IF I=Z4 THEN I=Z5
2010 POKE Z1,Z(I):RETURN
```

ampere fuse located inside the VIC 20.) 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 stepper 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 VIC 20 by cycling the power switch to avoid overheating the stepper 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 37136, 10
30 POKE 37136, 6
40 POKE 37136, 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.

LISTING 5-6,
Dual stepper controller program.

```
10 DATA 10,6,5,9
20 FOR I=1 TO 4
30 READ Z(I):NEXT I
40 POKE37138,255
50 Z1=37136:Z2=1:Z3=5:Z4=0:Z5=4
60 I=1:J=1
70 REM*********START YOUR CODE HERE************
80 REM
90 REM*************SUBROUTINES*****************
1000 REM MOTOR1 CW
1010 I=I+Z2:IF I=Z3 THEN I=Z2
1020 POKEZ1,Z(I)+16*Z(J):RETURN
2000 REM MOTOR1 CCW
2010 I=I-Z2:IF I=Z4 THEN I=Z5
2020 POKEZ1,Z(I)+16*Z(J):RETURN
3000 REM MOTOR2 CW
3010 J=J+Z2:IF J=Z3 THEN J=Z2
3020 POKEZ1,Z(I)+16*Z(J):RETURN
4000 REM MOTOR2 CCW
4010 J=J-Z2:IF J=Z4 THEN J=Z5
4020 POKEZ1,Z(I)+16*Z(J):RETURN
```

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 exactly as was done for the first motor, except the 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 6522 VIA's port B.

The software required to operate the two motors appears in Listing 5-6. 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.

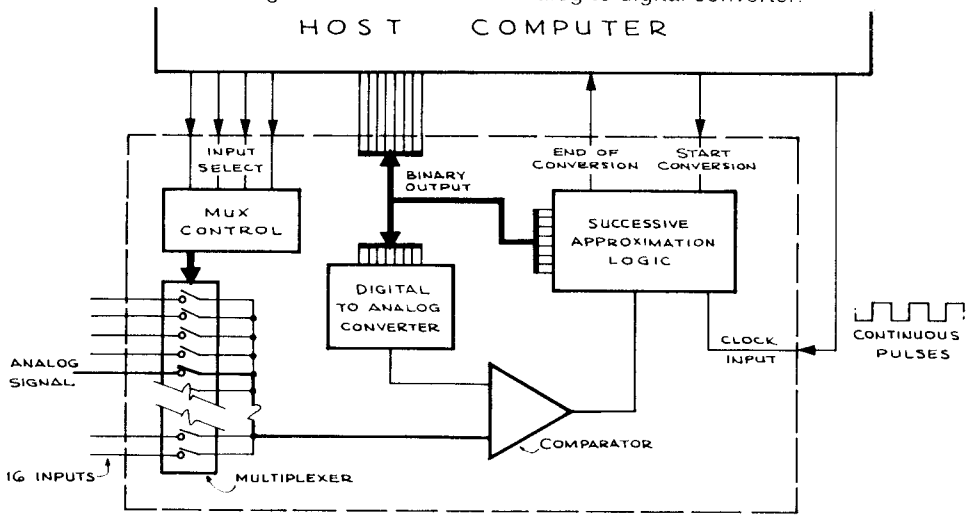
6

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.

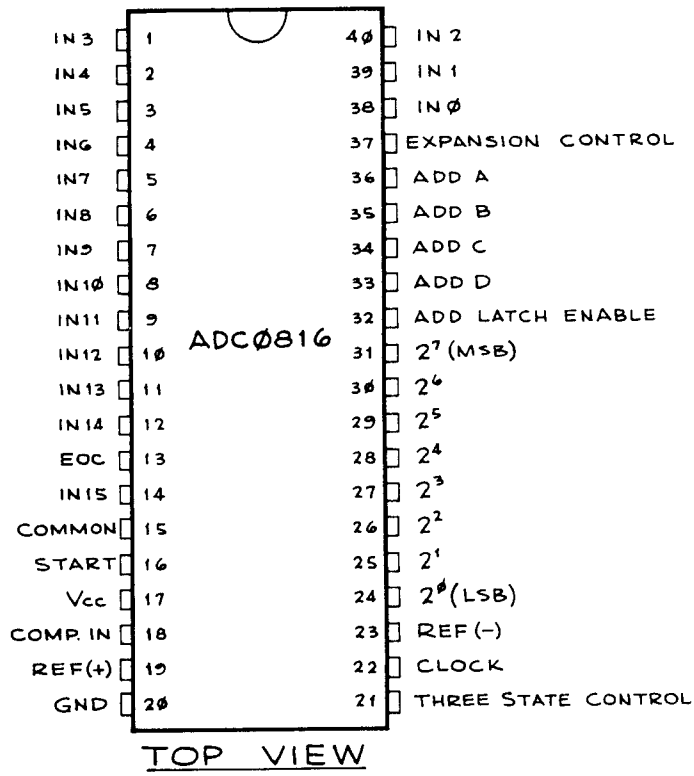


The ADC0816 Chip

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, 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 6502 data bus. The ADC0816 will complete a conversion in under 100 microseconds. Figure 6-3 shows the ADC0816 interfaced to a VIC 20 computer. Note that the VIA port A bit 2 goes to both the address latch enable (ALE) of the multiplexer and the START pin. The multiplexer address is latched from PB0-PB3 on the rising edge of the strobe pulse and the successive approximation logic is started on the falling edge. In this scheme, making PA2 high and then low with the port B outputting a 4-bit address will select one of the 16 channels in the multiplexer and start the converter. A PEEK to 37136 port B will retrieve the answer once the conversion is completed. An end-of-conversion signal (EOC) is available from the

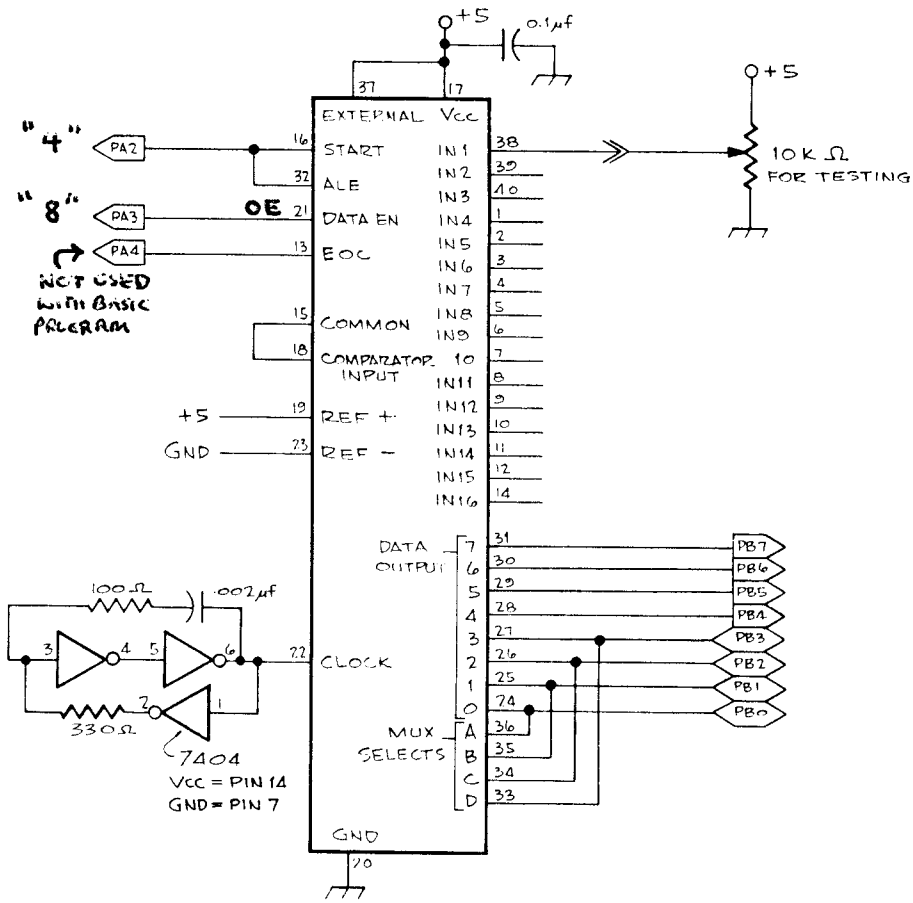
FIGURE 6-2
Pinouts for the ADC0816.



ADC0816 but is not at all needed with BASIC for the simple reason that the VIC 20 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 connected to PA4 and have the machine language program check its status 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 precision 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

FIGURE 6-3
Interconnections between the ADC0816 and the VIC 20.

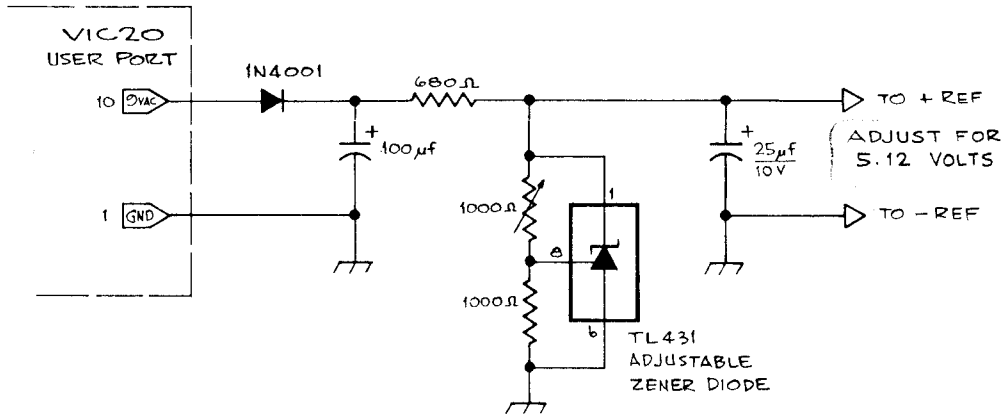


that 256, the maximum capacity of the 8-bit word, is an even multiple of 5.12. Thus each bit of the converted value will equal 5.12 ÷ 256, or 0.02 volts. If you use the 5-volt supply as the reference, the calibration factor will be 5.00 ÷ 256 or .01953, which is slightly less convenient. We have found 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 to 1 MHz clock. This clock was generated with an oscillator constructed from three inverters of a 7404, two resistors, and a .002 mfd capacitor.

Checkout of the ADC

Make the connections as shown in Figure 6-3. Doublecheck the connections and install only the 7404 into its socket. Be sure all power and

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



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 VIC 20. 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 with your logic probe for pulses from the clock oscillator on pin 6 of the 7404. Next, 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 37139,12                    (Initialize PA2 and PA3 outputs)
20 POKE 37138,15 : POKE 37136,0     (PB0-PB3 outputs : select channel 0)
30 POKE 37151,4 : POKE 37151,0      (Latch in channel : start conversion)
40 POKE 37151,8 : POKE 37138,0      (Enable ADC output : port B input)
50 PRINT PEEK (37136)               (Print digital value at port B)
60 GOTO 20
```

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 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 wiring between pins 16 and 32 of ADC and PA2 or 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. The following code illustrates how to initialize port A and select a channel.

```
10 POKE 37139,12                    (Initialize port A)
20 INPUT CH                         (Get channel # from
                                    user)
30 POKE 37138,15 : POKE 37136, CH   (Set port B bits 0-3 as
                                    output : output chan-
                                    nel number through
                                    port B)
```

```
40 POKE 37151,4 : POKE 37151,0        (Make ALE high : then
                                      START low)
```

This example of selecting an analog channel is straightforward. The access sequence starting in line 30 starts by setting bits 0–3 port B, making bits 0–3 outputs. The second POKE sends the channel number to the ADC, and the sequence in line 40 forces the ALE line high, 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 in the VIC 20 to do so.

Many analog-to-digital conversion applications require sampling at present time intervals. The internal real-time clock in the VIC 20 can be used to accomplish this timing. As an example of this technique, a simple acquisition program is included. The following code causes the ADC to take 100 samples at a rate of two per second.

```
10 DIM S(100)
20 PB=37236 : PA=PB+15
30 BD=PB+2 : AD+BD+1
40 POKE AD,12
50 GOTO 1000

100 POKE BD,15: POKE PB,CH
110 POKE PA,4: POKE PA, 8
120 POKE BD,0: AI=PEEK(PB)
130 POKE PA,0: RETURN

1000 T=TI
1010 FOR J=1 TO 100
1020 IF TI=T < 30 THEN 1020
1030 CH=0: GOSUB 100
1040 S(J) = AI: T=T+30
1050 NEXT J
```

The variables in lines 20 and 30 allow for ready code reading and speed up execution. The subroutine starting at 100 does the complete channel select and reading the analog results. Keeping this subroutine near the top of the program also helps with execution speed. The real-time clock is sampled in line 1010 and line 1020 allows a delay until 1 2 second has elapsed (30 Jiffies). Variations of this code can allow multichannel sampling and varied timing. Keep in mind, however, that the program overhead can become an upper limit to the number of samples you can make each second.

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 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 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 above program cause it to sample a voltage on channel 1 and set X equal to the same value in volts.

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, E in (max).

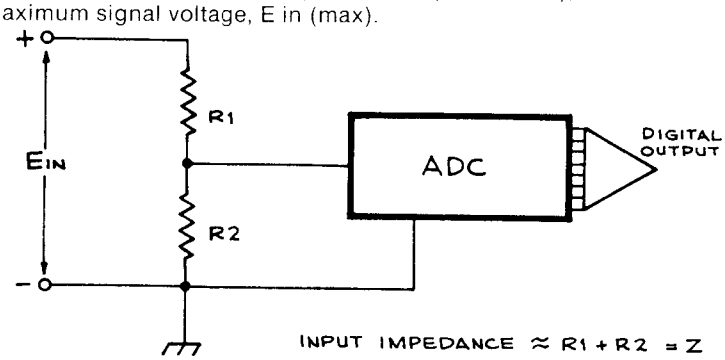
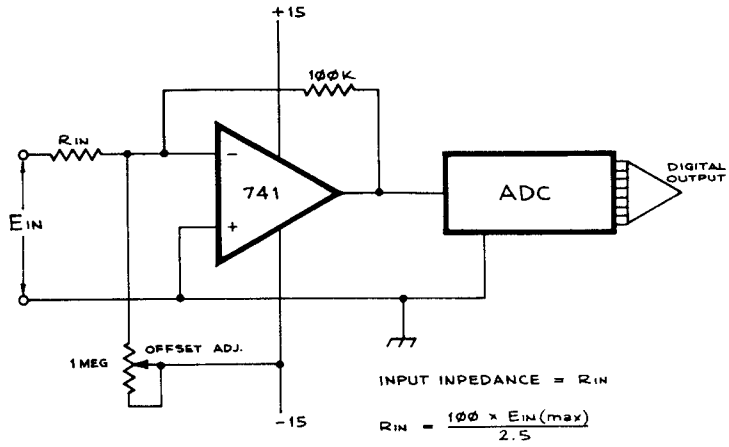


FIGURE 6-6
Op-amp signal conditioner. Use this circuit when Ein (max) is either less than 5 volts or when the signal is bipolar.




```
1030 CH = 0 : GOSUB 100
1040 X = (128-AI) * (1 128)
```

PARTS LIST FOR THE ADC

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 ¼ watt resistor
1 330 ohm ¼ watt 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 ¼ watt resistor
1 1000 ohm ¼ watt 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 VIC 20

This chapter describes how an ordinary portable audio cassette tape recorder can easily be connected to the VIC 20 and used to take the place of Commodore's commercial data cassette. No changes are required to either your audio recorder or your VIC 20. 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 tapes.

The Commodore cassette recorder is basically a converted audio recorder with modifications to make it useful 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 on tape as normal audio information.

By adding the simple electrical circuits given here, you can make your VIC 20 think it is connected to a compatible recorder and make use of your inexpensive standard audio recorder.

What Kind of Recorder?

For this project you will need a standard monophonic (single channel) portable audio cassette recorder* with microphone 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 result in a relatively inexpensive means of mass storage. A good information recording technique such as that found in the VIC 20 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.

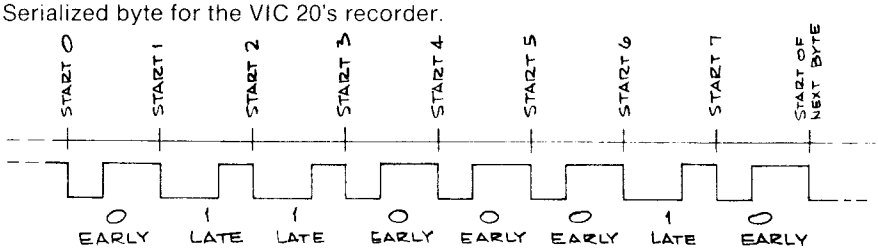
Before we dive deep into construction, let's first investigate the basics of how information can be put on tape as audio information and later be retrieved for reuse. All the details of how the VIC 20 saves information on the magnetic tape are beyond the scope of this book. However, a basic description of this process will help you understand the interface you will be building.

The unit of information saved is a 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 all taken care of automatically for us by programs built into your VIC 20 computer. The owner's manual that came with your VIC 20 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 to 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

*We used the Radio Shack CTR 56 for testing this project.

FIGURE 7-1
Serialized byte for the VIC 20's recorder.



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. 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 VIC 20. 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 a CMOS 4093 quad NAND gate to square the playback signal from the recorder.

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

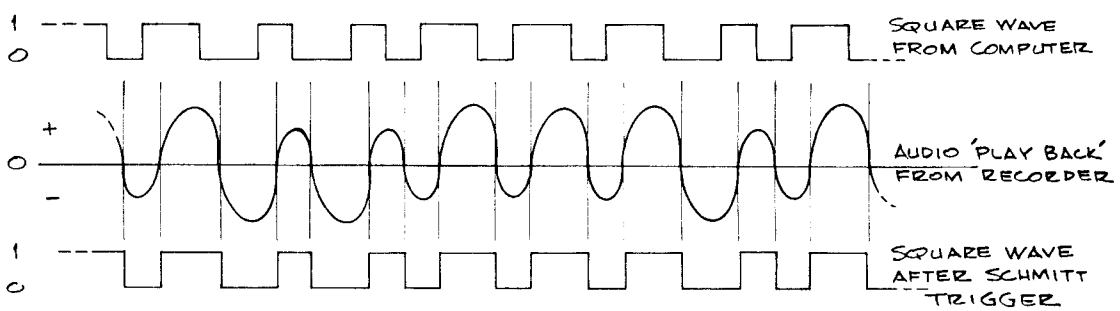
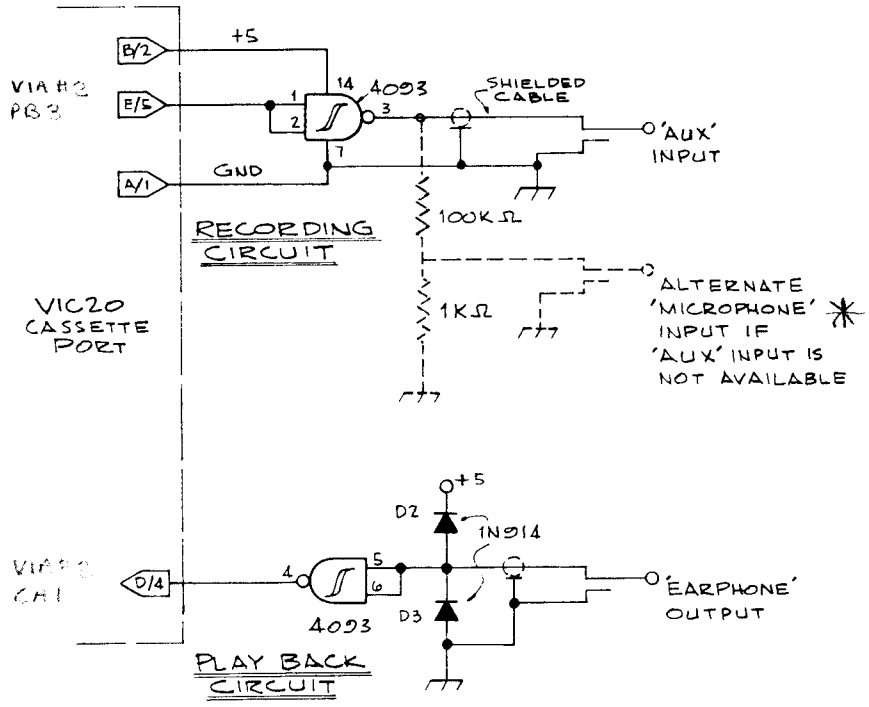


FIGURE 7-3
The interface between the audio recorder and the VIC 20.



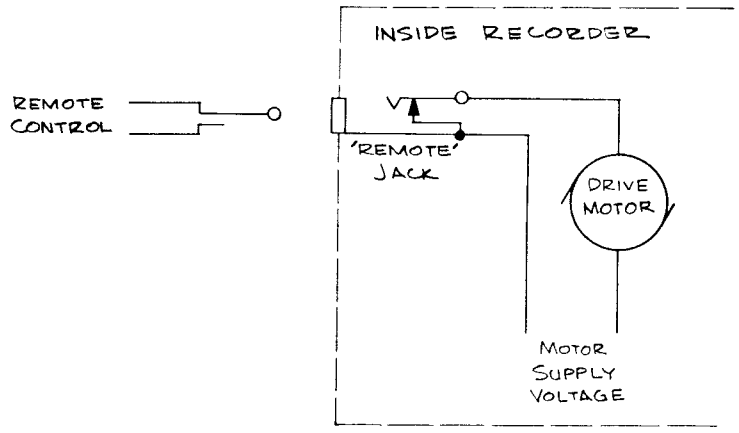
Since the 4093 is an inverter, the recorded data should also be inverted in order to preserve the polarity of the information we started with. Since there are four separate gates in the 4093, we can use one gate, again wired as an inverter, in the computer-to-recorder connection to invert the output. In this case, the Schmitt Trigger characteristic is not required; 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 VIC 20. Use the 115 VAC adaptor or batteries. You will, however, have to control the motor through the remote input of the

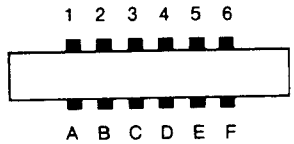
FIGURE 7-4
Remote control circuit in the recorder.



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 VIC 20 can start and stop the recorder motor. This is accomplished with a 5-volt, normally-open, single-pole relay from the VIC 20 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 VIC 20 when you have the tape recorder ready to SAVE or LOAD. The Commodore recorder PLAY button is connected to pins F or 6 to tell the VIC 20 when the PLAY button is pressed. Our motor control circuit must also provide this feedback signal to keep the VIC 20 happy. The input on pin F 6 has to be brought logically low before the VIC 20 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 VIC 20 when the tape recorder is ready. The VIC 20 will take care of starting and stopping the motor, but only so long as the VIC 20 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.

FIGURE 7-5
Pinouts for the cassette port.



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

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" x 2" piece of phenolic experimenters board with predrilled holes at 1/10" 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" ends of the board. Since the upper and lower pins of the VIC 20 cassette port are the same, they may be connected together. An easy way 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.

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

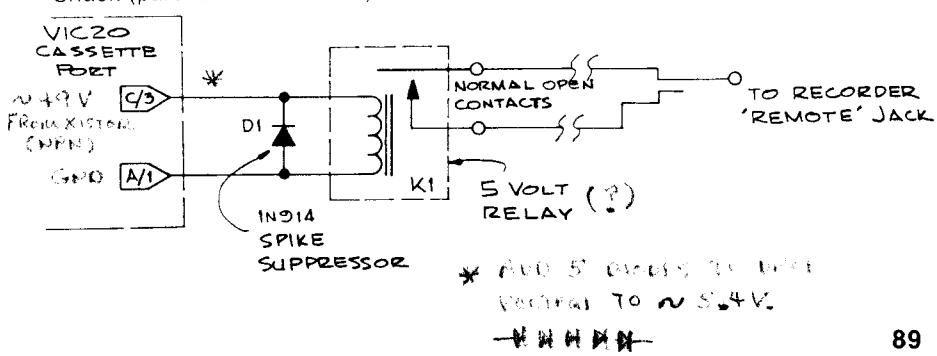
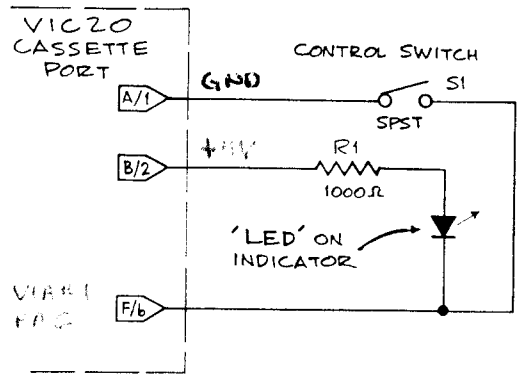
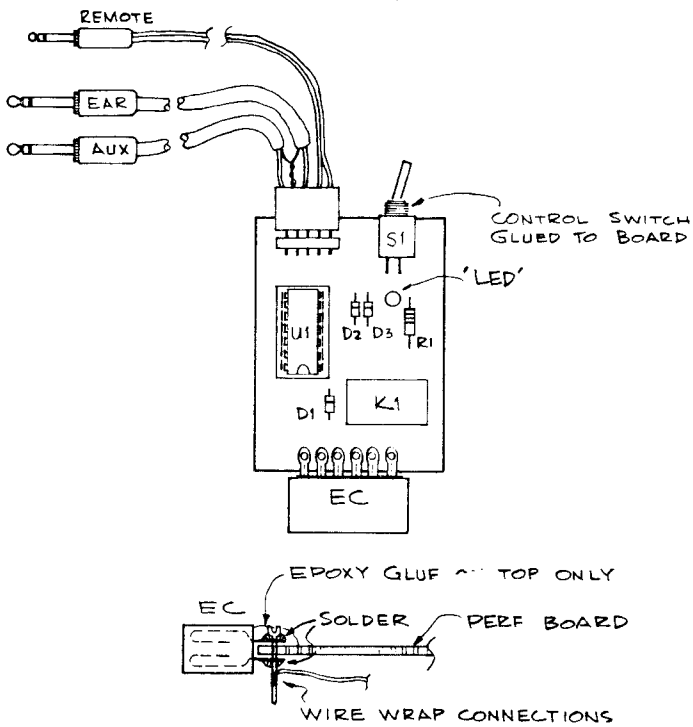


FIGURE 7-7
Ready control circuit for the recorder interface.



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

FIGURE 7-8
Mechanical details of the recorder interface.



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

PARTS LIST

1½" x 2" phenolic project board predrilled @ .100" x .100" grid (Cut from Radio Shack 4½" x 5-⅝" 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 .156 pin spacing
2 Mini phono jacks (AUX & EAR plug-ins)
1 Micro phono jack (REMOTE plug-in)
1 1000 ohm ¼ watt resistor
1 LED 15ma low power type

Misc.

4 feet of small-diameter shielded cable
2 feet of #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. Electrical circuits are like a computer program in that a circuit will only work properly if it is wired exactly as it should be. With the VIC 20 turned off, connect the interface to both the computer and the recorder. Set the recorder volume control to ¾ full and power up the VIC 20. 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 Vcc fuse inside the VIC 20's case.

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 a 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 which point the VIC 20 will output READY.

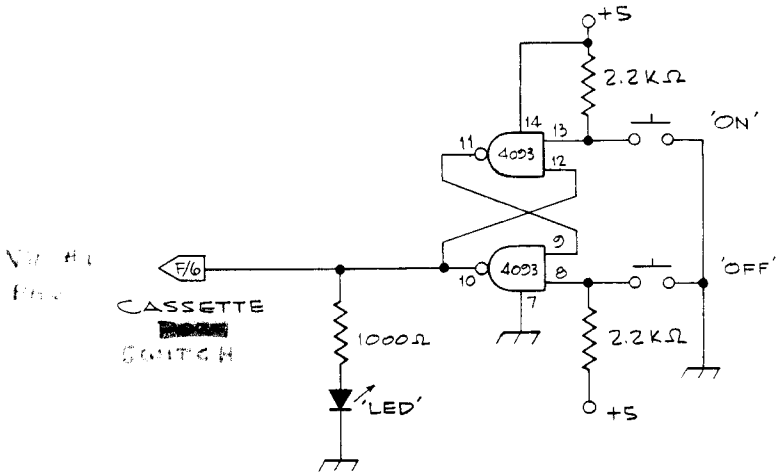
Now let's see if it worked. Press STOP and then REWIND on the recorder. The motor will not start until you turn the switch off, then back on. When the tape is rewound, turn the switch off and press STOP on the recorder. Type in NEW to erase the program from memory and then press both the SHIFT and RUN STOP keys on the VIC 20 at the same time. This is the auto load run request and the VIC 20 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. At the same time the recorder stops, the program will begin to execute. If everything went right, the program that was loaded will tell you so.

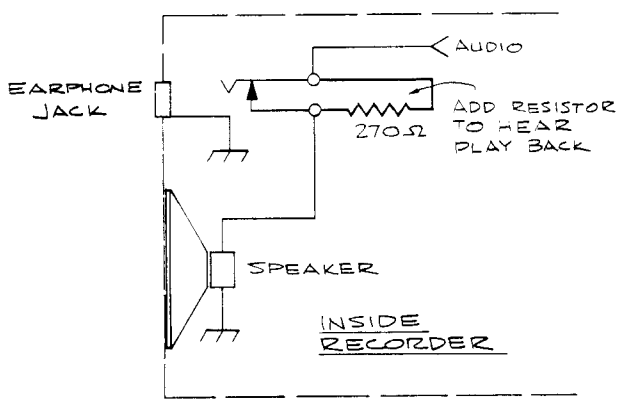
Troubleshooting

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

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





illustrated in Chapter 3, you can throughly 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 exactly the same as Commo-

dore'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. For more practice, try the experiment in the VIC 20'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 4093 wires 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 VIC 20.

About ROMs, PROMs, EPROMs, 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 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 VIC 20 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.

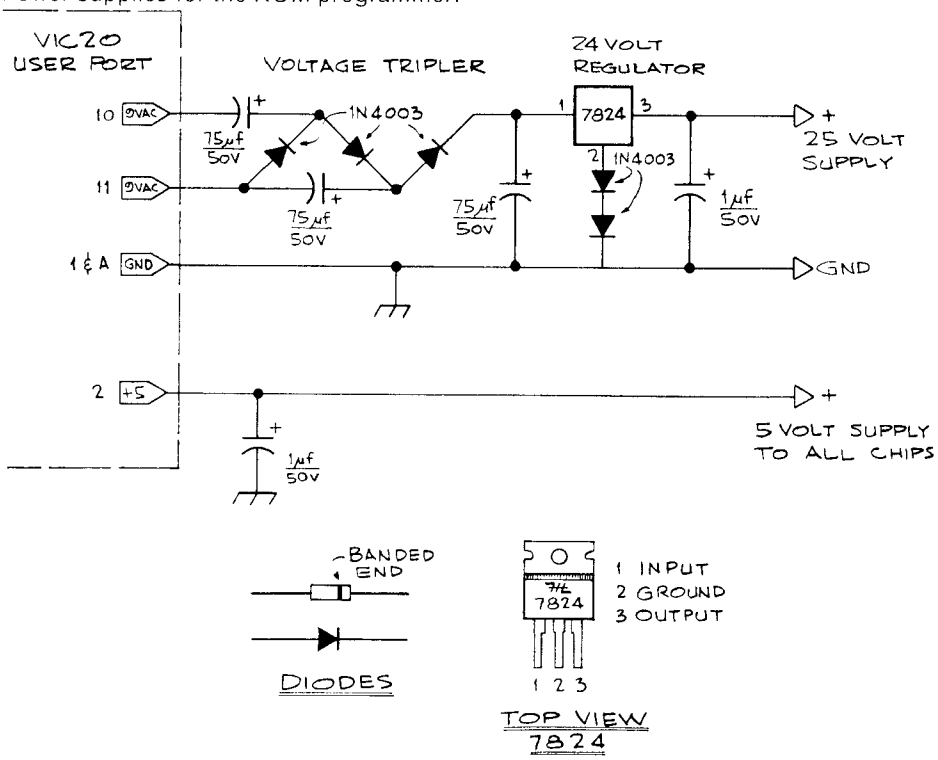
EEPROMs (electrically erasable PROMs) are a recently developed set of devices that are programmed in similar fashion to EPROMs but can be erased by an electrical pulse. Such ROMs are useful when in-circuit reprogramming is desired. Some are byte-erasable, and others are completely erased by the erase 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 VIC 20 is simple because of the availability of the two 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″ x 4.5″ 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″ 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 VIC 20 and thus not interfere with the other I/O connectors. It was necessary to cut about ⅛″ off the unused connectors so that the card and connector would fit properly into the recess for the VIC 20 user port.

FIGURE 8-1
Power supplies for the ROM programmer.

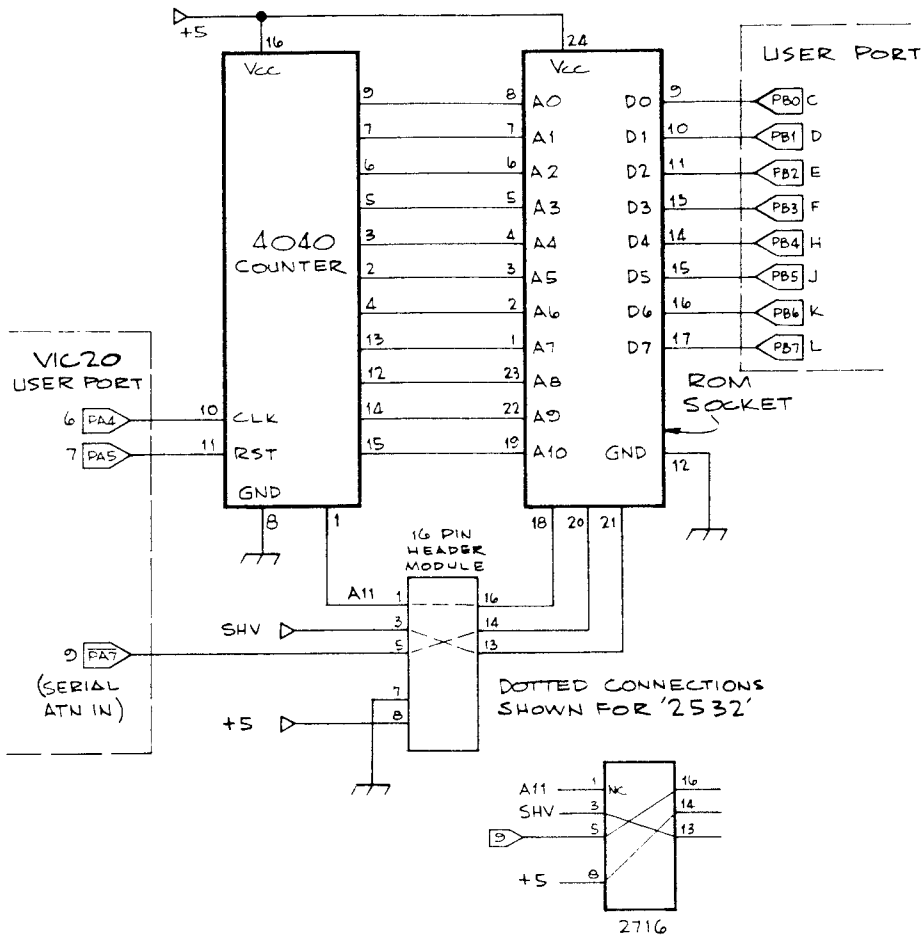


We ran heavy ground and +5V busses along the bottom of the board for convenient tie points. Discrete parts may be placed through holes and soldered together on the bottom side of the board. We suggest that you use wire-wrap sockets for the ICs and for the header module (optional) and the ROM socket.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

PARTS LIST

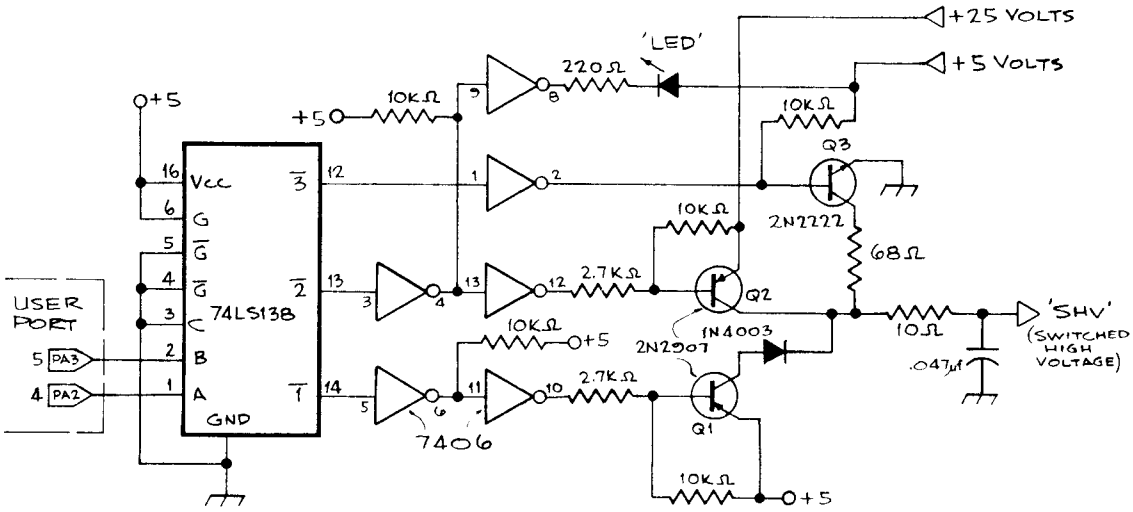
1 Vector 4.5 x 4.5 circuit card #3662-5 or equivalent
1 Edge connector w/ solder eyelets, 12/24 position
1 4040 12-stage binary counter
1 74LS138 3-line to 8-line decoder
1 7406 hex inverting buffer (open-collector)
1 14-pin wire-wrap socket
3 16-pin wire-wrap socket
1 24-pin wire-wrap socket. For extended use, substitute ZIF (zero insertion force) socket ($10).
1 7824 voltage regulator, TO-220 case

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.



6 1N4003 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 ¼ watt resistor
1 220 ohm ¼ watt resistor
1 68 ohm ¼ watt resistor
1 10 ohm ¼ watt resistor
2 2700 ohm ¼ watt 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 9V AC, which is available on the VIC 20 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 activated by bringing PA5 momentarily high. After each byte is programmed, the counter is advanced one step by bringing its CLOCK momentarily high using PA4. Data is supplied to the ROM using all eight 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. PA7 is provided for controlling the program pulse and can also be used as A12 for programming 8K ROMs. Note that the inverse of PA7 is the signal that is available on the user port pin 9, labelled "serial ATN in" in the VIC 20 documentation.

The pulse-switching circuit (Figure 8-3) uses PA2 and PA3 to select one of three of the outputs of a 74LS138 to control the SHV signal via transistors Q1, Q2, and Q3. The following table shows the voltage resulting from combinations of PA2 and PA3:

PA2	PA3	SHV
LOW	LOW	Floating
LOW	HIGH	25V
HIGH	LOW	5V
HIGH	HIGH	0V (Power-on condition)

This switching system is designed to be failsafe. Normally, the system will not inadvertently turn on the 25V programming voltage. Misoperation of the software can cause failure, however.

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 VIC 20 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:

```
POKE 37138,255
POKE 37136,255
(All port B outputs should go high)
POKE 37136,0
(all port B outputs should go low)
POKE 37139,255
POKE 37137,255
(all port A outputs should go high)
POKE 37137,0
(all port A outputs should go low)
Enter the following program and RUN:
10 POKE 37139,255
20 POKE 37137,16
30 POKE 37137,0
40 GOTO 20
(Pulses should appear on pin 10 of the 4040.)
The 4040 output pins should show pulses at various
frequencies. It will take several seconds for the last stage
(pin 1) to change from low to high and back again.
POKE 37137,32
```

```
POKE 37137,0
(should reset all 4040 outputs to low)
```

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

```
POKE 37139,255
POKE 37137,4
(5 volts should appear on SHV)
POKE 37137,8
(25 volts should appear on SHV and the red LED should
light)
POKE 37137,12
(0 volts should appear on SHV)
```

If the VIC 20 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 VIC 20 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 byte 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 VIC 20 system's 8K ROMs and some of VIC 20's game packs. Unfortunately, its present cost of about $40 at the time of this writing is somewhat high for experimenters. The 4K byte 2732 and 2732A have different pinouts on pins 18–21 from that used in the VIC 20 cartridges. With adaptors, however, they are suitable. The 4K byte 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

TABLE 8-1: A Partial Listing of Common EPROMs Programmable with Our Hardware

EPROM TYPE	*MANUFACTURER*	*#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

*Price at the time of this writing (Fall 1983).

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 may have to be high. Disaster in the form of ROM destruction is the result if this sort of minor detail is overlooked.

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 Figure 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 VIC 20 entirely. Alternately, you may verify the ROM by going into read mode before removing it from the socket.

FIGURE 8-4
Chip data for the type 2532 EPROMs (Courtesy of Motorola).

MOTOROLA

MCM2532

4096 × 8-BIT UV ERASABLE PROM

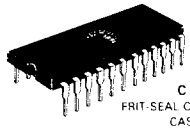
The MCM2532 is a 32,768-bit Erasable and Electrically Reprogrammable PROM designed for system debug usage and similar applications requiring nonvolatile memory that could be reprogrammed periodically. The transparent window in the package allows the memory content to be erased with ultraviolet light.

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

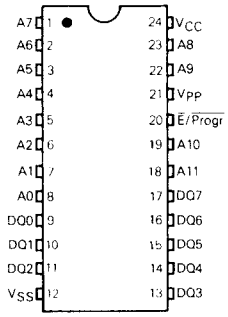
MOS
(N-CHANNEL, SILICON-GATE)

4096 × 8-BIT
UV ERASABLE PROM

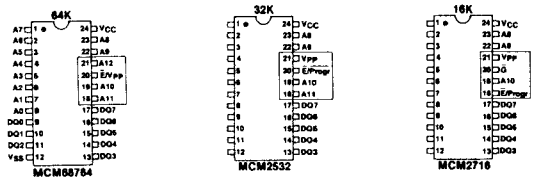


C SUFFIX
FRIT-SEAL CERAMIC PACKAGE
CASE 623A-02

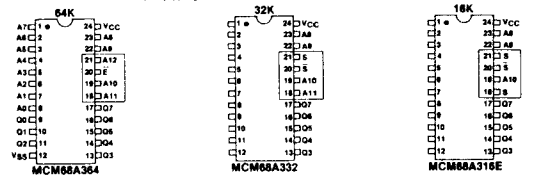
PIN ASSIGNMENT



MOTOROLA'S PIN-COMPATIBLE EPROM FAMILY



MOTOROLA'S PIN-COMPATIBLE ROM FAMILY



INDUSTRY STANDARD PINOUTS

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

*New Industry standard nomenclature

Unlike some earlier ROMs, the bytes of the 2532 can be programmed in any order we desire, and as many or as few bytes can be programmed as we require. Any address not programmed will retain with 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 wish 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 wish 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 3–8 minutes.

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 three 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 $1000, 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 VIA #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 $1C34 turns on the high voltage and brings pin 20 of the ROM high. The programming pulse is applied at $1C4F and after 50 ms delay is turned off at $1C63. The counter is incremented at $1C6B, and

LISTING 8-1 M/L program 2532.

```
ASM

                    1000 * PROGRAM 2532 IN VIC20
                    1010         .OR $1C00
                    1020         .TA $0800
                    1030 *       .TF VIC.PR2532.OBJ
                    1040 *---------------------------------
9110-               1050 VIA1    .EQ 37136
9110-               1060 PORTB   .EQ VIA1+0
9111-               1070 PORTA   .EQ VIA1+1
9112-               1080 DDRB    .EQ VIA1+2
9113-               1090 DDRA    .EQ VIA1+3
                    1100 *
00FB-               1110 RAMBEG .EQ $FB       ADH OF DATA - USER-SET
00FC-               1120 PAGCNT .EQ $FC       PAGE COUNTER
00FD-               1130 ADRPTR .EQ $FD & FE ADDRESS POINTER
                    1140 *
0318-               1150 NMIVEC .EQ $0318
1E00-               1160 VIDEO  .EQ $1E00
FF5B-               1170 RTI    .EQ $FF5B     ROM HAS RTI CODE HERE
                    1180 *---------------------------------
1C00- A9 FF         1190 START  LDA #$FF
1C02- 8D 13 91      1200        STA DDRA      ALL OUTPUT
1C05- 8D 12 91      1210        STA DDRB      ALL OUTPUT
1C08- A9 00         1220        LDA #$00
1C0A- 8D 11 91      1230        STA PORTA     ALL LOW
1C0D- 09 20         1240        ORA #$20      RESET COUNTER
1C0F- 8D 11 91      1250        STA PORTA
1C12- 29 DF         1260        AND #$DF      RESET OFF
1C14- 8D 11 91      1270        STA PORTA
1C17- A9 00         1280        LDA #$00
1C19- 85 FC         1290        STA PAGCNT
1C1B- 85 FD         1300        STA ADRPTR    BEGINNING OF PAGE
1C1D- A5 FB         1310        LDA RAMBEG    POINT TO ORIGINAL
1C1F- 85 FE         1320        STA ADRPTR+1
1C21- AD 18 03      1330        LDA NMIVEC    SAVE OLD NMI
1C24- 48            1340        PHA           ON STACK
1C25- AD 19 03      1350        LDA NMIVEC+1
1C28- 48            1360        PHA
1C29- A9 5B         1370        LDA #RTI      DISABLE NMI'S
1C2B- 8D 18 03      1380        STA NMIVEC
1C2E- A9 FF         1390        LDA /RTI
1C30- 8D 19 03      1400        STA NMIVEC+1
1C33- 78            1410        SEI           NO IRQ'S
1C34- A9 08         1420 DO2532 LDA #$08      PB7 LOW - PIN 20 HIGH
                    1430 *                    PB2/3 - 26V ON
1C36- 8D 11 91      1440        STA PORTA
1C39- A0 00         1450        LDY #0
1C3B- A5 FC         1460 ZAPLP  LDA PAGCNT
1C3D- C9 10         1470        CMP #$10      DONE 16 PAGES ?
1C3F- F0 3B         1480        BEQ DONE
1C41- B1 FD         1490 ZAPLP2 LDA (ADRPTR),Y DATA BYTE
1C43- C9 FF         1500        CMP #$FF
1C45- F0 1F         1510        BEQ STEP      DONT PGM FF'S
1C47- 8D 10 91      1520        STA PORTB     DATA TO PORTB
1C4A- AD 11 91      1530 ZAP    LDA PORTA
1C4D- 09 80         1540        ORA #$80      PROGRAM !
1C4F- 8D 11 91      1550        STA PORTA
1C52- 98            1560        TYA           SAVE Y
1C53- A2 28         1570        LDX #40
1C55- A0 00         1580        LDY #0
1C57- 88            1590 LP50   DEY           DELAY 50 MS
1C58- D0 FD         1600        BNE LP50
1C5A- CA            1610        DEX
```

```
1C5B- D0 FA     1620          BNE LP50
1C5D- A8        1630          TAY             RESTORE Y
1C5E- AD 11 91  1640          LDA PORTA
1C61- 29 7F     1650          AND #$7F
1C63- 8D 11 91  1660          STA PORTA       PGM PULSE OFF
1C66- AD 11 91  1670 STEP     LDA PORTA
1C69- 09 10     1680          ORA #$10
1C6B- 8D 11 91  1690          STA PORTA       BUMP THE COUNTER
1C6E- 29 EF     1700          AND #$EF
1C70- 8D 11 91  1710          STA PORTA
1C73- C8        1720 INCADR   INY
1C74- D0 CB     1730          BNE ZAPLP2
1C76- E6 FE     1740          INC ADRPTR+1
1C78- E6 FC     1750          INC PAGCNT
1C7A- D0 BF     1760          BNE ZAPLP       ALWAYS BRANCH
1C7C- A9 00     1770 DONE     LDA #$00        ALL INPUT
1C7E- 8D 13 91  1780          STA DDRA
1C81- 68        1790          PLA
1C82- 8D 19 03  1800          STA NMIVEC+1
1C85- 68        1810          PLA
1C86- 8D 18 03  1820          STA NMIVEC
1C89- 58        1830          CLI
1C8A- 00        1840          BRK             CHANGE TO 60-RTS FOR
                1850 *                        CALL FROM BASIC
                1860 *--------------------------------
```

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 7168
```

The 160 can be changed to the high-order address of other data locations. If you have relocated our machine code, change the 7168 accordingly.

This machine language system is best entered and called from a machine language monitor, such as VICMON.

The BASIC language in the VIC 20 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 which 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 leaving any portion unprogrammed. The first number in each pair is the decimal value of the relative address 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

LISTING 8-2 BASIC ROM program 2532.

```
READY.
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 PB=37136
230 PA=PB+1:DB=PB+2:DA=PB+3
240 POKE DA,255:POKE DB,255
250 POKE PA,0:POKE PB,0
260 POKE PA,32:POKE PA,0
270 POKE PA,8
280 B=0:K=0
290 FOR I=0 TO 4095
300 IF B(B,1)=0 THEN 350
310 BY=PEEK(B(B,1)+K)
320 IF BY=255 THEN 350
330 POKE PB,BY
340 POKE PA,136:FOR BY=1 TO 45:NEXT:POKE PA,8
350 POKE PA,26:POKE PA,8
360 K=K+1
370 IF I+1=B(B+1,0)THEN K=0:B=B+1
380 PRINT I:NEXT I
400 POKE DA,0
410 PRINT"■DONE
READY.
```

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.

(Continued)

(*Continued*)

ENTRY	*EXPLANATION*
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 8½ 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.
Lines 220–230 set up addresses for ports A and B and their data direction registers.
Line 240 sets both ports as outputs.
Line 250 sets all bits of both ports low.
Line 260 resets the hardware address counter.
Line 270 sets pin 20 of the ROM high and turns on the 25 volts to pin 21.
Line 280 sets B (block counter) to 0 and sets K (offset in current block) to 0.
Line 290 starts a loop to process 4096 bytes.
Line 300 bypasses bytes that are not to be programmed.
Line 310 fetches a byte from data location.
Line 320 bypasses bytes that are all ones.
Line 330 stores data byte in port B.
Line 340 turns on program pulse for 50 ms and then turns it off.
Line 350 steps the counter to the next address.
Line 360 increments the position in the block.
Line 370 tests for block done. If done, advances to next block.
Line 380 prints status report and goes to next byte.
Line 400 sets port A as input to normalize voltages.

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.

The only complication in the READ program is that, since many users have only the original VIC 20 memory complement, the program as written will read only the latter half of the ROM into RAM at addresses $1400-1BFF. To read the first half, delete lines 570 and 580; the same RAM buffer will be used. For owners who have expanded RAM, omit these two lines, change the 2047 in line 590 to read 4095, and set a suitable memory buffer location in line 500.

LINE-BY-LINE EXPLANATION OF LISTING 8-3.

Line 500 sets the location of the RAM buffer.
Line 510 sets the values of operations to be POKEd into port A.
Lines 520-530 set addresses of ports A and B and DDRs A and B.
Line 540 makes port A an output and port B an input.
Line 550 sets voltage to idle for read operation.
Line 560 resets the 4040 counter.
Lines 570-580 step the counter 2048 times to reach the second half of the ROM.
Line 590 sets up to read 2048 bytes.
Line 600 gets data from ROM and stores in RAM.
Line 610 steps the counter to the next address.
Line 620 gives visual feedback and goes to next byte.
Line 630 idles the system by making port A an input.

The astute reader will note that neither BASIC program has any string manipulation whatsoever. This omission is deliberate in order to

LISTING 8-3
BASIC ROM reader.

```
500 RAM=5120:PRINT"□"
510 IDLE=140:RESET=IDLE+32:BUMP=IDLE+16
520 BPRT=37136:APRT=BPRT+1
530 BDIREC=BPRT+2:ADIREC=BPRT+3
540 POKE ADIREC,255:POKE BDIREC,0
550 POKE APRT,IDLE
560 POKE APRT,RESET:POKE APRT,IDLE
570 FOR I=1 TO 2048:POKE APRT,BUMP:PRINT"-";
580 POKE APRT,IDLE:NEXT
590 FOR I=0 TO 2047
600 POKE RAM+I,PEEK(BPRT)
610 POKE APRT,BUMP:POKE APRT,IDLE
620 PRINT".";:NEXT I
630 POKE ADIREC,0:PRINT"DONE
READY.
```

avoid the use of string space in higher memory. We have also eliminated remarks to conserve memory. Establishing variables such as IDLE rather than using numbers improves the execution speed. If you can tolerate a blank screen for several minutes, omit the PRINT statements, because number evaluation and screen scrolling are time-consuming.

You can determine how much memory is available to you by loading the BASIC program, running it without a ROM in the socket, and entering:

```
PRINT FRE(I)
```

The value printed will indicate how much space is available. Since all variables are numeric and or numeric arrays, the space occupied by the BASIC program and its variables will be contiguous.

The user can be more casual in operating programs that do ROM reads than for ROM writes, because 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 "GORK" game you bought in a cartridge last Christmas, you can do some interesting things with the printed circuit board inside. Open the plastic cover and remove the circuit board. Using a solder-sucker and or solder-removing wick, loosen the pins of the ROM until the ROM can be removed. If you are careful, the ROM can be reused. Now, put a 24-pin, low-profile socket into the space vacated by the ROM and carefully resolder all 24 pins. On the top of the board, the connections from pins 12 (BLK3) and 13 (BLK5) go to an area where a solder blob can connect one of these traces to the ROM chip-select, pin 20. This connection method allows the ROM to exist either at addresses $6000-7FFF or $A000-BFFF, depending on the connection you choose.

Here is an example of a custom ROM application. Commodore sells a machine language monitor cartridge called VICMON. We highly recommend VICMON for any machine language programming on the VIC. It will certainly come in handy if you use the machine language ROM-programming routine. It is much easier to enter the hex code with VICMON than by using POKEs, and the monitor also has provisions for saving machine code on tape and reloading it.

The version we bought started at location $6000. (The documentation indicated that some of the VICMON cartridges start at $A000, so be sure to check the version you have.) Although it is provided in an 8K byte ROM, the actual program only occupied addresses $6000-$6EBF. Therefore, we copied the VICMON program into a 2532 EPROM and

used the remaining 320 bytes for the Epson printer driver, which is described in Chapter 9. With the new ROM in place, both VICMON and the printer driver are resident in the system at all times.

How to Make an Auto-Start System

Commodore provided for automatic program startup in the VIC 20 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 $A000 (decimal 40960). If the plug-in ROM has the arbitrary signature pattern

```
$A004  $41
 A005   30
 A006   C3
 A007   C2
 A008   CD
```

then normal startup does not take place. Instead, the program control is transferred to the address in the two bytes at $A000 and $A001 (low-order byte first, in standard 6502 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 $A002 and $A003. 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 wish to start your machine language immediately after the signature and that you wish RUN/STOP/RESTORE to act the same as power-up. The contents of your new ROM will start as follows:

```
A000  09  ADL of your program
A001  A0  ADH of your program
A002  09  ADL of your program
A003  A0  ADH of your program
A004  41  Signature byte
A005  30  Signature byte
A006  C3  Signature byte
A007  C2  Signature byte
A008  CD  Signature byte
A009  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, PIAs, 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 $A000–AFFF by having its chip-select pin 20 tied to connector pin 13 (BLK5). Since BLOCK 5 covers the range $A000–BFFF, a duplicate image will exist at $B000–BFFF, but this condition will not cause any problems.

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 VIC 20 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 VIC 20 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 at $A101 instead of $1001, 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 $1001 through $1DFF 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 $A100 instead of $1000. 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 its RAM location to the new ROM. (Be sure to copy the zero at $1000 to $A100, 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 VIC 20,

LISTING 8-4
Auto-start preamble.

```
:ASM
                    1000 * PREAMBLE FOR BASIC AUTO-START PROGRAM
                    1010         .OR $A000
                    1020         .TA $0800
                    1030 *---------------------------------
C483-               1040 BASICW  .EQ $C483     BASIC'S WARM START
FEC7-               1050 WARM    .EQ $FEC7     RESTORE WARM START
C660-               1060 CLEAR   .EQ $C660     BASIC CLEAR ROUTINE
                    1070 *
002B-               1080 SOB     .EQ $2B       START OF BASIC POINTER
002D-               1090 SOV     .EQ $2D       START OF VARIABLES POINTER
00C6-               1100 KEYCNT  .EQ $C6       # CHARS. IN KBD BUFFER
0277-               1110 KEYBUF  .EQ $0277     START KEY BUFFER
                    1120 *---------------------------------
A000- 0E A0         1130         .DA COLD      WHERE TO GO ON POWERUP
A002- C7 FE         1140         .DA WARM      RUN/STOP/RESTORE
A004- 41 30 C3
A007- C2 CD         1150         .HS 4130C3C2CD     SIGNATURE
A009- 93 52 55
A00C- 4E 0D         1160 RUN     .HS 9352554E0D     'RUN' LITERAL
A00E- 20 8D FD      1170 COLD    JSR $FD8D     CLR PG.02,03, DO MEMTEST
A011- 20 52 FD      1180         JSR $FD52     RESTORE I/O VECTORS
A014- 20 F9 FD      1190         JSR $FDF9     SETUP VIA'S
A017- 20 18 E5      1200         JSR $E518     INIT. VIC CHIP
A01A- 20 5B E4      1210         JSR $E45B     MOVE HOOKS TO RAM
A01D- 20 A4 E3      1220         JSR $E3A4     INIT. BASIC
A020- 20 04 E4      1230         JSR $E404     PRINT 'CBM' AND BYTES FREE
A023- A2 FB         1240         LDX ##FB      
A025- 9A            1250         TXS           SET STACK POINTER
A026- A5 2B         1260         LDA SOB       MOVE VARIABLES DOWN
A028- 85 2D         1270         STA SOV
A02A- A5 2C         1280         LDA SOB+1
A02C- 85 2E         1290         STA SOV+1
A02E- A9 01         1300         LDA ##01      SET SOB TO THIS ROM
A030- 85 2B         1310         STA SOB
A032- A9 A1         1320         LDA ##A1
A034- 85 2C         1330         STA SOB+1
A036- 20 60 C6      1340         JSR CLEAR
A039- A2 05         1350         LDX ##05      SET FOR 5 CHARS.
A03B- 85 C6         1360         STA KEYCNT
A03D- BD 08 A0      1370 LOOP    LDA RUN-1,X   STORE 'RUN' IN BUFFER
A040- 9D 76 02      1380         STA KEYBUF-1,X
A043- CA            1390         DEX
A044- D0 F7         1400         BNE LOOP
A046- 4C 83 C4      1410         JMP BASICW    TO BASIC WARM
                    1420 *---------------------------------
```

POKE 44,xx:NEW

where xx is the high-order address of the start of normal RAM BASIC programs, for example, 16 for an unexpanded VIC. To return to the auto-start program, emulate power-up by entering

SYS64802

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

LISTING 8-5
Relinker

```
                        1000 * SUBROUTINE TO RE-LINK FOR BASIC IN ROM
                        1010         .OR $1C00
                        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
A101-                   1070 ROMBAS  .EQ $A101      START OF BASIC IN ROM
                        1080 *--------------------------------
1C00- A5 2B             1090 START   LDA SOB        COPY SOB
1C02- 85 22             1100         STA WORK1
1C04- A5 2C             1110         LDA SOB+1
1C06- 85 23             1120         STA WORK1+1
1C08- A9 01             1130         LDA #ROMBAS    POINT TO BASIC IN ROM
1C0A- 85 24             1140         STA WORK2
1C0C- A9 A1             1150         LDA /ROMBAS
1C0E- 85 25             1160         STA WORK2+1
1C10- 18                1170         CLC
1C11- A0 01             1180 LOOP    LDY #$01
1C13- B1 22             1190         LDA (WORK1),Y
1C15- D0 01             1200         BNE SKIP
1C17- 60                1210         RTS            EXIT SUBROUTINE
1C18- A0 04             1220 SKIP    LDY #$04
1C1A- C8                1230 GETEND  INY
1C1B- B1 22             1240         LDA (WORK1),Y
1C1D- D0 FB             1250         BNE GETEND
1C1F- C8                1260         INY
1C20- 98                1270         TYA
1C21- 48                1280         PHA
1C22- 65 24             1290         ADC WORK2
1C24- A0 00             1300         LDY #$00
1C26- 91 22             1310         STA (WORK1),Y
1C28- 85 24             1320         STA WORK2
1C2A- A5 25             1330         LDA WORK2+1
1C2C- 69 00             1340         ADC #0
1C2E- C8                1350         INY
1C2F- 91 22             1360         STA (WORK1),Y
1C31- 85 25             1370         STA WORK2+1
1C33- 68                1380         PLA
1C34- 65 22             1390         ADC WORK1
1C36- 85 22             1400         STA WORK1
1C38- A9 00             1410         LDA #0
1C3A- 65 23             1420         ADC WORK1+1
1C3C- 85 23             1430         STA WORK1+1
1C3E- 90 D1             1440         BCC LOOP       ALWAYS BRANCH
                        1450 *--------------------------------
```

PRINT FRE(∅)

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

If you have difficulty accomplishing these steps for placing your BASIC program in ROM, a programming service is available from The Bit Stop.* Finally, if your program will not fit into a 2532 ROM, you may want an MCM68764, which has twice the capacity.

*The Bit Stop, 5958 S. Shenandoah Rd., Mobile, AL 36608, Attn: Don Rindsberg

9

INTERFACING A PARALLEL PRINTER

Parallel printers are very easily interfaced to your VIC 20, because all the active components required are on the VIC 20'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 on "The Enhanced VIC 20," *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, which 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 commonly 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 VIC 20 character set in the graphics mode on the Epson printers. This last item is probably of greatest interest to VIC 20 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 book.

Connecting the Hardware

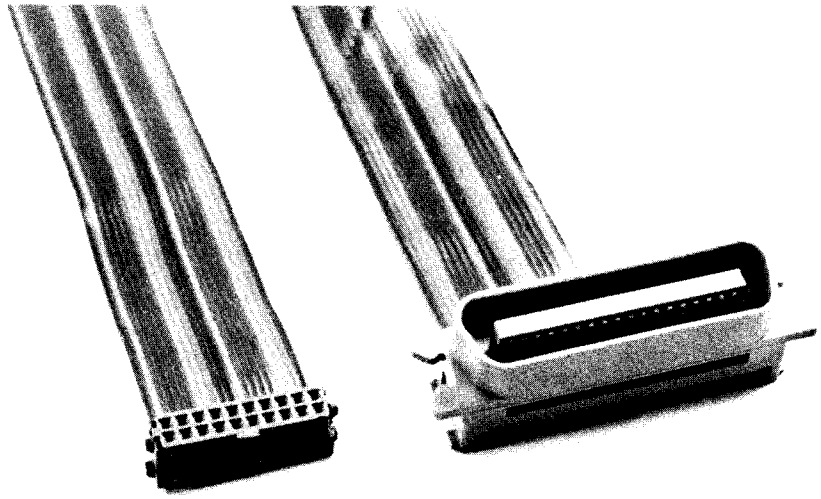
Any method of connecting 10 signal wires and one ground wire from the VIC 20 user port to the parallel printer will be satisfactory. We used a 4" x 4.5" Radio Shack board with 0.1" x 0.1" perforations and a 22 44 position edge connector. We sawed off the unused connector area and soldered a 12 24 pin 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 ohm meter for wiring continuity and shorts before the printer and computer are turned on.

TABLE 9-1: Interconnection Data for Parallel Printer with VIC 20

MX-70 OR MX-80 PIN NO.	*SIGNAL DESCRIPTION*	*VIC USER PORT PIN*	*VIC VIA FUNCTION*
1	STROBE to printer	M	CB2
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	CB1
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 VIC 20 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 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 VIC 20'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 printer.

The software is assembled for address $6EC0 (decimal 28352), but it can be reassembled to reside in any convenient location. We selected $6ECO because the routine can be placed into spare space in the VIC-MON ROM (see Chapter 8). The principal limitation of this system is that the VIC 20 does not issue a line-feed to start a new line, just a carriage returns. There are three methods of solving this problem:

1. Check to see if your printer has an automatic line-feed switch or jumper and set it accordingly. LIST will work well with this option.
2. Print only from a BASIC program and end your print lines with a line-feed CHR$ (10), such as:
   2040 PRINT "TOTAL"; T; CHR$(10)

LISTING 9-1
Epson print routine.

```
                1000 * PRINT ON EPSON IN ASCII MODE
                1010         .OR $6EC0
                1020         .TA $6EC0
                1030 *--------------------------------
9110-           1040 IORB1   .EQ $9110     I/O REG B
9112-           1050 DDRB1   .EQ $9112     DATA DIREC. B
911C-           1060 PCR1    .EQ $911C     PERIF. CONTRL. REG
911D-           1070 IFR1    .EQ $911D     INTERRUPT FLAG REG.
911E-           1080 IER1    .EQ $911E     INTERRUPT ENABLE REG.
                1090 *--------------------------------
F27A-           1100 VICOUT .EQ $F27A     VIC CHAR.OUTPUT ROUTINE
                1110 *--------------------------------
0326-           1120 DISVEC .EQ $0326     VECTOR TO CHAR. OUTPUT ROUTINE
                1130 *--------------------------------
6EC0- A9 EF     1140 OPEN    LDA #ASCOUT  REROUTE THE CHAR. OUTPUT VECTOR
6EC2- 8D 26 03 1150          STA DISVEC    TO POINT TO OUR ASCOUT
6EC5- A9 6E     1160         LDA /ASCOUT
6EC7- 8D 27 03 1170          STA DISVEC+1
6ECA- AD 1C 91 1180 INITOT LDA PCR1       GET PCR
6ECD- 29 0F     1190         AND #$0F     CLEAR B PORT BITS
6ECF- 09 A0     1200         ORA #$A0     SET TO AUTO-PULSE MODE
6ED1- 8D 1C 91 1210          STA PCR1     STORE IN PCR
6ED4- A9 10     1220         LDA #$10     DISABLE INTERRUPT
6ED6- 8D 1E 91 1230          STA IER1
6ED9- A9 FF     1240         LDA #$FF     SET ALL BITS TO OUTPUT
6EDB- 8D 12 91 1250          STA DDRB1
6EDE- A9 00     1260         LDA #0       START WITH A NULL
6EE0- 8D 10 91 1270          STA IORB1
6EE3- 60        1280         RTS
                1290 *--------------------------------
6EE4- A9 7A     1300 CLOSE   LDA #VICOUT  REROUTE VECTOR TO NORMAL
6EE6- 8D 26 03 1310          STA DISVEC
6EE9- A9 F2     1320         LDA /VICOUT
6EEB- 8D 27 03 1330          STA DISVEC+1
6EEE- 60        1340         RTS
                1350 *--------------------------------
6EEF- 20 7A F2 1360 ASCOUT JSR VICOUT    SEND TO SCREEN
6EF2- 48        1370         PHA          SAVE CHAR.
6EF3- A9 10     1380         LDA #$10     TEST INTERRUPT BIT
6EF5- 2C 1D 91 1390 WAITO  BIT IFR1
6EF8- F0 FB     1400         BEQ WAITO    WAIT FOR PRINTER READY
6EFA- 68        1410         PLA          GET CHAR.
6EFB- 8D 10 91 1420          STA IORB1    SEND IT
6EFE- 60        1430         RTS
                1440 *--------------------------------
```

A LIST command will overprint the lines.

3. Add a few bytes to the ASCOUT routine that will test for a RETURN character $0D, and, if so, first output a line-feed $0A. LIST will work well with this option.

The OPEN routine is called by executing SYS28352 either in keyboard mode or from within a BASIC program. The printer is closed by SYS28388. Note: When you open the printer, the VIC 20 will hang up if the printer's power is not on or if the cable is not connected. Also, trying to print the VIC 20'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 VIC 20 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 operation 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 *current* VIC 20 character generator, whether it be in ROM or RAM. 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 that transformation in our print routine.

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

LISTING 9-2
Epson graphics routine.

```
                1000 * PRINT ON EPSON IN GRAPHICS MODE
                1010         .OR $6EC0
                1020         .TA $6EC0
                1030 *       .TF VICEPS.OBJ3
                1040 *---------------------------------
00C7-           1050 RVSMOD .EQ $C7       REVERSE MODE FLAG
00D4-           1060 QUOTMO .EQ $D4       QUOTE MODE FLAG
00FB-           1070 VICPAT .EQ $FB       TEMPORARY POINTER
00FD-           1080 ASAVE  .EQ $FD       SAVE ACC. TEMP.
00FE-           1090 LCOUNT .EQ $FE       COUNT ON CURRENT LINE
                1100 *---------------------------------
9110-           1110 IORB1  .EQ $9110     I/O REG B
9112-           1120 DDRB1  .EQ $9112     DATA DIREC. B
911C-           1130 PCR1   .EQ $911C     PERIF. CONTRL. REG
911D-           1140 IFR1   .EQ $911D     INTERRUPT FLAG REG.
911E-           1150 IER1   .EQ $911E     INTERRUPT ENABLE REG.
9005-           1160 TVCTL5 .EQ $9005     CHAR. SET LOCATION
                1170 *---------------------------------
F27A-           1180 VICOUT .EQ $F27A     VIC CHAR.OUTPUT ROUTINE
                1190 *---------------------------------
0326-           1200 DISVEC .EQ $0326     VECTOR TO CHAR. OUTPUT ROUTINE
                1210 *---------------------------------
000D-           1220 CR     .EQ $0D       ASCII CARRIAGE RET.
0014-           1230 INSDEL .EQ $14       INSERT/DELETE CHAR.
001B-           1240 ESC    .EQ $1B       ASCII ESCAPE
000A-           1250 LF     .EQ $0A       ASCII LINEFEED
003C-           1260 LINLIM .EQ 60        CHARS./LINE
000C-           1270 LPI6   .EQ 12        CODE FOR 6 LINES/INCH
                1280 *---------------------------------
```

```
6EC0- A9 02    1290 OPEN   LDA #GRFOUT  REROUTE THE CHAR. OUTPUT VECTOR
6EC2- 8D 26 03 1300        STA DISVEC    TO POINT TO OUR GRFOUT
6EC5- A9 6F    1310        LDA /GRFOUT
6EC7- 8D 27 03 1320        STA DISVEC+1
6ECA- AD 1C 91 1330 INITOT LDA PCR1      GET PCR
6ECD- 29 0F    1340        AND #$0F      CLEAR B PORT BITS
6ECF- 09 A0    1350        ORA #$A0      SET TO AUTO-PULSE MODE
6ED1- 8D 1C 91 1360        STA PCR1      STORE IN PCR
6ED4- A9 10    1370        LDA #$10      DISABLE INTERRUPT
6ED6- 8D 1E 91 1380        STA IER1
6ED9- A9 FF    1390        LDA #$FF      SET ALL BITS TO OUTPUT
6EDB- 8D 12 91 1400        STA DDRB1
6EDE- A9 00    1410        LDA #0        START WITH A NULL
6EE0- 8D 10 91 1420        STA IORB1
6EE3- A9 1B    1430 INITPR LDA #ESC      ESC-A-# SETS LINES/INCH
6EE5- 20 D7 6F 1440        JSR PUTCHR     ON EPSON PRINTER
6EE8- A9 41    1450        LDA #'A
6EEA- 20 D7 6F 1460        JSR PUTCHR
6EED- A9 0C    1470        LDA #LPI6
6EEF- 20 D7 6F 1480        JSR PUTCHR
6EF2- 4C E4 6F 1490        JMP INILIN    INITIALIZE FIRST LINE
               1510 *--------------------------------
6EF5- A9 7A    1520 CLOSE  LDA #VICOUT   REROUTE VECTOR TO NORMAL
6EF7- 8D 26 03 1530        STA DISVEC
6EFA- A9 F2    1540        LDA /VICOUT
6EFC- 8D 27 03 1550        STA DISVEC+1
6EFF- 4C 75 6F 1560        JMP FILLIN    FINISH LAST LINE
               1570 *--------------------------------
6F02- 85 FD    1580 GRFOUT STA ASAVE     SAVE A,X,Y
6F04- 48       1590        PHA
6F05- 8A       1600        TXA
6F06- 48       1610        PHA
6F07- 98       1620        TYA
6F08- 48       1630        PHA
6F09- A5 FD    1640        LDA ASAVE
6F0B- 20 7A F2 1650        JSR VICOUT    CHAR. TO SCREEN
6F0E- AA       1660        TAX           SET SIGN FLAG
6F0F- 10 1C    1670        BPL BITOFF    BYPASS IF POSITIVE
6F11- 29 7F    1680        AND #$7F      TURN OFF BIT 7
6F13- C9 7F    1690        CMP #$7F      CONVERT $7F TO $5E
6F15- D0 02    1700        BNE NOT7F
6F17- A9 5E    1710        LDA #$5E
6F19- C9 20    1720 NOT7F  CMP #$20      CONTROL CHAR?
6F1B- B0 0C    1730        BCS NOTCTL    BRANCH IF NOT
6F1D- C9 0D    1740        CMP #CR       CAR.RET?
6F1F- F0 47    1750        BEQ FINLIN    BRANCH IF SO
6F21- A6 D4    1760        LDX QUOTMO    IN QUOTE MODE?
6F23- F0 49    1770        BEQ GRFBAK    NO, THEN IGNORE IT
6F25- 09 C0    1780        ORA #$C0      SET BITS 6 & 7
6F27- D0 26    1790        BNE SAVPOK    ALWAYS BRANCH
6F29- 09 40    1800 NOTCTL ORA #$40      TURN ON BIT 6
6F2B- D0 1C    1810        BNE CKRVS     ALWAYS BRANCH
6F2D- C9 0D    1820 BITOFF CMP #CR       IS IT RETURN?
6F2F- F0 37    1830        BEQ FINLIN    IF SO, END LINE
6F31- C9 20    1840        CMP #$20      CONTROL CHAR?
6F33- B0 0A    1850        BCS NOCTL     BRANCH IF NOT
6F35- C9 14    1860        CMP #INSDEL   DELETE?
6F37- F0 35    1870        BEQ GRFBAK    IGNORE IT
6F39- A6 D4    1880        LDX QUOTMO    QUOTE MODE ON?
6F3B- D0 10    1890        BNE HIBIT     IF SO, PRINT IT
6F3D- F0 2F    1900        BEQ GRFBAK    IF NOT, IGNORE
6F3F- C9 60    1910 NOCTL  CMP #$60      OVER $60?
6F41- 90 04    1920        BCC LOWER     BRANCH IF NOT
6F43- 29 DF    1930        AND #$DF      BIT 5 OFF
6F45- D0 02    1940        BNE CKRVS     BRANCH ALWAYS
6F47- 29 3F    1950 LOWER  AND #$3F      BITS 6/7 OFF
```

```
6F49- A6 C7     1960 CKRVS  LDX RVSMOD   REVERSE MODE?
6F4B- F0 02     1970        BEQ SAVPOK   BRANCH IF NOT
6F4D- 09 80     1990 HIBIT  ORA #$80     BIT 7 ON
6F4F- 85 FD     2000 SAVPOK STA ASAVE    SAVE THE CODE
6F51- A5 FE     2010        LDA LCOUNT   CHECK CHAR COUNT
6F53- C9 3C     2020        CMP #LINLIM  AT LIMIT?
6F55- D0 08     2030        BNE NOTFUL   BRANCH IF NOT
6F57- A9 0A     2040        LDA #LF      LINEFEED TO PRINTER
6F59- 20 D7 6F  2050        JSR PUTCHR
6F5C- 20 E4 6F  2060        JSR INILIN   START NEW LINE
6F5F- A5 FD     2070 NOTFUL LDA ASAVE    GET CHAR. BACK
6F61- 20 89 6F  2080        JSR SEND     SEND TO PRINTER
6F64- E6 FE     2090        INC LCOUNT   BUMP CHAR.COUNT
6F66- D0 06     2100        BNE GRFBAK   BRANCH ALWAYS
6F68- 20 75 6F  2110 FINLIN JSR FILLIN   FILL LINE WITH BLANKS
6F6B- 20 E4 6F  2120        JSR INILIN   START NEW LINE
6F6E- 68        2130 GRFBAK PLA          RESTORE Y,X,A
6F6F- A8        2140        TAY
6F70- 68        2150        PLA
6F71- AA        2160        TAX
6F72- 68        2170        PLA
6F73- 18        2180        CLC          MAKE VIC HAPPY
6F74- 60        2190        RTS          AND RETURN
                2200 *-------------------------------
6F75- A6 FE     2210 FILLIN LDX LCOUNT   GET CHAR COUNT
6F77- E0 3C     2220        CPX #LINLIM  AT LIMIT?
6F79- B0 09     2230        BCS NXTLIN   IF SO, DO LINEFEED
6F7B- A9 20     2240        LDA #$20     SEND BLANK
6F7D- 20 89 6F  2250        JSR SEND
6F80- E6 FE     2260        INC LCOUNT
6F82- D0 F1     2270        BNE FILLIN   BRANCH ALWAYS
6F84- A9 0A     2280 NXTLIN LDA #LF      SEND LINEFEED
6F86- 4C D7 6F  2290        JMP PUTCHR
                2300 *-------------------------------
6F89- 85 FB     2310 SEND   STA VICPAT   SAVE POKE CODE
6F8B- A9 00     2320        LDA #0
6F8D- 85 FC     2330        STA VICPAT+1
6F8F- A0 02     2340        LDY #2       MULTIPLY BY 8 TO GET
6F91- 18        2350 MULT8  CLC           OFFSET INTO CURRENT CHAR SET
6F92- 26 FB     2360        ROL VICPAT
6F94- 26 FC     2370        ROL VICPAT+1
6F96- 88        2380        DEY
6F97- 10 F8     2390        BPL MULT8    LOOP 3 TIMES
6F99- A2 80     2410        LDX #$80
6F9B- AD 05 90  2420        LDA TVCTL5   USE DATA FROM TV CHIP TO
6F9E- 48        2430        PHA           GET LOCATION OF CHAR SET
6F9F- 29 0C     2440        AND #$0C
6FA1- F0 02     2450        BEQ NOALT
6FA3- A2 10     2460        LDX #$10
6FA5- 8A        2470 NOALT  TXA          NOW CALCULATE ADDRESS OF
6FA6- 18        2480        CLC           CURRENT CHAR.
6FA7- 65 FC     2490        ADC VICPAT+1
6FA9- 85 FC     2500        STA VICPAT+1
6FAB- 68        2510        PLA
6FAC- 29 03     2520        AND #$03
6FAE- 18        2530        CLC
6FAF- 0A        2540        ASL
6FB0- 0A        2550        ASL
6FB1- 65 FC     2560        ADC VICPAT+1
6FB3- 85 FC     2570        STA VICPAT+1
6FB5- A9 01     2580        LDA #1       THIS ROUTINE CONVERTS DOT ROWS TO
6FB7- 48        2590 BYTLP  PHA           DOT COLUMNS, SAVES ON STACK,
6FB8- AA        2600        TAX           AND OUTPUTS THE COLUMNS TO PRINTER
6FB9- A0 07     2610        LDY #7
6FBB- B1 FB     2620 BITLP  LDA (VICPAT),Y
6FBD- 0A        2630 SHFTLP ASL
```

```
6FBE- CA        2640         DEX
6FBF- D0 FC     2650         BNE SHFTLP
6FC1- 66 FD     2660         ROR ASAVE
6FC3- 68        2670         PLA
6FC4- 48        2680         PHA
6FC5- AA        2690         TAX
6FC6- 88        2700         DEY
6FC7- 10 F2     2710         BPL BITLP
6FC9- A5 FD     2720         LDA ASAVE
6FCB- 20 D7 6F  2730         JSR PUTCHR
6FCE- 68        2740         PLA
6FCF- 18        2750         CLC
6FD0- 69 01     2760         ADC #1
6FD2- C9 09     2770         CMP #9
6FD4- D0 E1     2780         BNE BYTLP
6FD6- 60        2790         RTS
                2800 *--------------------------------
6FD7- 48        2820 PUTCHR PHA          SAVE CHAR.
6FD8- A9 10     2830        LDA #$10     TEST INTERRUPT BIT
6FDA- 2C 1D 91  2840 WAITO  BIT IFR1
6FDD- F0 FB     2850        BEQ WAITO    WAIT UNTIL PRINTER READY
6FDF- 68        2860        PLA          GET CHAR.
6FE0- 8D 10 91  2870        STA IORB1    OUTPUT TO VIA
6FE3- 60        2880        RTS
                2890 *--------------------------------
6FE4- A9 00     2900 INILIN LDA #0       CLEAR LINE COUNT
6FE6- 85 FE     2910        STA LCOUNT
6FE8- A9 1B     2920        LDA #ESC     SET PRINTER FOR 480 DOT COLUMNS
6FEA- 20 D7 6F  2930        JSR PUTCHR
6FED- A9 4B     2940        LDA #'K
6FEF- 20 D7 6F  2950        JSR PUTCHR
6FF2- A9 E0     2960        LDA #480
6FF4- 20 D7 6F  2970        JSR PUTCHR
6FF7- A9 01     2980        LDA /480
6FF9- 4C D7 6F  2990        JMP PUTCHR
                3000 *--------------------------------
```

placed in ROM. The entire routine will fit into the spare space in VICMON (see Chapter 8 on ROM programming). 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. To activate the printer, execute SYS28352. To deactivate it, execute SYS28405. Both program listing and printing from within a program work well. Sixty characters can be printed on each line, using paper of 8.5″ width.

10

THE GAME PORT

The game port, located on the right side of the VIC 20, provides yet another means by which the computer can communicate with the outside world. Commodore has already used this capability for game paddles, joysticks, and their light pen. In this chapter, we will explore some of the capabilities of this port.

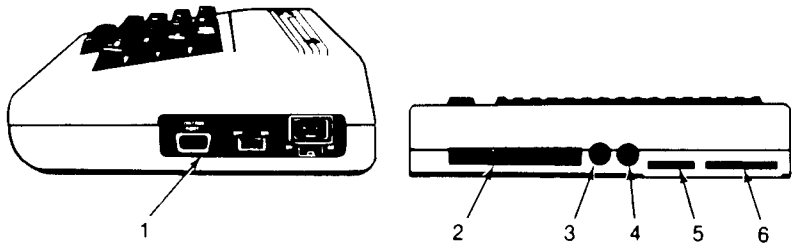
The Hardware

Figure 10-1 shows the pin assignments of the game port connector. The game port requires a 9-pin, female, type D, subminiature connector (available from Radio Shack). The game port's functions actually come from two different devices in the VIC 20. The joystick leads come from a 6522 VIA chip and can be used as five digital output lines or as five TTL level-sensitive input lines. The game paddle and light pen inputs both come from the 6560 VIC chip. The game paddle provides two pseudo-analog to digital converters while the light pen line provides the ability to sense a light pen's position on the monitor's screen.

Joystick Functions

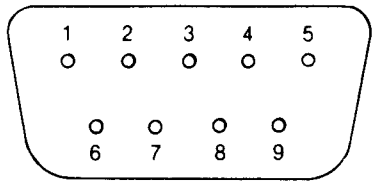
The joystick lines, Joy 0-Joy 2 and the fire button (located on the light pen line) are connected to PA2-PA5 of VIA #1. Of course, these lines are

FIGURE 10-1
Pin assignments for the VIC 20's game port (Courtesy of Commodore).



1) Game I/O
2) Memory Expansion
3) Audio and Video
4) Serial I/O (disk)
5) Cassette
6) User Port (modem)

Game I O



PIN #	TYPE	NOTE
1	JOY∅	
2	JOY1	
3	JOY2	
4	JOY3	
5	POT Y	
6	LIGHT PEN	
7	+5V	MAX.100mA
8	GND	
9	POT X	

available to the user for use either as outputs or inputs and can be used as outlined in Chapter 3. Be aware, however, that the remaining bits of this port are connected to the serial interface and the cassette recorder. Thus, when you enable bits 2-5, be sure not to disturb the status of the remaining bits. The data direction register for port A of VIA #1 is at $9113 (37139 decimal). To enable only these lines for inputs:

```
POKE 37139,130 AND PEEK (37139).
```

To enable these lines as outputs, you can use this command:

```
POKE 37139,120 OR PEEK (37139).
```

Joystick switch 3 (Joy 3) is on PB7 of VIA #1. This also happens to be used to read column 7 of the keyboard and *must be restored as it was* when you are through using it. Otherwise, a dead keyboard results. Port

B's data direction register is at $9122 (37154 decimal). The port A register is located at $9111 (37137 decimal) while the port B register is at $9120 (37152 decimal). Chapter 3 describes a variety of ways to interface these VIA lines to the outside world, but here are several more ways that may be of interest.

Demonstrating the Operation of a Switch Closure

Connect wires to pin 3 (Joy 2) and pin 8 (GND) of the game port and enter the following program.

```
10 POKE 37139,130 AND PEEK (37139)
20 PRINT 16 AND PEEK (37137)
30 GOTO 20
```

Run the program. The screen should fill with 16s. Now, short the two wires together and note that the 16s are replaced with 0s. The program reads bit 4 (the 16 weight bit) of port A. When these lines are in the input mode, 10K resistors inside the VIA chip pull these lines high to a logic 1. Shorting the line to ground causes it to change state. 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.

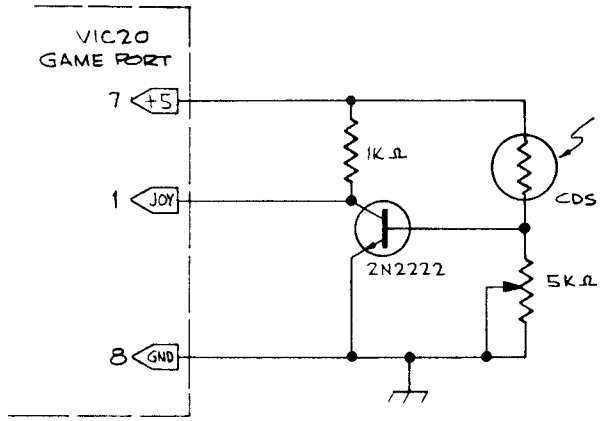
Optical Detectors

A useful application is to have the VIC 20 sense a changing light level. This has many applications, such as 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, whereas 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.

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,

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



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 of 276-142. Figure 10-4 shows a pair of these interfaced to the VIC 20 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 VIC 20 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

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

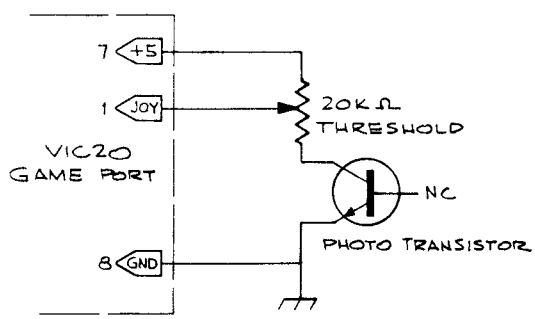
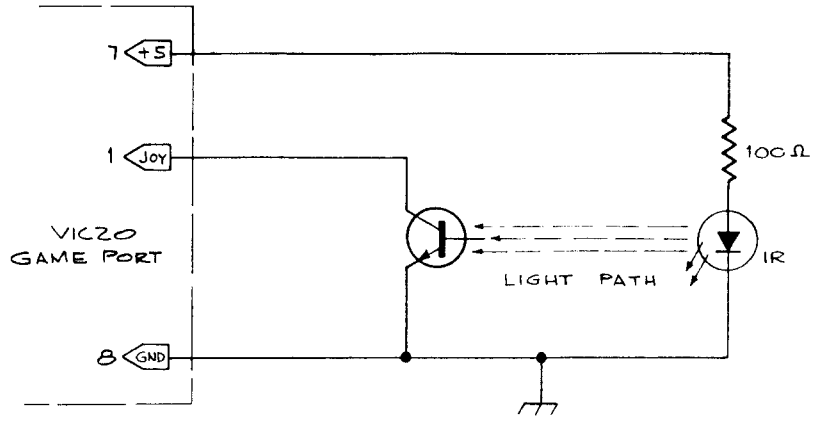
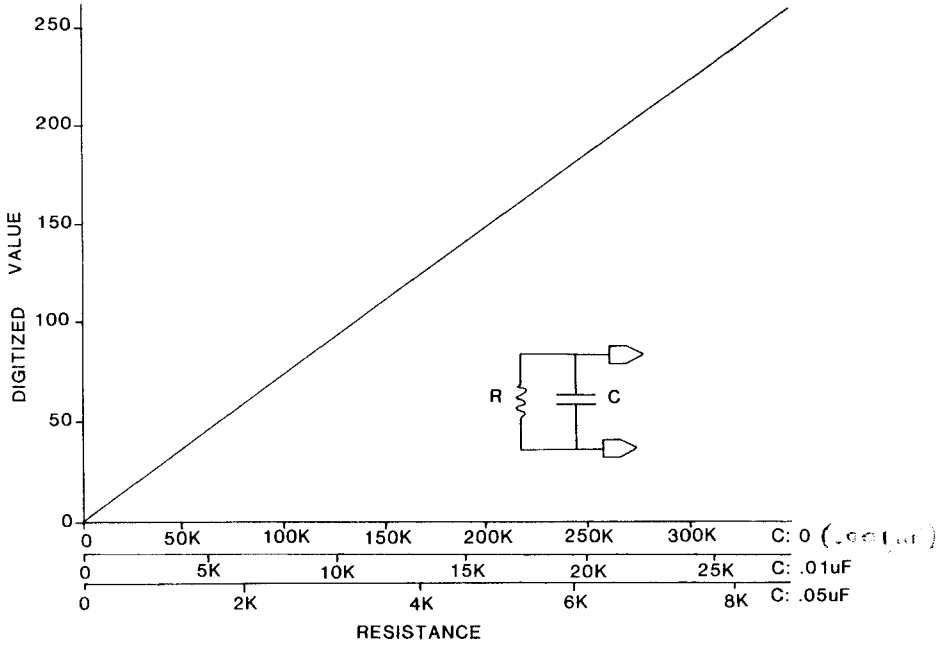


FIGURE 10-4
An infrared emitter and detector interfaced to a joystick input. The infrared detector is unaffected by ambient light. Radio Shack # 276-142 infrared emitter-defector pair is recommended.



the machine to sense the position of a potentiometer. Two of these are implemented in the VIC 20 and are on pins 5 and 9 of the game port. Actually these are pseudoanalog-to-digital converters. They actually measure the resistance between the +5 volt supply and the input line. The

FIGURE 10-5
A graph that relates the number returned by the VIC 20 as a function of the resistance across the game paddle's input. The three horizontal scales result from three different values of capacitance in parallel with the resistance.



game paddle inputs are not capable of sensing a voltage directly. The resistance is determined by measuring the RC time constant of the external resistor and an internal .001 uf (microfarad) capacitor. Figure 10–5 shows a graph of the value computed by the VIC chip on the vertical axis and the resistance on 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. The resistance cannot be too small; otherwise, the capacitor cannot be discharged between cycles. About 200 ohms was as low as we could use. If the resistance is too low, the oscillator locks up and 255s are returned.

Obviously, any variable resistance, such as a potentiometer, can be coupled to the game paddle to become an analog sensor. The +5 volts are safe to humans, and thus two electrodes (dimes are good for this) placed on the skin turns the VIC 20 into a galvanic skin response meter. Place two properly spaced electrodes in a cup of fluid and you have a conductance meter. Also, the VIC 20 makes a fairly accurate ohm meter for calibrating any unmarked resistors you may have lying around.

Photosensors

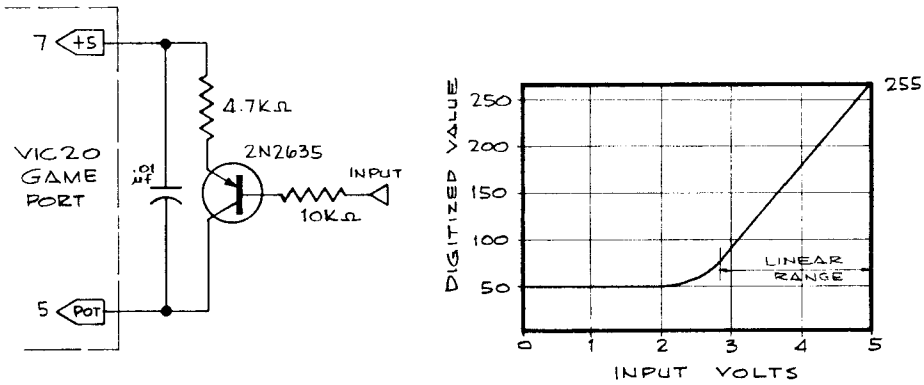
Just as the CdS cell could be used to control the joystick input, it is ideally suited to the game paddle input as well. The added advantage here is that the CdS cell, placed on the game paddle lines, makes the VIC 20 an accurate light meter. The CdS cell is connected between +5V and the game paddle 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 uF capacitor in parallel with the photocell.

Connect the CdS cell as described above, add a relay on one of the joystick 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 in the VIC 20 (TI) 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 pseudoanalog to digital converter. By substituting the circuit shown in Figure 10–6 for the resistance across the game paddle input, the VIC 20 can be made to sense analog voltages in the range of 2 to 5 volts. The PNP

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.



transistor 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 digitized value will vary with the input voltage, E_{in}. The graph in the figure 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 could 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.

11

USING THE RS-232 PORT ON THE VIC 20

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 VIC 20 design, which gives it a tremendous advantage over its predecessor, the PET 2001. This capability allows the VIC 20 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 VIC 20, 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 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, 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 which 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 −12 VDC, and a space is +12 VDC. When Commodore says that the VIC 20 supports RS-232 devices, that is not exactly true. The VIC 20 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 VIC 20'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 to 9600 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 not implemented in the VIC 20, however.

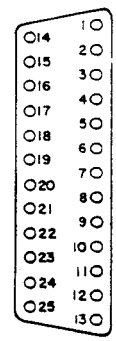
Parity

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 could lead to 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

FIGURE 11-1
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 VIC 20 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	"	



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 VIC 20 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.

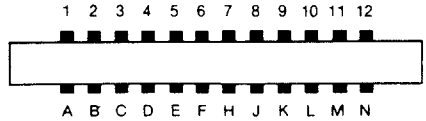
Let's Interface Your Device

As you can see from the description above, there are many options to consider when trying to interface an RS-232 device to the VIC 20. Let's say that you have a printing ASCII terminal that you want to interface to the VIC 20. 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: (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 11-1), 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.

FIGURE 11-2
Pin assignments for the RS-232 line on the VIC 20's user port (Courtesy of Commodore).



PIN #	TYPE	NOTE	PIN #	TYPE	NOTE
1	GND		A	GND	
2	+5V	100mA MAX.	B	CB1	
3	RESET		C	PB0	
4	JOY0		D	PB1	
5	JOY1		E	PB2	
6	JOY2		F	PB3	
7	LIGHT PEN		H	PB4	
8	CASSETTE SWITCH		J	PB5	
9	SERIAL ATN IN		K	PB6	
10	+9V	100mA MAX.	L	PB7	
11	+9V		M	CB2	
12	GND		N	GND	

(6522 DEVICE #1 loc $9110-911F)

PIN ID	6522 ID	DESCRIPTION	EIA	ABV	IN/ OUT	MODES
C	PB0	RECEIVED DATA	(BB)	Sin	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	3
L	PB7	DATA SET READY	(CC)	DSR	IN	2
B	CB1	RECEIVED DATA	(BB)	Sin	IN	1 2
M	CB2	TRANSMITTED DATA	(BA)	Sout	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 (Sin,Sout,GND)
2)—X-LINE INTERFACE (Full handshaking)
3)—USER AVAILABLE ONLY (Unused/unimplemented in code.)
*—These lines are held high during 3-LINE mode.

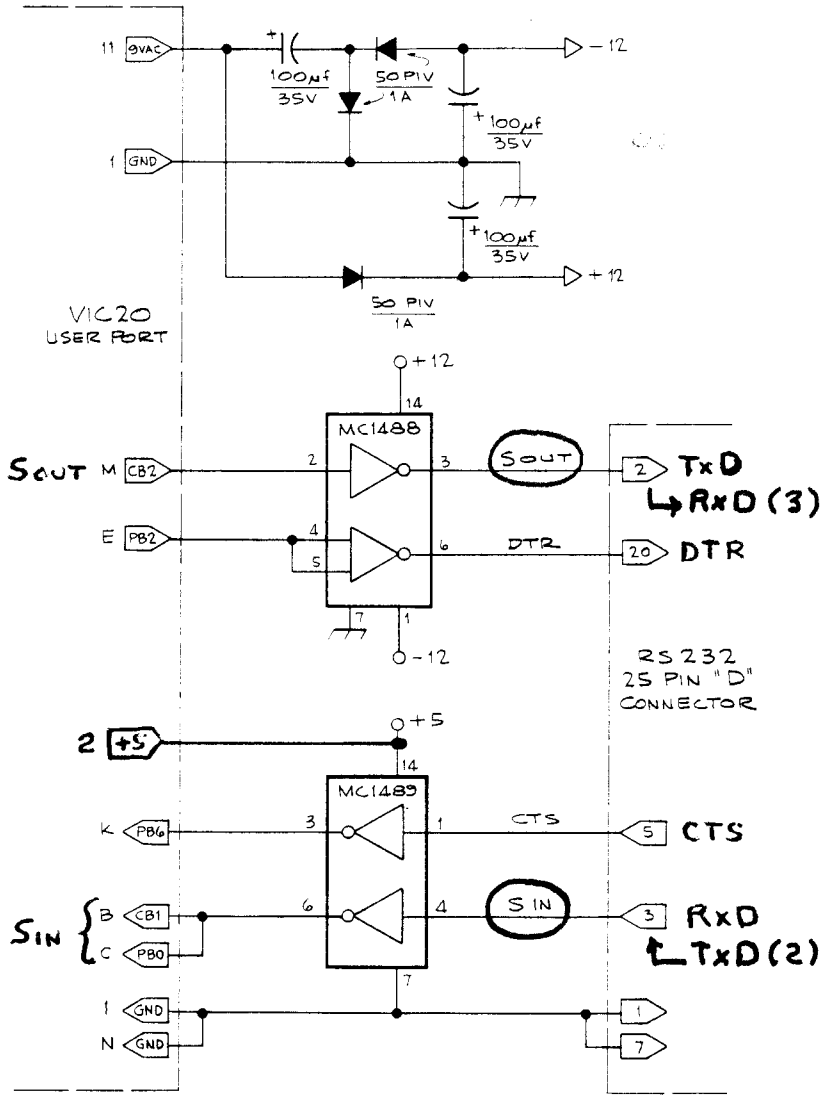
*Note: PB6 CLEAR TO SEND is not implemented and must be read with a short machine language routine.

SEE PAGE 143

20 ma Current Loop

To our knowledge, there are no current loop interfaces commercially available for the VIC 20. Since ASR33 teletypes are readily available and make fine low-cost printers, we will describe a current loop interface for

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



the VIC 20. 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 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 VIC 20. On the other hand, most modems are active and require a passive interface. A quick way to determine your requirement is to put a volt meter 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.

Software Considerations

Assuming that you have a functional interface, it is now time to program it. You must first tell the VIC 20 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 channel. The VIC 20 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

FIGURE 11-4
A 20 ma "active" interface. SEE NEXT PAGE

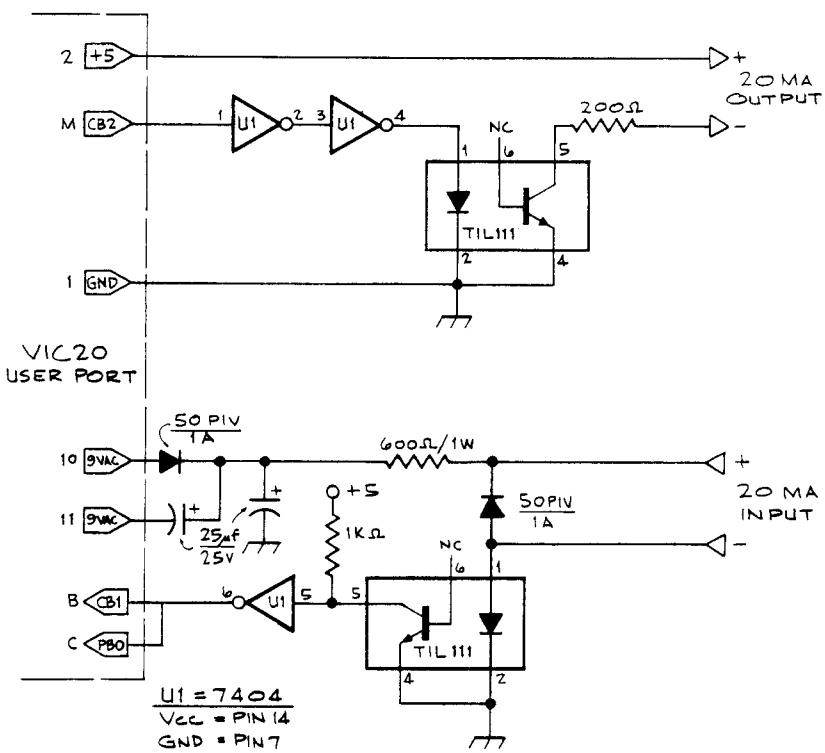
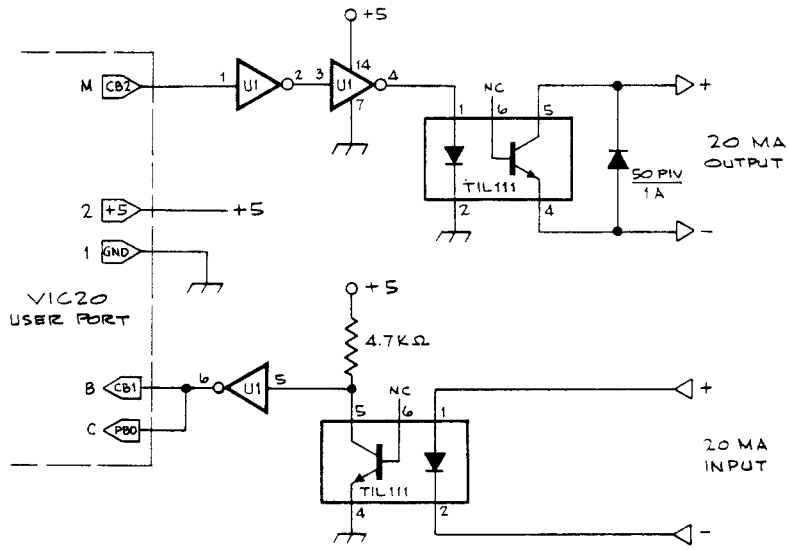


FIGURE 11-5
A 20 ma "passive" interface.



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 VIC 20 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 lf, 2, 0, "[Control Register][Command Register]"
```

If refers to the logical file number. This is any number between 1 and 255, and is arbitrarily assigned by the user. If If is > 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.

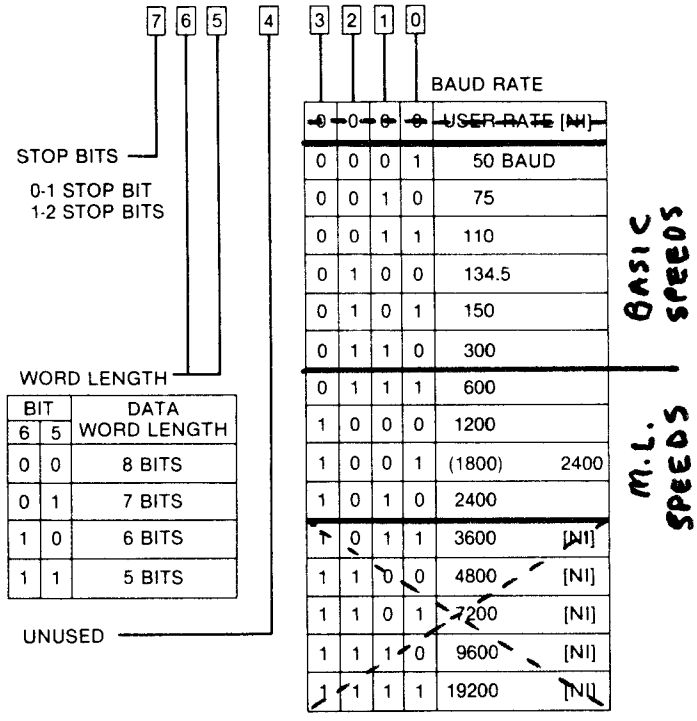
[Control Register] is a single-byte character that configures the format and baud rate of the RS-232 transmission (see Figure 11-6). Note that the quotes indicate that this argument must be a string character.

[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 6 we see that bits 7, 2, and 1 of the control register must be set. The code

```
10000110
```

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

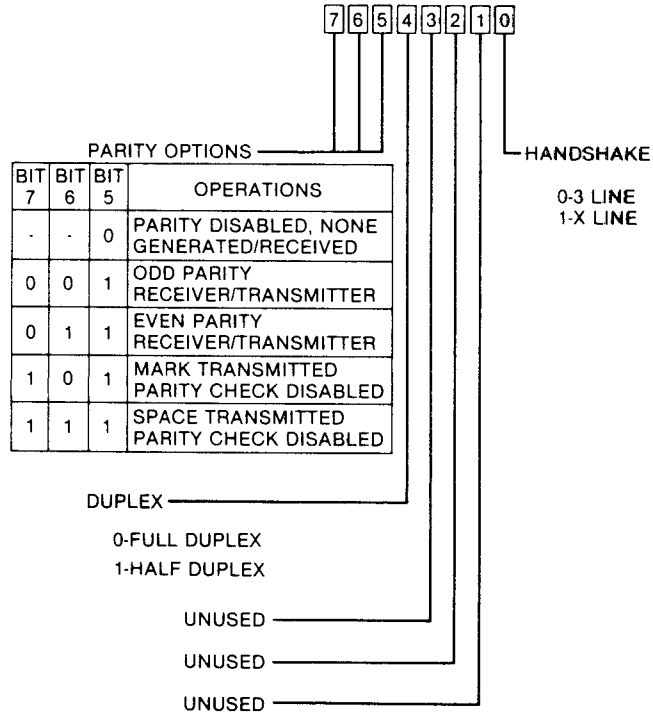
FIGURE 11-6
Bit assignments for the VIC 20's control register (Courtesy of Commodore).



BIT 6	BIT 5	DATA WORD LENGTH
0	0	8 BITS
0	1	7 BITS
1	0	6 BITS
1	1	5 BITS

3	2	1	0	USER RATE (NI)
0	0	0	1	50 BAUD
0	0	1	0	75
0	0	1	1	110
0	1	0	0	134.5
0	1	0	1	150
0	1	1	0	300
0	1	1	1	600
1	0	0	0	1200
1	0	0	1	(1800) 2400
1	0	1	0	2400
1	0	1	1	3600 [NI]
1	1	0	0	4800 [NI]
1	1	0	1	7200 [NI]
1	1	1	0	9600 [NI]
1	1	1	1	19200 [NI]

FIGURE 11-7
Bit assignments for the Vic 20's command register (Courtesy of Commodore).



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

CHR$ (128 + 4 + 2)

The numbers are the result of bit 7, the 128 weight bit; bit 2, the 4 weight bit; and bit 1, the 2 weight bit being set (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 VIC 20 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. It turns out that handshaking is not implemented in the VIC 20 anyway, so this bit should always be 0 (see CTS below).

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.

```
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 VIC 20 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" ($0D) 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 will be for interfacing a serial printer to the VIC 20, 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

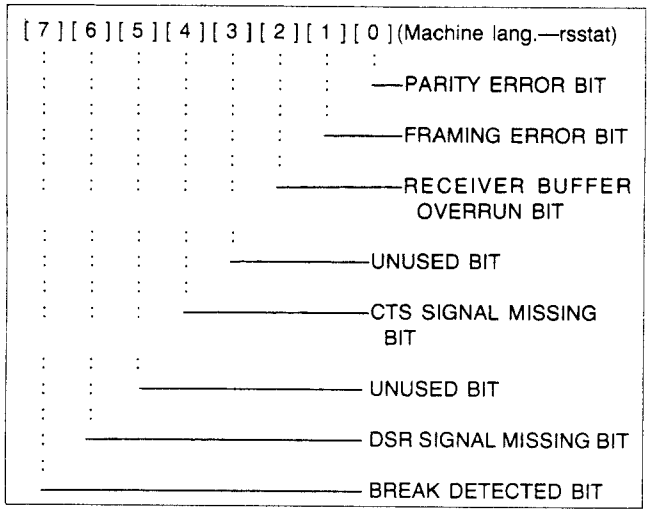
The CLOSE lf command removes the RS-232 device from the system. This is useful because it deallocates the 512 words of buffer and makes it available to the system again. One word of caution, however: If a CLOSE is executed before the output buffer is empty, all data left in the buffer will be lost. BASIC 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 pointers 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 VIC 20 BASIC is reserved for the value of the status register. This register tells 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.

FIGURE 11-8
Bit assignments for the VIC 20's status register (Courtesy of Commodore).



CTS - SEE PAGE 135, 140 (FOR USER/PARALLEL PRINTERS)

The VIC 20 manual states that handshaking is implemented in the VIC 20's RS-232 port. This basically refers to the clear to send line (CTS). That is a line by which a device can signal the VIC 20 that it is not ready for data at that particular time (see *Handshaking,* above). Unfortunately, there is a bug in the VIC 20's software. It reads the wrong VIA chip to check the status of the CTS signal! Thus, CTS does not work. Unfortunately, the CTS line is very important for many printers. In addition to the example discussed earlier there is a second common use of the CTS line. In many printers, the carriage return cannot be accomplished in the time allocated for a single character, and data will be lost if the computer does not delay transmission after sending a carriage return character. The obvious solution to this problem is to use the CTS signal to alert the VIC 20 when the printer is not ready for data. Such printers pull their DTR (data terminal ready) line to a mark condition until their print head is back in position. By connecting the printer's DTR line to the VIC 20 CTS line, this handshake was supposed to have been made.

We have found a way to reimplement the CTS signal. When the VIC 20 outputs a character, it looks to locations $0326 and $0327 for the address of the output routine. It then transfers control to that subroutine, which outputs the character to the specified output device. We cannot change the output routine because it is in ROM, but we can change its RAM vector at $0326 and $0327. We change the vector to point to a

routine either in RAM or, better yet, in a user ROM (see Chapter 8), which monitors the CTS line and then passes control to the VIC 20's output routine.

Listing 11-1 shows the machine language patch. When entered, the routine checks to see if output is to device 2, the RS-232 channel. If so, the patch delays briefly and then loops if either the output buffer still has a character in it or if PB6, the CTS line, is high. When released from the loop, control is passed to the VIC 20's output routine, which allows the

LISTING 11-1
Output handshake.

```
                    1000 * RS-232 OUTPUT HANDSHAKE
                    1010         .OR $1800     OR ANYWHERE
                    1020         .TA $0800
                    1030 *---------------------------------
009A-               1040 DEVICE .EQ $009A
029D-               1050 XMTPTR .EQ $029D
029E-               1060 OUTPTR .EQ $029E
9110-               1070 PORTB  .EQ $9110
F27A-               1080 OUTPUT .EQ $F27A
                    1090 *---------------------------------
                    1100 * TO INITIALLIZE:
                    1110 *   POKE 806,AL:POKE 807,AH
                    1120 *      WHERE AL IS LOW-ORDER ADDRESS OF HANDSH
                    1130 *      AND AH IS HIGH-ORDER.
                    1140 *
                    1150 * DEMONSTRATION BASIC PROGRAM:
                    1160 *    10 OPEN 222,2,0,CHR$(6)+CHR$(0)
                    1170 *    20 FOR I=1 TO 20
                    1180 *    30 PRINT#222,"0123456789"
                    1190 *    40 NEXT
                    1200 *---------------------------------
                    1210 * THIS ROUTINE CAN BE LOCATED ANYWHERE
                    1220 *---------------------------------
1800- 48            1230 HANDSH PHA           SAVE CHAR.
1801- A5 9A         1240        LDA DEVICE
1803- C9 02         1250        CMP #2        DEVICE=2 ?
1805- D0 1F         1260        BNE PULL      ONLY 2 ALLOWED
1807- 8A            1270        TXA           SAVE OTHER REGS
1808- 48            1280        PHA
1809- 98            1290        TYA
180A- 48            1300        PHA
180B- A2 60         1310        LDX #96       ADJUST FOR YOUR BAUD RATE
180D- 88            1320 DELAY  DEY           WAIT UNTIL PREVIOUS
180E- D0 FD         1330        BNE DELAY      CHAR. CAN CHANGE HANDSHAKE
1810- CA            1340        DEX
1811- D0 FA         1350        BNE DELAY
1813- AD 9D 02      1360 WAIT1  LDA XMTPTR
1816- CD 9E 02      1370        CMP OUTPTR    WAIT FOR ZERO CHARS
1819- D0 F8         1380        BNE WAIT1
181B- AD 10 91      1390 WAIT2  LDA PORTB
181E- 29 40         1400        AND #$40      GET BIT 6
1820- D0 F9         1410        BNE WAIT2     WAIT IF HIGH
1822- 68            1420        PLA           RESTORE REGS
1823- A8            1430        TAY
1824- 68            1440        PLA
1825- AA            1450        TAX
1826- 68            1460 PULL   PLA           RESTORE CHARACTER
1827- 4C 7A F2      1470        JMP OUTPUT    DO THE CHARACTER
                    1480 *---------------------------------
```

RS-232 OPERATES VIA THE NMI VECTOR!

character to be placed in the output buffer. The effect is to prevent characters from stacking up the output buffer.

Once the patch is located in memory, it can be implemented by POKEing $0326 (806 decimal) with the low-order byte of the start address of the patch and $0327 (807 decimal) with the high-order byte of the start address. The channel is then opened and closed in the normal way. Caution: A restore will reset the vector and disable the CTS patch.

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 11-2 configures the RS-232 channel so

LISTING 11-2
Input from RS-232 device.

```
                1000 * INPUT FROM RS-232 DEVICE
                1010         .OR $1800
                1020         .TA $0800
                1030 *-------------------------------
0080-           1040 FILNUM .EQ 128       USE FILE# 128 OR HIGHER
00C6-           1050 KEYCNT .EQ $00C6     VIC'S KEY COUNTER
0277-           1060 KEYBUF .EQ $0277     START OF KEY BUFFER
0314-           1070 IRQVEC .EQ $0314     RAM HOOK FOR IRQ
EABF-           1080 SVCIRQ .EQ $EABF     ROM IRQ SERVICE ROUTINE
F14F-           1090 GETRS  .EQ $F14F     ROM ROUTINE TO GET CHAR FROM RS-232
FFC6-           1100 SET.IO .EQ $FFC6     KERNAL ROUTINE
                1110 *-------------------------------
                1120 * TO INITIALLIZE, USE THIS BASIC LINE:
                1130 *    OPEN 128,2,0,CHR$(  ):SYS X:GET#128,A$
                1140 *      WHERE X IS LOCATION OF INIT ROUTINE
                1150 *      AND CHR$(  ) CONTAINS YOUR PARAMETERS
                1160 *-------------------------------
1800- 08        1170 INIT   PHP           SAVE IRQ MASK
1801- 78        1180        SEI           NO IRQ'S FOR A FEW USEC.
1802- A9 13     1190        LDA #PATCH    LOW ORDER ADDR OF PATCH
1804- 8D 14 03  1200        STA IRQVEC    RE-VECTOR
1807- A9 18     1210        LDA /PATCH    HIGH ORDER ADDR
1809- 8D 15 03  1220        STA IRQVEC+1
180C- A2 80     1230        LDX #FILNUM
180E- 20 C6 FF  1240        JSR SET.IO    TELL SYSTEM ABOUT IT
1811- 28        1250        PLP           RESTORE OLD IRQ STATUS
1812- 60        1260        RTS
                1270 *-------------------------------
                1280 * PATCH IS ALL RELOCATABLE CODE
                1290 *-------------------------------
1813- 48        1300 PATCH  PHA           SAVE REGS. WE WILL USE
1814- 98        1310        TYA
1815- 48        1320        PHA
1816- 20 4F F1  1330        JSR GETRS     GET CHAR FROM RS-232 BUFFER
1819- A8        1340        TAY           TEST IT FOR ZERO
181A- F0 0B     1350        BEQ RTRN      RETURN IF ZERO
181C- C9 03     1360        CMP #$03      END OF TRANSMISSION
181E- F0 0D     1370        BEQ END
1820- 8D 77 02  1380        STA KEYBUF    CHAR. TO KEY BUFFER
```

```
1823- A9 01     1390         LDA #1         ONE CHAR.
1825- 85 C6     1400         STA KEYCNT     TELL SYS THERE IS ONE
1827- 68        1410 RTRN    PLA            RESTORE WHAT WE USED
1828- A8        1420         TAY
1829- 68        1430         PLA
182A- 4C BF EA  1440         JMP SVCIRQ     CONTINUE NORMAL IRQ SERVICE
182D- A9 BF     1450 END     LDA #SVCIRQ    RESTORE ORIGINAL IRQ VECTOR
182F- 8D 14 03  1460         STA IRQVEC
1832- A9 EA     1470         LDA /SVCIRQ
1834- 8D 15 03  1480         STA IRQVEC+1
1837- D0 EE     1490         BNE RTRN       FORCED
                1500 *--------------------------------
```

that it operates in parallel with the VIC 20's keyboard. Any input on the RS-232 channel will cause the VIC 20 to operate exactly as if the corresponding key on the VIC 20 were pressed.

The interrupt vector, locations $0314 and $0315, is changed to point to the patch. One of the major functions of the 60 Hz interrupt in the VIC 20 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 should be located at the top of memory or in a ROM, if you plan to use it often (see Chapter 8). To initialize the program, you must do a SYS command to the address of INIT.

Listing 11-3 is a BASIC program that opens the channel, finds the

LISTING 11-3 Input from RS-232 device.

```
5 REM MAKES RS232 THE  SYSTEM INPUT DEVICE
10 OPEN 128,2 ,0,CHR$(163)+CHR$(160)
20 H=PEEK(56)
30 L=PEEK(55)
40 MT=256*H+L
50 PRINT MT
60 MT=MT-53
70 H=INT (MT/256)
80 L=MT-256*H
90 POKE 55,L
100 POKE 56,H
110 MT=MT+1
120 FOR I=MT TO MT+51
130 READ X
140 POKE I,X
150 NEXT I
160 DATA 8,120,169,14,141,20,3,169,24,141,21
165 DATA 3,40,96,72,152,72,32
170 DATA 79,241,168,240,11,201,3,240,13,141
175 DATA 119,2,169,1,133,198
180 DATA 104,168,104,76,191,234,169,191,141
190 DATA 20,3,169,234,141,21,3,208,238
200 IR=MT+14
210 H=INT (IR/256)
220 L=IR-H*256
230 POKE MT+3,L
240 POKE MT+8,H
250 SYS MT
260 GET#128,A$
270 NEW
```

top of memory, POKEs the patch into protected space at the top of memory, 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 you SAVE the loaded program on tape, the VIC 20 does a warm start and restores the interrupt vector. Thus you must reload the patch before another program can be received. 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 VIC 20 as a Dumb Terminal

A good use of the VIC 20'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 the key presses over the RS-232 output. Such terminals are usually used in communications networks and to access large computer systems. The BASIC programs in Listings 11-4 and 11-5 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 VIC 20. Thus the appearance of your keypresses on the monitor verifies that the remote machine has, indeed, received the transmission properly. In half-duplex mode, the VIC 20 is expected to display its own keypresses. Listing 11-4 is for a full-duplex connection, and 11-5 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, which is also suitable.

LISITING 11-4.

```
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 11-5.

```
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
```

12

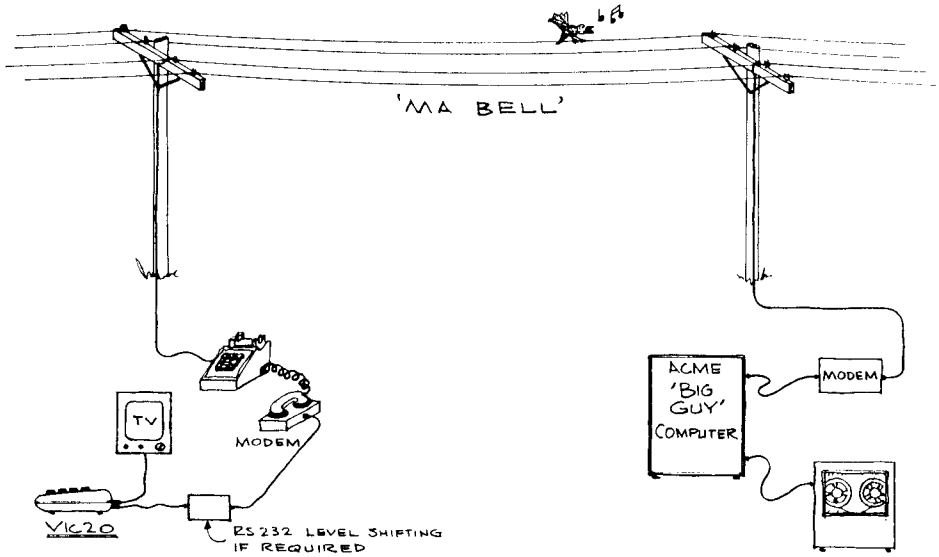
MODEMS AND THE VIC 20

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 VIC 20.

The purpose of the modem (modulator demodulator) between the telephone line and the computer or terminal is to generate and receive audio tones 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 VIC 20, with its built-in RS-232 capability, can easily accommodate a modem. Virtually *any commerical RS-232 modem can be used but will require the proper level shifting adaptation described earlier in Chapter 11*, because the VIC 20 signal levels at the connector are TTL. Commodore also makes a compatible modem specially designed for the VIC 20 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 VIC 20 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 key presses 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, which 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 allows 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 the phone 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" 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" 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" 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" x 5½" perforated experimenter card and attaches to the VIC 20 with the 22 44 to 12 24 adapter plug introduced in Chapter 3. Begin construction with the three power supplies. We

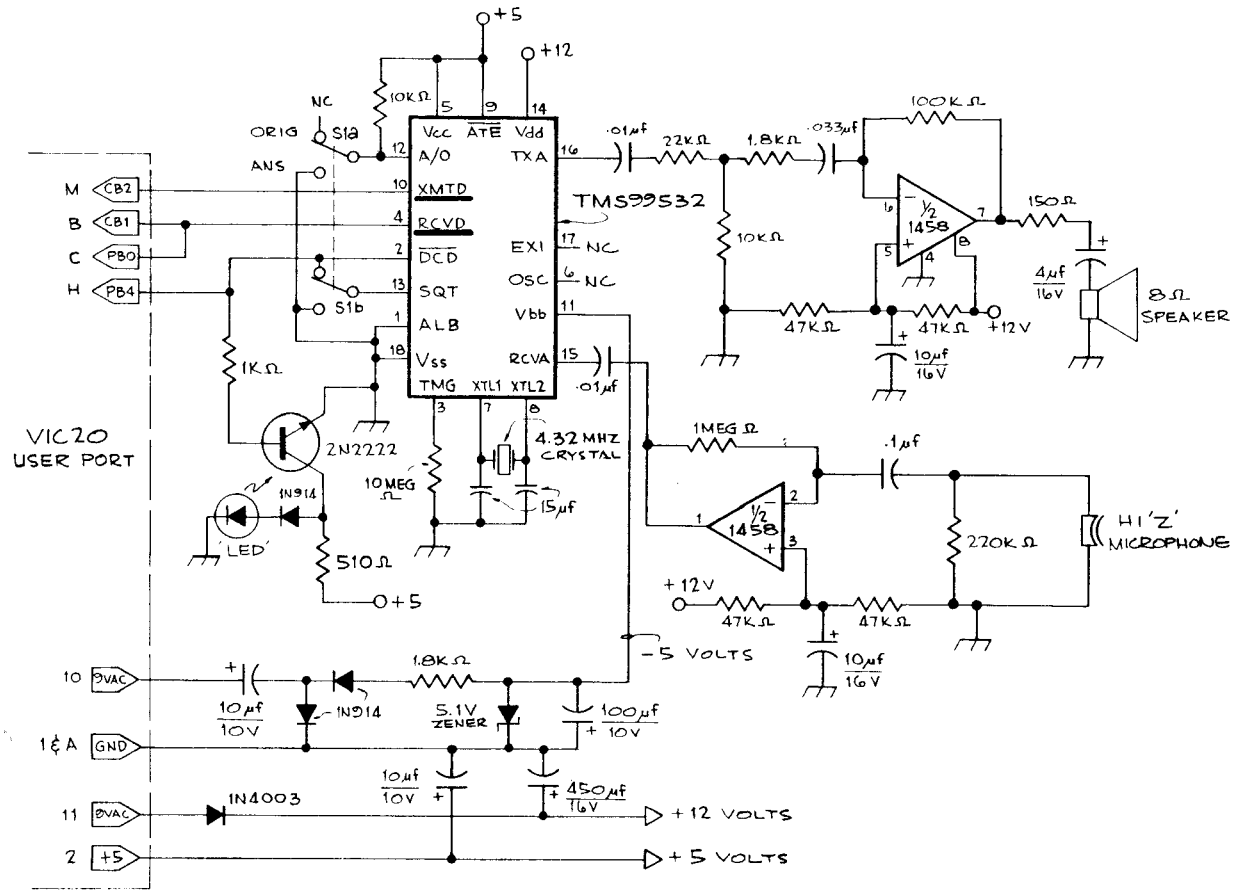
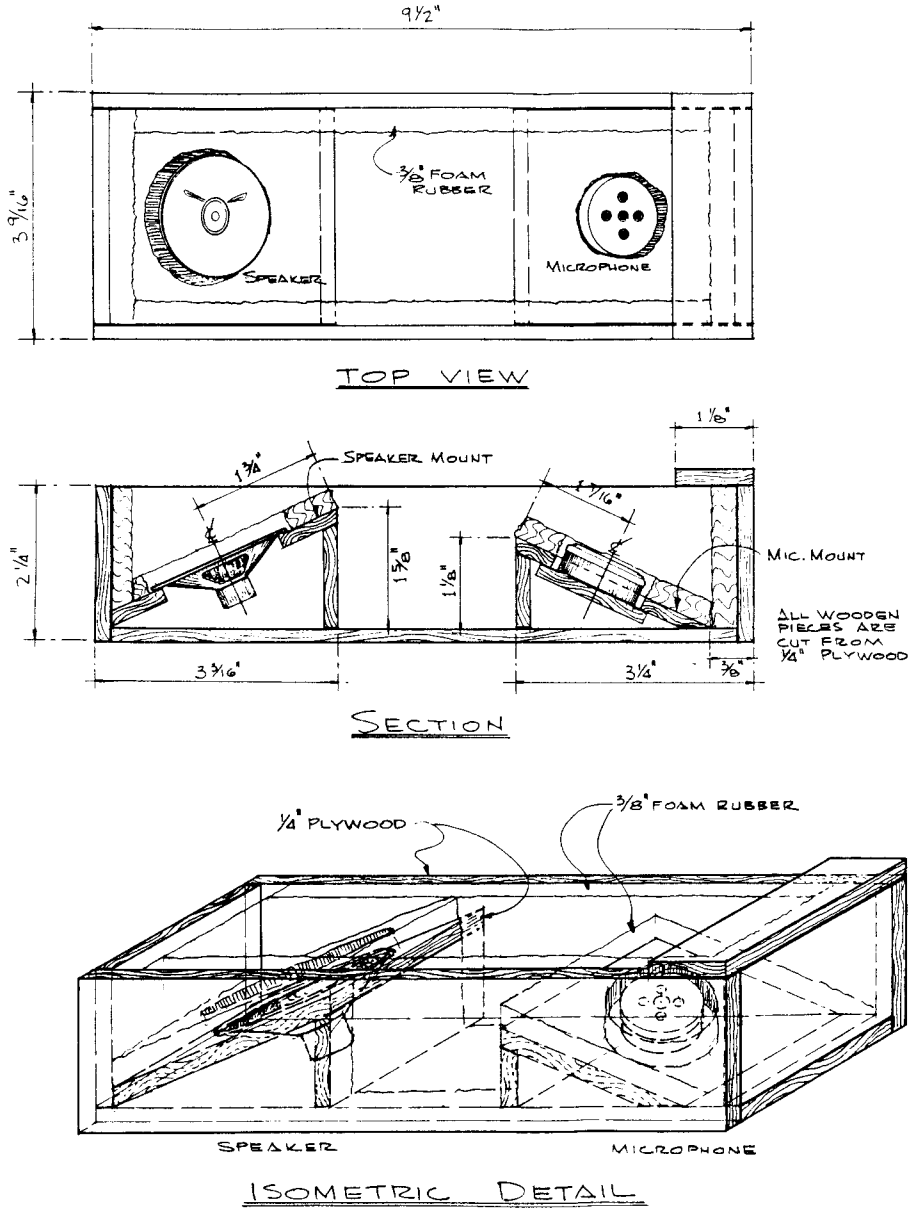


FIGURE 12-2
Schematic for modem.

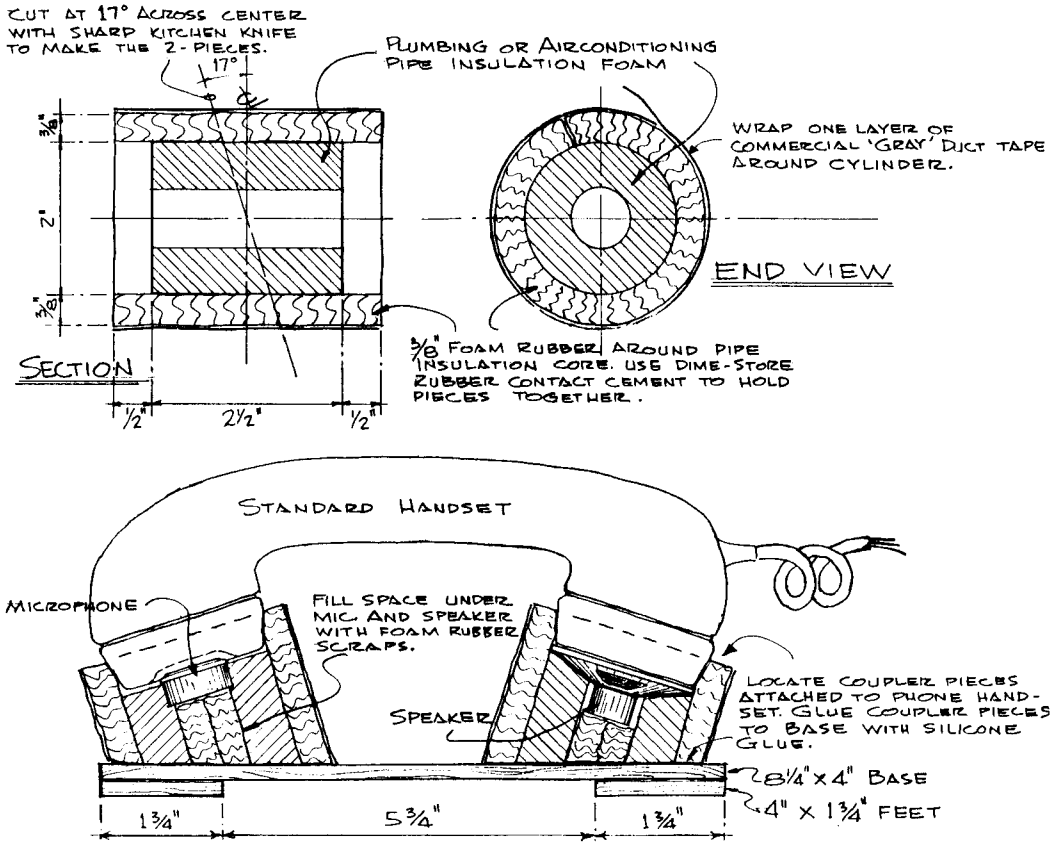
FIGURE 12-3
Dimensions for a cradle for a Princess-type telephone handset



suggest that you test for the correct voltages with the card connected to the VIC and not complete the modem circuit until all three supplies are working.

Epoxy glue the 8, 16, and 18 pin sockets and the answer originate switch to the card. Mount the remaining parts as you wish, keeping the

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



amplifier components close to the 1458. Wire up the components, but leave the ICs out of their sockets for testing.

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 VIC 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 ohm meter. 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.

TABLE 12-1: IC Socket Voltage Chart for Modem

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

```
10 OPEN 2,2,0,CHR$(161)+CHR$(160)   REM Initialize
                                    RS-232 for 50 baud
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 VIC 20 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 is Listing 12-1 has such an initialization routine. It is

LISTING 12-1
Terminal routine.

```
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$(13)
615 LF=0 :IF CH>127 THEN LF=1
620 IF RU<>127 THEN RO$=CHR$(RU)
630 PRINT"↵      ■INITILIZED■!!"
640 PRINT:PRINT"GO- ";
645 REM***START COMMUNICATIONS ROUTINE*
650 GOTO 20
READY.
```

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. Refer back to Chapter 11 for several machine language programs and other ideas for enhancing this program.

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
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 TMS99532 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 deg. C), (+/−.005% temp. stable), (+/−.005% freq. tol.), (22 pf loading), (70 ohm series res.), (2 mw drive level), (Fundamental mode)
Available for $8.00 postpaid from The Bit Stop, 5958 South Shenandoah Road, 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 1N914 diode
1 1N4002 rectifier diode
1 2N2222 NPN transistor
1 5.1 volt zener diode
1 green LED
1 150 ohm ¼ watt resistor
1 510 ohm ¼ watt resistor
1 1000 ohm ¼ watt resistor
1 1800 ohm ¼ watt resistor
1 10,000 ohm ¼ watt resistor
1 22,000 ohm ¼ watt resistor
4 47,000 ohm ¼ watt resistor
1 100,000 ohm ¼ watt resistor
1 220,000 ohm ¼ watt resistor
1 1 meg ohm ¼ watt resistor
1 10 meg ohm ¼ watt resistor
2 15 pf 25V ceramic capacitors
3 .01 mfd 50V Mylar capacitors
1 .033 mfd 50V Mylar capacitors
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 square feet ¼" plywood
60 inches small-diameter shielded cable
1 square foot 3/8" thick foam rubber

APPENDIX: SCREEN CODES FOR THE VIC 20

The character set for the VIC 20 differs somewhat from that for the ASCII standard. Most of the differences are in the additional graphics characters. In this section, we present a detailed printout of the characters and their codes. This information can be very useful, both in graphics programming and in finding appropriate key presses to use as arguments in an OPEN statement to replace complicated CHR$ arguments.

0		64	@	128		192	—
1		65	A	129		193	♠
2		66	B	130		194	|
3		67	C	131		195	—
4		68	D	132		196	—
5	WHITE	69	E	133	F1	197	—
6		70	F	134	F3	198	—
7		71	G	135	F5	199	|
8	NO C=	72	H	136	F7	200	|
9	OK C=	73	I	137	F2	201	╮
10		74	J	138	F4	202	╰
11		75	K	139	F6	203	╯
12		76	L	140	F8	204	╭
13	RETRN	77	M	141	RETRN	205	╲
14	L.C.	78	N	142	U.C.	206	╱
15		79	O	143		207	╭
16		80	P	144	BLACK	208	┐
17	DOWN	81	Q	145	C UP	209	●
18	REVRS	82	R	146	RVOFF	210	—
19	HOME	83	S	147	CLR	211	♥

20	DEL		84	T	t	148	INST		212		T
21			85	U	u	149			213		U
22			86	V	v	150			214		V
23			87	W	w	151			215		W
24			88	X	x	152			216		X
25			89	Y	y	153			217		Y
26			90	Z	z	154			218		Z
27			91	[	[	155			219		
28	RED		92	£	£	156	PURPL		220		
29	RIGHT		93	]	]	157	LEFT		221		
30	GREEN		94	↑	↑	158	YELLO		222		
31	BLUE		95	←	←	159	CYAN		223		
32	SPACE		96	—	—	160	SPACE		224	SPACE	
33	!	!	97	♠	A	161			225		
34	"	"	98		B	162			226		
35	#	#	99		C	163			227		
36	$	$	100		D	164			228		
37	%	%	101		E	165			229		
38	&	&	102		F	166			230		
39	'	'	103		G	167			231		
40	(	(	104		H	168			232		
41	)	)	105		I	169			233		
42	*	*	106		J	170			234		
43	+	+	107		K	171			235		
44	,	,	108		L	172			236		
45	-	-	109		M	173			237		
46	.	.	110		N	174			238		
47	/	/	111		O	175			239		
48	0	0	112		P	176			240		
49	1	1	113	●	Q	177			241		
50	2	2	114		R	178			242		
51	3	3	115	♥	S	179			243		
52	4	4	116		T	180			244		
53	5	5	117		U	181			245		
54	6	6	118		V	182			246		
55	7	7	119		W	183			247		
56	8	8	120		X	184			248		
57	9	9	121		Y	185			249		
58	:	:	122	◆	Z	186			250		
59	;	;	123	+	+	187			251		
60	<	<	124			188			252		
61	=	=	125			189			253		
62	>	>	126			190			254		
63	?	?	127			191			255		

INDEX

Computers

Make your Vic 20 do more for you!

This clearly written guide provides dozens of *simple, useful* interfacing projects *specifically designed to maximize the big power inside your little Vic 20.* Each project comes with a complete listing of BASIC programs to run so that you can quickly tap into the Vic 20's built-in hardware without knowing machine language. 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 Vic 20 and this handbook:

- a speech synthesizer
- a light sensor suitable for a burglar alarm
- mechanical acuators
- a telephone modem
- analog-to-digital converters
- printer interfaces
- how to use a standard 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.

Don 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.

PRENTICE-HALL, Inc., Englewood Cliffs, New Jersey 07632

Cover design by Hal Siegel

0 21898 22342 5

ISBN 0-13-223421-