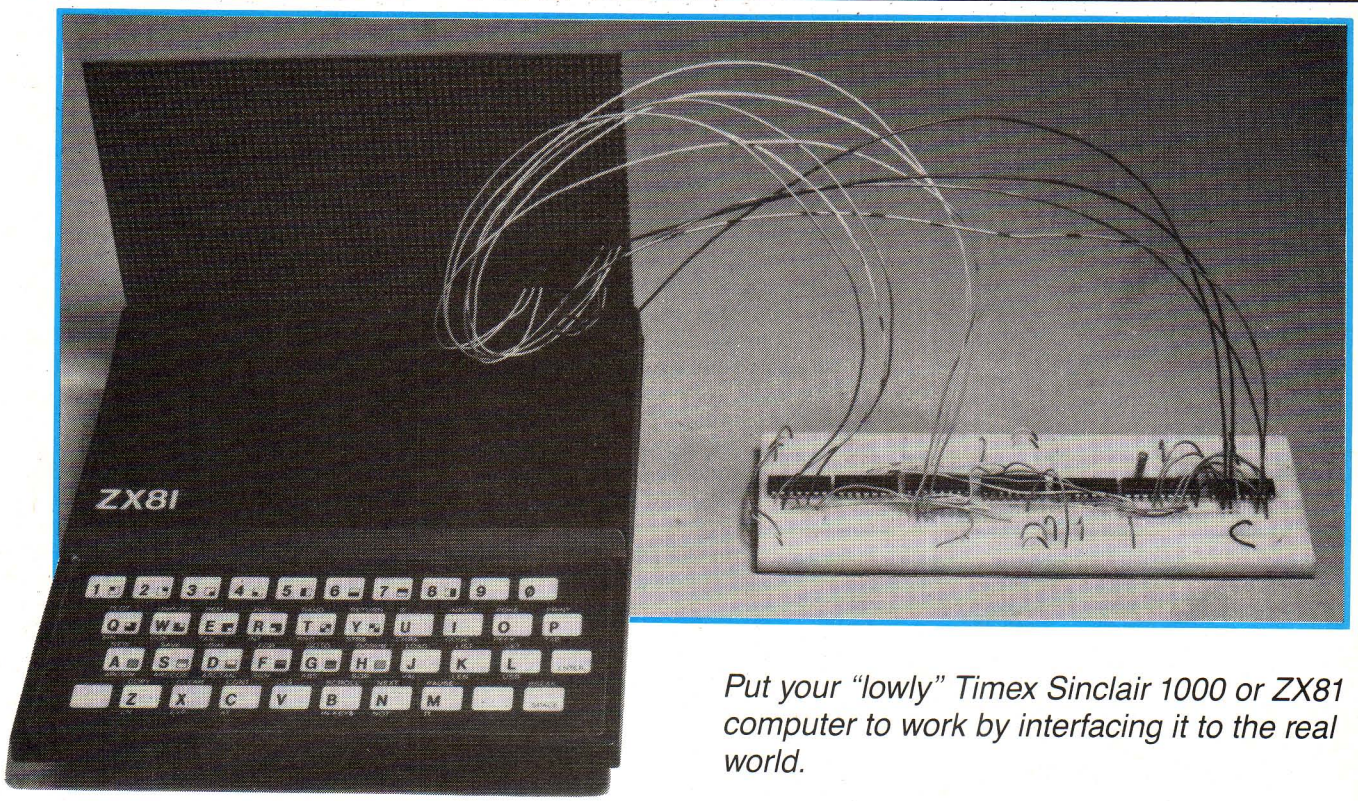


Put your "lowly" Timex Sinclair 1000 or ZX81 computer to work by interfacing it to the real world.

Interfacing the ZX81

NEIL BUNGARD

THE SINCLAIR *ZX81* (OR TIMEX SINCLAIR *1000*) was the first computer to be sold for less than $100. It was quite a bargain at that price. Perhaps you're one of the many people who bought one so that you (or your kids) could learn something about computers. Or perhaps you've seen the computer on sale for a very attractive price and have only recently considered buying it just to see what it was like.

If you did buy the machine when it first became available, you may have outgrown it by now. After all, it's not the most practical computer available. Some of its problems—including its membrane keyboard and wobbling 16K RAM extension—can make using the computer almost a chore.

However, the *ZX81* is unbeatable for the hobbyist who wants to experiment—as we will do—with computers, computer interfacing, and Z80 machine language. And now that the computer is no longer being produced, you can often find it selling for as little as $30 (and maybe even less).

Let us note that for the remainder of the article, we'll refer to the Timex-Sinclair *1000* as the *ZX81*, if for no other reason than because it's shorter. There is little difference between the two machines, save that the *1000* has 2K of built-in RAM as compared to the *ZX81*'s 1K.

What we'll do

In this article, we'll show you how to interface the *ZX81* to a clock/calendar integrated circuit, a temperature indicator, a light controller, and a heater controller. You might find use for such devices in photographic dark rooms, home weather stations, and home security systems.

In addition to interfacing those special devices to the *ZX81*, hardware and software interfacing principles will be covered in general. That means that you can make use of the inexpensive *ZX81* for any particular monitor or control application that you have in mind—you won't be limited to the specific applications that we'll discuss. It's true that we won't end up with something as elegant as the home-control computer that was presented in **Radio-Electronics** in April through June. But we will have something to quiet the people who scoff at the *ZX81* and call it a useless machine.

Using a computer for monitoring and control has become much simpler than it once was because of the number of interesting microprocessor-compatible integrated-circuits now available. For example, we'll be using OKI's MSM5832 clock/calendar integrated-circuit (IC).

Where to begin

You will, of course, need a *ZX81* computer and a TV. You should also have a standard cassette-tape player so that you can store your programs. You might also consider building the 8K non-volatile RAM accessory that was described in the July and August 1983 issues of **Radio-Electronics**: It makes storing and using machine code much more convenient.

On the back of the *ZX81* is a card-edge with 44 "fingers." That gives you access to (for all practical purposes) all of the pins of the Z80 microprocessor. Figure 1 shows the pinout of the card edge, while the signals and functions are listed in Table 1. When you interface to the port, you are essentially interfacing directly to the Z80 CPU. To get the most out of the project, you might consider obtaining a Z80 technical manual and other reference material relating to the Z80.

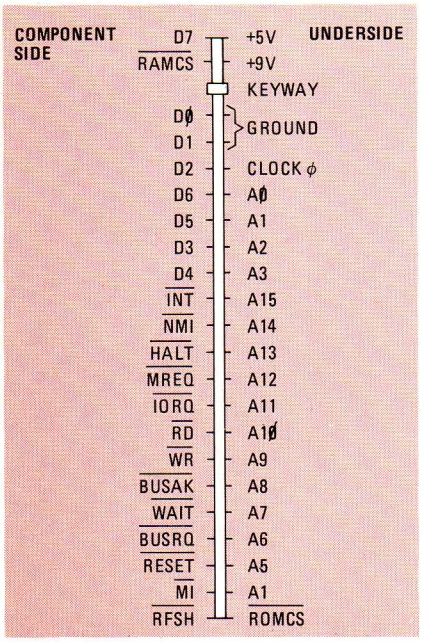


FIG. 1—THE CARD-EDGE PINOUT of the *ZX81*'s expansion port.

sequently doing any type of memory-mapped input/output (I/O) is practically impossible. That leaves you with only the Z80's IN and OUT instructions for getting peripheral data in and out of the microcomputer. Unfortunately, the *ZX81* does not include IN and OUT commands with its BASIC language instructions. That means that all I/O will have to be accomplished from machine-language subroutines that will be incorporated within the BASIC program you write. Fortunately the *ZX81*'s BASIC language does supply you that capability with the "USR" command.

When using the IN and OUT instructions with the *ZX81*, some strange things occur in two specific situations: 1) if you use the OUT machine language instruction with an odd device code (in other words, if A0 = 1), the *ZX81* monitor system will crash. 2) Any input using the IN machine language instruction and an even device code ignores the top two data bits (D6 and D7). As long as those characteristics (that are unique to the *ZX81*) are known, they can be circumvented. They will not be a problem for our applications. If one is not aware of those characteristics, however, they can be maddening and take weeks to figure out.

TABLE 1—TIMEX/SINCLAIR EXPANSION PORT

Description	Signal	Function
Address bus	A0 - A15	Outputs memory-and I/O device addresses
Data bus	D0 - D7	Transmits bidirectional data into/out of CPU
System control	$\overline{MREQ}$	Identifies any memory-access in progress
	$\overline{IORQ}$	Identifies any I/O operation in progress
	$\overline{RD}$	Indicates that the CPU wants to read data
	$\overline{WR}$	Indicates that the CPU wants to output (write) data
	$\overline{MI}$	Identifies the op-code fetch cycle of instruction Also used with IORQ to acknowledge interrupt
	$\overline{RFSH}$	Synchronizes dynamic-memory refresh
System clock	φ	3.25 MHz clock (output)
	$\overline{RESET}$	Resets CPU when pulled low
	$\overline{INT}$	Interrupt request input
CPU control	$\overline{NMI}$	Interrupt request input; cannot be disabled
	$\overline{WAIT}$	Initiates wait state in machine cycle
	$\overline{HALT}$	Indicates CPU has executed a HALT instruction
Bus control	$\overline{BUSRQ}$	Request to CPU for control
	$\overline{BUSAK}$	Acknowledgement of release of control by CPU
RAM/ROM select	$\overline{RAMCS}$	If pulled high will disable computer's on board RAM
	$\overline{ROMCS}$	If pulled high will disable computer's ROM
Power	+9V	Unregulated
	+5V	Regulated
	GROUND	

Interfacing principles

The procedures for interfacing to a microcomputer are relatively straightforward. Once you learn the techniques, you'll be able to apply what you learn here to other computers. However, the *ZX81* has some unique idiosyncrasies that can give you a real headache unless you are forewarned. We will list those idiosyncrasies now for those that have done some microprocessor interfacing. If you don't understand them now, don't worry—we'll explain later them in more detail when we review general interfacing principles.

The first thing that makes interfacing with the *ZX81* not as straightforward as interfacing with just the Z80 CPU is that the *ZX81* does not use absolute address decoding in its memory scheme. Con-

Interfacing basics

Interfacing to a microcomputer can be analogous to tossing a baseball. If you try to catch the ball before it is within your reach—or after it has gone past—you will miss it. Data moving in and out of the microprocessor works much the same way. When the microprocessor wishes to input or output data, it always does three operations:

- It places the address of the device with which it wishes to communicate on the eight lower bits of the address bus.
- It places the data it wishes to send to (or receive from) the external device on the data bus.
- After the address and data buses have had time to settle, it sends a control pulse to initiate the data transfer.

The microprocessor does those three operations every time it wishes to communicate. But it is your responsibility to perform the following tasks in response:

- You must decode the address from the address bus and signal the device that its particular address is being sent.
- You must be certain that the communication lines of the device are properly connected to the data bus, and if it is an input device, its communication lines must be disabled (set to a high-impedance) until it receives an input-control pulse from the microcomputer.
- When interfacing to the Z80 CPU, you must decode three control signals to generate the input and output control pulses. Once decoded, you must send the appropriate control pulse to the device to initiate data transfer.

The interface circuit

The schematic of the interface circuit is shown in Fig. 2. Let's take a look at it to see how the principles that we just mentioned apply. Two sections of a quad OR gate, IC1-a and IC1-b, receive three control signals from the microcomputer on its inputs. It decodes those three signals, such that when $\overline{IORQ}$ and $\overline{WR}$ are true (low) an $\overline{OUT}$ control pulse is generated. When $\overline{IORQ}$ and $\overline{RD}$ are true (low) an $\overline{IN}$ control pulse is generated. Those input- and output-control pulses initiate the data transfers. Each of the control pulses goes to a 74LS138 3-to-8-line decoder (IC3 and IC4). Figure 3 shows how the 74LS138 operates: When $\overline{E1}$ and $\overline{E2}$ are high (logic 1) the logic levels of inputs A, B, and C do not matter. However, if $\overline{E1}$ and $\overline{E2}$ are low (which is the condition when either $\overline{IN}$ or $\overline{OUT}$ is true), the inputs A, B, and C are decoded such that one of the outputs on the 74LS138 goes low. As shown in the table in Fig. 3, the output that goes low depends on the code at the inputs A, B, and C (pins 1, 2, and 3).

Referring back to the schematic in Fig. 2, you will notice that the 74LS138 inputs are connected to A2, A3 and A4 of the *ZX81* address bus. This means that—assuming that A0, A1, A5, A6 and A7 are all logic zeros—the device codes as, shown in Table 2, can be generated by the *ZX81* and will be decoded by the 74LS138.

So with two OR gates and two 74LS138's you have the capability of generating eight output- and eight input-device code pulses. In other words, you have the capability of sending an output to eight separate devices from the *ZX81* and receiving an input from eight separate devices. In the interface circuit in Fig. 2, three of the output-device code pulses (IC4, pins 13, 14 and 15) and one of the

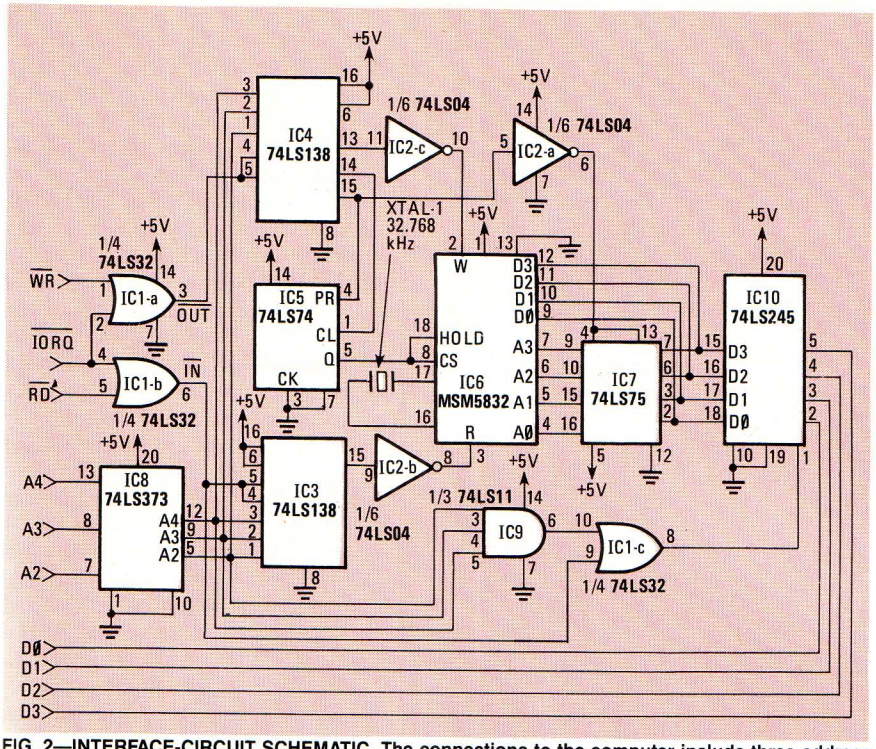
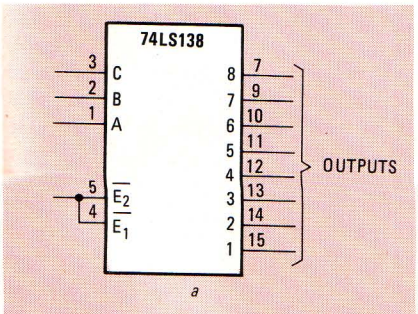


FIG. 2—INTERFACE-CIRCUIT SCHEMATIC. The connections to the computer include three address lines, three control lines, and four data lines. Note that +5 volts can also be taken from the computer.



a

$\overline{E_1}$ AND $\overline{E_2}$	C B A	OUTPUTS 1 2 3 4 5 6 7 8
1	X X X	1 1 1 1 1 1 1 1
0	0 0 0	0 1 1 1 1 1 1 1
0	0 0 1	1 0 1 1 1 1 1 1
0	0 1 0	1 1 0 1 1 1 1 1
0	0 1 1	1 1 1 0 1 1 1 1
0	1 0 0	1 1 1 1 0 1 1 1
0	1 0 1	1 1 1 1 1 0 1 1
0	1 1 0	1 1 1 1 1 1 0 1
0	1 1 1	1 1 1 1 1 1 1 0

b

FIG. 3—THE 74LS138 3-to-8-line decoder/multiplexer. By using three address lines and two control lines, we can generate 8 input- and 8 output-device codes.

input-device code pulses (IC3 pin 15) are used. The remaining input and output device code pulses will be used in future interface projects.

Once the device-code pulses are generated via the input and output 74LS138's, they are sent to the clock/calendar circuit to get information into and out of the MSM5832 clock/calendar IC.

The remainder of the circuit in Fig. 2 is specifically for interfacing the MSM5832 to the *ZX81*. For example, when output-device code pulse 00H is generated (at IC4, pin 15), it goes to pin 4 of IC5, a 74LS74 flip-flop. (**Note:** A hexidecimal number is indicated by a capital "H" following the digits.) When the flip-flop receives a low-going pulse on pin 4, it sets its output (pin 5)—and thus the CS and HOLD inputs of the MSM5832 clock/calendar IC—to a logic 1. A logic 1 on those inputs selects the MSM5832 and holds all of its internal registers stable during the input or output operation.

TABLE 2

A4	A3	A2	Device Code (Hex)
0	0	0	00
0	0	1	04
0	1	0	08
0	1	1	0C
1	0	0	10
1	0	1	14
1	1	0	18
1	1	1	1C

Note: A7, A6, A5, A1, and A0 = 0

In addition to going to pin 4 of the 74LS74, the device-code pulse is inverted and sent to pins 4 and 13 of a IC7, a 74LS75 4-bit latch. When the 74LS75 receives a high-going pulse on pins 4 and 13, it catches the data present on its inputs (pins 2, 3, 6, and 7). Those input pins are connected (through a buffer) to the four lower-order bits of the data bus of the *ZX81*.

The information transferred from the *ZX81* to the latch is the code or address for the MSM5832 clock/calendar register to which it wishes to send or receive information. A listing of the register options and their codes are shown in Table 3. Once the register code is sent to the 74LS75, it will remain present on its outputs until a new register code is sent.

PARTS LIST

IC1—74LS32 quad OR gate
IC2—74LS04 hex inverter
IC3,IC4—74LS138 3-to-8-line decoder/multiplexer
IC5—74LS74 dual D-type flip-flop
IC6—MSM5832 microprocessor real-time clock/calendar
IC7—74LS75 4-bit bistable latch
IC8—74LS373 octal D-type latch
IC9—74LS11 triple 3-input AND gate
IC10—74LS245 octal bus transceiver
XTAL1—32.768 kHz

Miscellaneous: Card-edge connector for *ZX81*, wire-wrap sockets, universal plug-board, etc.

The following are available from Active Electronics, PO Box 8000, Westborough, MA 01581 (1-800-343-0874): MSM5832 clock/calendar IC, $12.95; 32.7680 kHz crystal, $3.45. Add $3.75 for postage, handling, and insurance.

Once the register code is sent, the MSM5832 is ready for communication and data can be sent to the selected register by generating an output-device code pulse 08H (pin 13, IC4). You may receive information from the selected register by generating an input-device code pulse 00H (pin 15, IC3). When you have completed your input or output operations, an output-device code pulse 04H (pin 14, IC4) is generated to reset the 74LS75 flip-flop output back to a logic zero. That deselects the MSM5832 and allows it to resume normal timing operations—which brings us to an important point: If the MSM5832 is kept on hold (CS and HOLD set to logic 1) for more than 1 second, its normal timing will be interrupted and you will lose a second for each second that those pins are held to a logic 1.

Two IC's in the interface, IC8 and IC10, have not been discussed, yet they are important. Those two IC's are bus buffers—they protect the *ZX81* from damage resulting from wiring errors in the interface circuit. For example, IC8 (74LS373) buffers the address lines and is unidirectional, while IC10 (74LS245) buffers the data-bus lines and is bidirectional. The 74LS245 allows information to flow from the *ZX81* data bus to the interface circuit on all OUT instructions. But it only allows information to flow from the interface circuit to the *ZX81* on certain IN instructions. Those IN instructions are determined by the IN address lines A2 through A4, and a decoder circuit constructed of IC9 (74LS11) and IC1-c (74LS32).

If any address line is a logic 0 and an IN is generated, the 74LS245 allows information to flow from the interface circuit to the *ZX81* data bus. That means that if the *ZX81* is trying to generate an IN instruction with A2, A3, and A4 all logic 1's, an input will not be accomplished.

TABLE 3—MSM5832 REGISTERS

Address Inputs A3	A2	A1	AØ	Internal Counter	Data I/O D3	D2	D1	DØ	Data Limits	*Notes
Ø	Ø	Ø	Ø	Seconds	X	X	X	X	Ø–9	Seconds and tens of seconds are reset to zero when write instruction is executed with address selection
Ø	Ø	Ø	1	Tens of seconds	—	X	X	X	Ø–5	
Ø	Ø	1	Ø	Minutes	X	X	X	X	Ø–9	
Ø	Ø	1	1	Tens of minutes	—	X	X	X	Ø–5	
Ø	1	Ø	Ø	Hours	X	X	X	X	Ø–9	
Ø	1	Ø	1	Tens of hours	*	*	X	X	Ø–1/Ø–2	D2 = 1 for PM; D2 = 0 for AM D3 = 1 for 24-hour format D3 = 0 for 12-hour format
Ø	1	1	Ø	Day of week	—	X	X	X	Ø–6	
Ø	1	1	1	Days	X	X	X	X	Ø–9	
1	Ø	Ø	Ø	Tens of days	—	*	X	X	Ø–3	D2 = 1 for 29 days in Feb D2 = 0 for 28 days in Feb
1	Ø	Ø	1	Months	X	X	X	X	Ø–9	
1	Ø	1	Ø	Tens of months	—	—	—	X	Ø–1	
1	Ø	1	1	Years	X	X	X	X	Ø–9	X Indicates data can be either 1 or Ø
1	1	Ø	Ø	Tens of years	X	X	X	X	Ø–9	

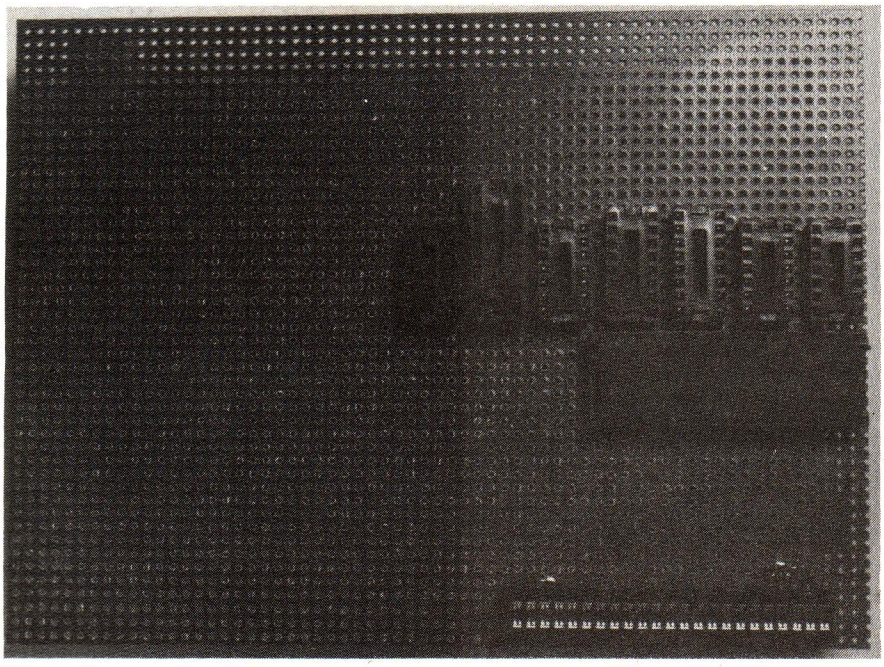


FIG. 4—THE INTERFACE BOARD. Wire wrapping is probably the easiest way to construct the interface. Be sure to leave space between the connector and the lowest IC socket so you can plug the board into the computer.

Even though the 74LS138 can decode and generate a device code pulse for that address when those three address lines are all logic 1's, the 74LS245 will not let the information pass. Keep that in mind when interfacing with the circuit.

Building the interface

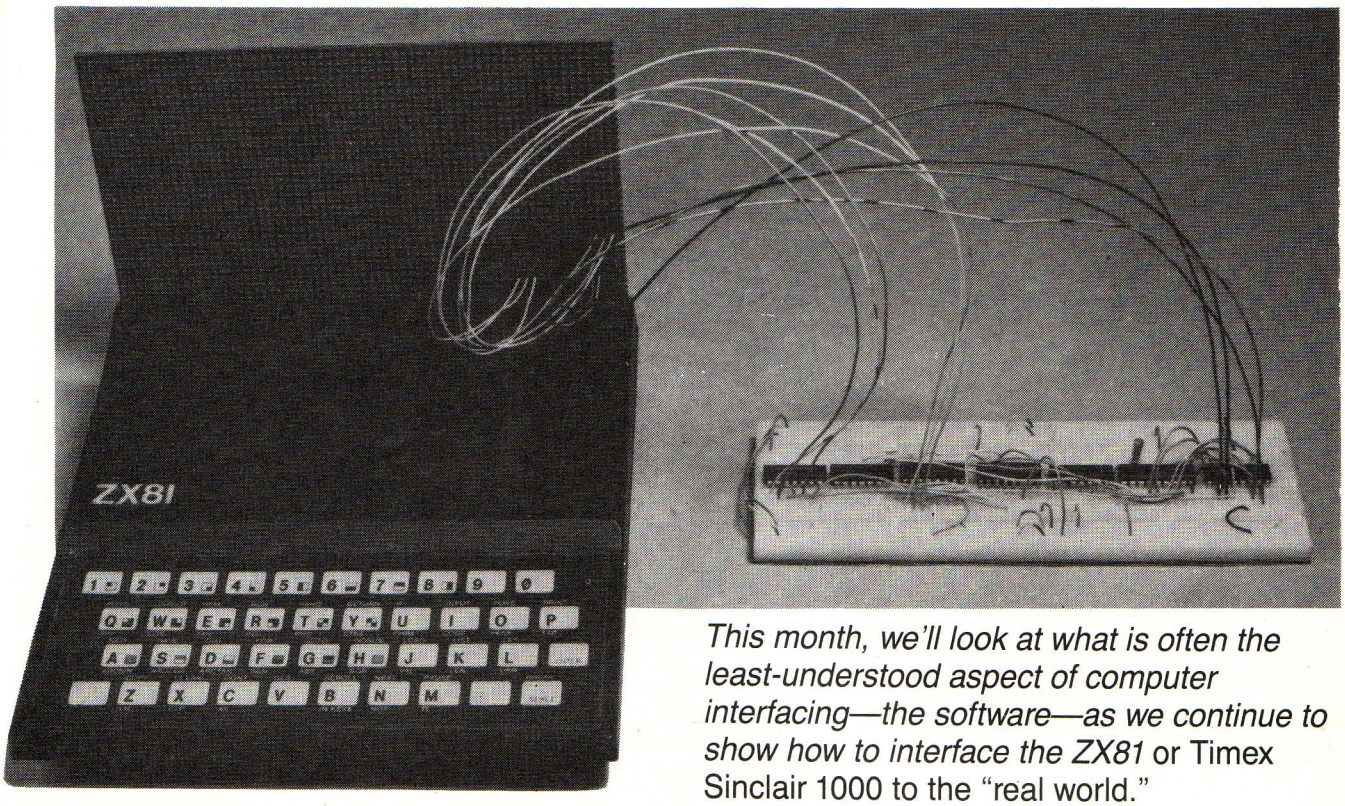
To wire anything to the card-edge port of the *ZX81*, you'll need a 44-pin card-edge connector. However, you cannot use a standard 44-pin connector. If you look closely at the card-edge fingers, you will notice that, in addition to the 44 fingers, there is a keyway. That keyway takes up exactly one pin location—it's used so that you won't plug anything in backward. It means that the connector you need is actually a 46-pin type. The problem is that standard connectors are available in 44- and 50-pin configurations but are not available with 46 pins.

You can do one of two things. You can buy a special connector that will fit the *ZX81*. (Such a connector is not too easy to find, but they are available from the "cottage industry" that has sprung up around the *ZX81*.) Your other choice is to buy a standard 50-pin connector and cut it to size. The 50-pin card-edge should have 0.100 by 0.200 inch spacings. It is available from many suppliers, in many different configurations. You'll probably want to buy the wire-wrap type because wire wrapping is easy to use and is easy to modify. But if you're not a wire-wrap fan, feel free to use some other construction method. Whatever method you use, you should make some note of the connector polarity. Either insert a "key" in the keyway or simply label the connector.

Once all the parts for the circuit have been acquired, the card-edge connector and wire-wrap sockets can be place on universal plugboard as shown in Fig. 4. Notice that there is approximately one inch between the top of the card-edge connector and the bottom of the lowest IC socket. If you intend to leave your *ZX81* in its case, be sure to do the same. Otherwise you won't be able to plug the expansion board into the computer. You will also notice that a a 40-pin IC socket has been placed on the plugboard, even though none of the IC's in the circuit requires it. That 40-pin socket has *not* been placed on the board to receive an integrated circuit; its purpose is for breadboarding. As new circuits are developed to interface to the *ZX81* they can be developed on a solderless breadboard and tested. (The photo on the title page of this article shows the clock/calendar circuit under construction before any IC sockets were placed on the plugboard.) When your circuits are working properly, they can be transferred to the plugboard in wire-wrap form. When you need a signal from the *ZX81* all you have to do is bring the signal to the back of the 40-pin IC socket by wire wrapping it from the card-edge connector. Then you can plug one end of a wire into the corresponding hole of the socket, and plug the other end of the wire into the solderless breadboard—very handy!

Another trick that you may find useful when wire wrapping is to use a small amount of quick-setting epoxy glue on the backs of the IC sockets before mounting them. That ensures a sturdy mount which makes wire wrapping easier. Labeling each IC and marking the number one pin of each IC makes the task of wire wrapping much easier; and in addition this aids in reducing wiring errors. Plastic slip-on pin identifiers are recommended.

Now that we have the basic hardware design completed, we can turn to what is often the most time-consuming and least understood aspect of any microprocessor-based project: the software—the program that makes your interface do what it has been designed to do. Next month, after we look at the software, we'll be able to put the interface to work as a temperature sensor, a security system, and more. **R-E**



This month, we'll look at what is often the least-understood aspect of computer interfacing—the software—as we continue to show how to interface the ZX81 or Timex Sinclair 1000 to the "real world."

Interfacing the ZX81

NEIL BUNGARD

Part 2 LAST MONTH, IN THE first part of this article, we took a look at hardware that allows us to connect external devices to the Sinclair *ZX81* (or Timex-Sinclair *1000*). We also looked at some basic principles of computer interfacing and described some of the *ZX81*'s interfacing idiosyncrasies.

We used the interface circuit to connect a clock/calendar IC (OKI's MSM5382) to the *ZX81*. But we couldn't do anything with the resulting circuit—we never discussed the software that is required to operate the it. This month, we'll start with a review of some general software principles and take a look at machine-language programming.

Machine language vs. BASIC

Why do we have to discuss machine-language programming? You might wonder why you can't use BASIC—a language that you're so familiar with and that is so easy to use—to control the interface. (If you're not familiar with BASIC and need more information, your *ZX81* user's manual has a good explanation of most instructions, and many examples of how to use them.) Actually, you *will* use BASIC to control the interface. But, because the Sinclair BASIC has no IN or OUT commands, you have to use machine language as well.

When you program in machine language, you are programming in the language that the Z80 microprocessor understands. BASIC simplifies programming by allowing many machine-language instructions to be represented by a single command. (When you program in BASIC, the *ZX81* must break each command into machine-language instructions before can execute it.) In addition, BASIC was written to be more understandable and easier to learn by the inexperienced programmer. Its word structure is similar to the English language, and the command-words more explicitly describe the operation that the command represents.

If you program in BASIC, you do not have to be familiar with the architecture of the microprocessor (CPU). However, when programming in machine-language the architecture of the CPU becomes increasingly more important: You must be more aware of how the microcomputer system is configured. That's because you become responsible for all of the "housekeeping" that BASIC takes care of automatically. If you don't completely understand that, don't worry—it will become clear as we study the machine-language mode of programming.

As mentioned earlier, the reason we must program in machine language is because no provisions were made in Sinclair's BASIC that allow us to output data to (or input data from) any external devices. There's another reason: Machine language executes many times faster than BASIC. That speed advantage, as you'll soon see, is a necessity in many interface applications.

Machine language and the *ZX81*

To begin programming in machine language, we must first consider the *ZX81* memory configuration. The *ZX81* system-control software, and a number of stacks and registers occupy the first 16,512 memory locations of the computer's memory space. (Actually, it only occupies the first 8K, but "repeats itself" in the next 8K.) That means that when you enter a program into the *ZX81* it begins storing your program at memory location 16513. For *ZX81* owners with 1K of memory, your user

memory extends to 17408, giving you only 895 memory locations into which you can store program instructions.

When you program in BASIC, the *ZX81* takes care of storing the instructions for you, and only bothers you with memory information (error-message 4) if you run out of space. But when programming in machine language, *you* are responsible for reserving memory locations for your program. Each instruction must be placed into a specific memory location within the reserved space.

Reserving space for machine code

One way to reserve space for the machine-language instructions or machine code is to set up a REM (*REM*ark) statement. The REM statement takes the following form:

1 REM 0123456789

Anything after the REM is not acted on by the computer—it is treated simply as a remark. However, the remark *does* occupy space in memory. For example, for the statement just listed, ten memory locations are occupied by the numbers from 0 to 9. Those ten locations are a good place to put the machine code. (Of course, the numbers 0 to 9 are not the machine code—they simply act as place holders until you enter the code.)

The reason we want to reserve the space in line number 1 is because we know the address where the first line of the program starts: The first character after the REM occupies location 16514. So the 0 in the REM statement is at location 16514, and the number 9 resides in memory location 16523. If your machine-code subroutine is longer than 10 bytes and you need more space, you can simply place more characters in the REM statement. If you want to know the ending address of your reserved space just add the number of characters after the REM statement to 16513. Once you have reserved sufficient space for the machine code you must then place the desired machine code into the reserved space.

We should point out that there are alternate ways to store machine code in the *ZX81*. One of the most convenient we've seen is to store it in RAM that occupies the addresses from 8K–16K. (That area is transparent to the *ZX81*'s operating system and is not affected by NEW or LOAD commands.) A circuit that allows you to do that was described in the July and August 1983 issues of **Radio-Electronics**.

Binary, decimal, and hexadecimal

If you're going to do any machine-language programming, you're going to have to get used to working in different number bases. The *ZX81* understands numbers only if you enter them in decimal (base 10) form. In other words, when you POKE anything into the *ZX81*'s memory, the address and data must be decimal numbers.

If you've never done any machine-language programming, you might think that using decimal numbers is convenient. It isn't. Usually, machine code is listed in hexadecimal (base 16) form. That's because it lets us represent one byte using just two symbols.

However, because everyone does not follow the same conventions, it is necessary to be able to convert from one number base to another. We will want to be able to convert to and from binary, decimal, and hexadecimal.

In this article, binary numbers will be listed with a capital "B" following the number while hexadecimal numbers will be followed by a capital "H." Decimal numbers will be written without any indication.

Before we can explain how to convert from one base to another, lets look at our decimal (base 10) system to see how it works. We all remember learning about the "ones' place," the "tens' place," the "hundreds' place," and so on. Each place is worth ten times the place to its right—as you move to the left, the value of each place increases by a factor of ten.

It's essentially the same when you work in other number bases—just the numbers change. For example, in the binary system, the value of each place increases by a factor of 2 as you move to the left. In the hexadecimal system, the value of each place increases by a factor of 16 as you move to the left. A look at Fig. 5 should clear up any questions you have—except, perhaps, one. Since the value of the "ones place" in hexadecimal doesn't change until after it reaches 15, you might wonder how you represent the numbers from 10–15 in hex while still using only one digit. It's rather simple: The numbers 10–15 are represented by the letters A–F. Here's a problem to see if you're following what we're talking about: What is D3H in decimal form? D3H = $(D \times 16) + (3 \times 1) = (13 \times 16) + (3) = 208 + 3 = 211$.

If you want to convert a hexadecimal formatted instruction to a decimal form, you have a few choices. First, you can follow Fig. 5 and the example above and multiply each number by its place value. But if you don't like multiplying things by 16, there's another way: Convert the hexadecimal form to binary and then convert the binary to decimal. It's very easy to convert hex to binary—it's a special case where you can simply take each number (or place) in hex and replace it with the corresponding number in binary. (See Table 4). For example, D=1101B and 3=0011B. Therefore, D3H = 11010011B.

TABLE 4

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

You can convert the binary number to decimal by multiplying each digit by its place value. For example, the decimal value for 11010011B is $(1 \times 1) + (1 \times 2) + (1 \times 16) + (1 \times 64) + (1 \times 128) = 211$.

Which of the two methods is easier? It all depends on your point of view. However, we're sure that you'll agree that the following is the easiest method of all: Use the table shown in Fig. 6. You can use it to convert from hex to decimal and from decimal to hex. Unfortunately, it works only for numbers between 0 and 255.

Whatever method you choose, we would advise you to become comfortable with base conversions—when you do a lot of machine-code programming, you'll do a lot of base conversions.

Entering machine code

We are now ready to begin storing machine-language instructions into the memory space you have previously reserved with the REM statement. How do you replace the contents of the REM statement with the machine code? You use BASIC! Specifically, the POKE instruction, which takes the form:

POKE *address, data*

where the *address* is the memory location

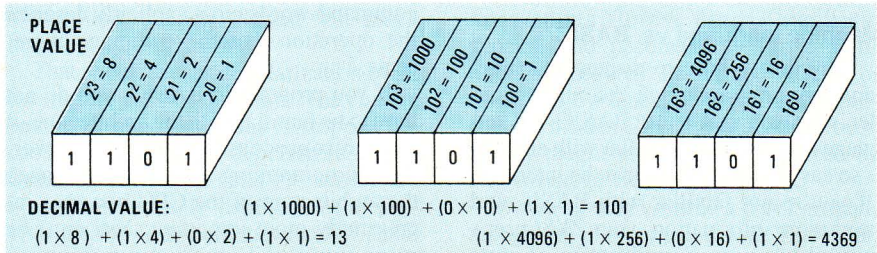


FIG. 5—CONVERTING FROM ONE BASE TO ANOTHER does not have to be difficult or confusing. Just remember that all number bases operate the same as the decimal system you're used to.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	1
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	2
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	3
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	4
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	5
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	6
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	7
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	8
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	9
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	A
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	B
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	C
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	D
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	E
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

FIG. 6—THE EASY WAY TO CONVERT FROM HEX TO DECIMAL. All decimal values from 0–255 can be represented by a two-digit hex number.

where the instruction is to be placed and the *data* is the machine-language instruction. In order for the *ZX81* to understand the POKE command, both the address and the instruction must (unfortunately) be in decimal form. If you want to place an instruction into a single memory location, a POKE instruction is probably the easiest way to do it. But in a situation where you want to store many instructions (the usual case), POKE-ing each value can be a tiresome (and error-prone) process.

One way to get around some of the tedium is to write a small *loader* program that sets the starting address of the machine code, POKEs the first instruction, and advances to the next memory storage location automatically. While we're at it, we'll include a couple of lines in the program to convert hex to decimal so that we can enter instructions in a hexadecimal format.

For the following programs, we'll assume that your *ZX81* has only 1K of memory, which is standard with the *ZX81*. (The Timex-Sinclair *1000* came equipped with 2K.) Unfortunately, all the programs we'll need to operate the clock/calendar interface cannot be stored in 1K of memory at the same time. Therefore, we'll have to write individual programs to perform specific tasks and then erase the programs when we have finished to make room for the next program needed. As you can see, cassette-tape storage is almost essential: Once the programs are written, they can be stored on tape and loaded into the *ZX81* via the tape player as they are needed. (Of course, if you have more memory than the standard 1K, you'll be able to write and save all of the programs in one shot. What? You can't plug in your RAM-pack when you have the interface board connected? We'll show you how to get around that problem next month.

Clear the *ZX81* with the NEW command and enter the program shown in Table 5. If you have a cassette recorder, SAVE the program.

The first line of the program reserves 17 memory addresses for storing a machine-language program. Line 10 sets the variable "X" to the value of the first address of the machine code storage area (16514) which is now occupied by the first number "1" in the REM statement. Line 40 converts your hexadecimal input to decimal and pokes it into memory.

The program takes your first input and POKEs it into location 16514. Your following inputs are POKEd into successive memory addresses until an ""S" is entered. When the machine-language program has been completely entered, input an S, and the program will end. Now you're ready to give the program a try. RUN it, and enter the machine code shown in Table 6.

Loading the registers

The left column is the machine code that is to be entered. The order of entering the code is: Ø6, 4Ø, ØE, 91, ØA, etc. (When prompted, enter one number—two digits—at a time, followed by hitting the ENTER key. Don't, of course, enter the commas.) The right-hand column contains the mnemonics that represents the Z80 machine-language instructions. Unfortunately, we cannot go into a detailed description of the mnemonics at this time. (A full explanation of the Z80 instruction set is contained in the *Z80 Software Manual* from Zilog Corporation, and in a number of Z80-programming books on the market.)

TABLE 5—MACHINE-CODE ENTRY

```
 1 REM 12345678901234567
10 LET X=16514
15 PRINT "INPUT DATA"
20 INPUT A$
30 IF A$="S" THEN STOP
40 POKE X,16*CODE A$+CODE
   A$(2)-476
50 LET X=X+1
60 GOTO 30
```

TABLE 6—REGISTER LOADING

```
Ø6 4Ø       LD B, 4Ø
ØE 91       LD C, 91
ØA          LD A, (BC)
D3 ØØ       OUT ØØ , A
ØE 92       LD C, 92
ØA          LD A, (BC)
D3 Ø8       OUT Ø8 , A
D3 Ø4       OUT Ø4 , A
C9          RET
```

After all of the machine code has been entered, enter an "S." Look at a LISTing of the BASIC program. In the REM statement, you'll notice that the first 15 numbers have been replaced with strange looking characters. Those characters represent the machine-language program that has just been entered. The reason that you do not see the machine code as it was entered is because the *ZX81* stores its "character set representation" of the hexadecimal numbers that were entered, and not the numbers themselves. A complete listing of the *ZX81* character set and the associated codes can be found in "Appendix A" of the *ZX81* User's Manual.

Now that you have the machine-language program in memory, you will no longer need the machine-code-entry program. Erase everything but the REM statement that contains the machine code (line 1) and enter the BASIC program in Table 7.

TABLE 7—REGISTER LOADING

```
10 FOR I=Ø TO 12
20 PRINT I
25 PRINT "INPUT VALUE FOR REGIS-
   TER";I
30 INPUT A
40 POKE 16529,I
50 POKE 16530,A
60 LET C=USR 16514
70 NEXT I
80 PRINT "END OF LOAD"
90 STOP
```

That BASIC program, along with the machine-language program previously entered, work together to load the MSM5832 clock/calendar IC with its initial settings. The BASIC program asks for an input of the value to be stored in each of the 12 registers of the MSM5832. (Table 8 defines the registers and their allowable data ranges.) The BASIC program then calls the machine-language routine (line 60) which does the actual loading of the values.

Let us follow the operation of these programs line-by-line as they load a register in the MSM5832: Line 10 sets the first register to be loaded to register 0 (seconds register). Line 20 will print the register number and line prompt you to enter the value you want stored there.

After you enter a value, the *ZX81* stores the register code in memory location 16529 (4091H) and stores the register val-

TABLE 8—MSM5832 REGISTERS

Register	Contents	DATA I/O D3	D2	D1	Ø	Allowable output range (decimal)
Ø	SECONDS	×	×	×	×	Ø–9
1	TENS OF SECONDS	—	×	×	×	Ø–5
2	MINUTES	×	×	×	×	Ø–9
3	TENS OF MINUTES	—	×	×	×	Ø–5
4	HOURS	×	×	×	×	Ø–9
5	TENS OF HOURS	*	**	×	×	Ø–1/Ø–2†
6	DAY OF WEEK	—	×	×	×	Ø–6
7	DAYS	×	×	×	×	Ø–9
8	TENS OF DAYS	—	††	×	×	Ø–3
9	MONTHS	×	×	×	×	Ø–9
10	TENS OF MONTHS	—	—	—	×	Ø–1
11	YEARS	×	×	×	×	Ø–9
12	TENS OF YEARS	×	×	×	×	Ø–9

Notes:
× Indicates that data can be either I or Ø
* D3 = 1 for 24-hour format. D3 = Ø for 12-hour format
** D2 = 1 for PM. D2 = Ø for AM
† Depends on D3
†† D2 = 1 for 29 days in Feb. D2 = Ø for 28 days in Feb.

ue in location 16530 (4092H). Line 60 calls the machine-language subroutine.

The first thing the subroutine does is take the register code out of memory location 16529 (4091H) and send it to the MSM5832 via OUT device-code pulse 00H. (Recall from Part 1 of this series that that device code selects the MSM5832 for inputting and stores the register code in the 74LS75.) The machine-language routine then retrieves the value to be placed into the selected register from memory location 16530 (4092H). The *ZX81* then sends the register value to the MSM5832 via the OUT device-code 08H. An OUT device-code 04H is then generated to deselect the MSM5832 so that it can resume normal timing operation. Program execution returns to the BASIC program and the remaining function registers are filled. When the last register is set, the *ZX81* prints "end of load" and the program execution stops.

Table 9 is an example of the values that must be loaded to initially set the MSM5832 to: Sunday, April 18, 1983, 1:25:00 PM. At completion of the above program, the MSM5832 clock/calendar IC will be loaded with initial values for time (*hours, minutes, seconds), date (year, month, day), and day of the week.

TABLE 9
INITIAL REGISTER VALUES FOR SUNDAY, AUGUST 19, 1984

Register code	Register value
Ø	Ø
1	Ø
2	5
3	2
4	1
5	4
6	6
7	9
8	1
9	8
10	Ø
11	4
12	8

Reading the register contents

Great—the MSM5832 knows the time and date, but we don't. A short program, however, will allow us to retrieve the time and date information from the MSM5832. To make room for the new retrieval program, clear the *ZX81* with the NEW command. That will erase both the machine-language and the BASIC programs used to load the MSM5839 with initial values. Since a new machine-language routine is to be used, the BASIC program which loads hexadecimal machine code must be re-entered. If you SAVEd the machine code entry program on cassette tape the first time you used it, it can be LOADed from tape. If not, you will have to type it again. (You can waste a lot of time that way, can't you?) After the hexadecimal machine-code entry program has been entered, you will need to reserve 39 memory locations for the machine-language program which retrieves the date and time values from the MSM5832. That is done via the REM statement:

1 REM (*40 characters*).

Type in the REM statement and load the machine-language program in Table 10, using the same method as you did earlier.

With the machine-language program in place, erase the entry program (lines 10 through 80) and enter the BASIC program in Table 11.

The BASIC program and machine-language routines work together to retrieve time and date information from the MSM5832. As the values of the 13 registers in the MSM5832 are retrieved, they are placed in memory locations 16540 through 16552. Table 6 lists the addresses and contents. (Those memory locations were reserved earlier by the REM statement with 40 characters.) Let's follow the program through one complete register retrieval.

TABLE 10—REGISTER RETRIEVAL

```
Ø6 4Ø       LD B,4Ø
ØE 99       LD C,99
ØA          LD A,(BC)
D3 ØØ       OUT ØØ,A
3A 9A 4Ø    LD A,(409A)
47          LD B,A
3A 9B 4Ø    LD A,(409B)
4F          LD C,A
DB ØØ       IN A, ØØ
E6 ØF       AND ØF
Ø2          LD (BC), A
D3 Ø4       OUT Ø4, A
C9          RET
```

TABLE 11—REGISTER RETRIEVAL

```
800 LET C=Ø
810 LET D=156
820 FOR I=Ø TO 12
830 POKE 16537,C
840 POKE 16538,64
850 POKE 16539,D
860 LET A=USR 16514
870 LET C=C+1
880 LET D=D+1
890 NEXT I
900 STOP
```

In lines 800 and 810, the initial register code (0 for seconds) and initial register content storage location are defined. In line 830, the initial register code is stored in location 16537 (409AH). In lines 840 and 850, the address of the initial register-content storage-location is placed into memory locations 16538 and 16539. Note that the address 16540 (409CH) has to be entered in two commands. Line 840 enters 64 (40H) and line 850 enters 156 (9CH). Line 860 calls the machine-language routine.

The first three instructions of the machine-language program place the contents of memory location 4099H (16537) into the Z80's accumulator. That value is the code that defines the MSM5832 register that will be accessed. (That location was previously loaded by line 830 before branching to the machine-language routine.) The OUT 00H,A instruction loads the 7475 (IC7 of the interface circuit) with the register code, and selects the MSM5832 for inputting. The next 4 instructions retrieve the address of where the MSM5832 register contents will be stored in the *ZX81*'s memory. (That was also previously defined in the BASIC program, lines 840 and 850, before branching to the machine-language program.)

The instruction IN A, 00H inputs the MSM5832 register contents. AND 0FH masks the four higher-order bits to logic zeros, and LD (BC), A stores the MSM5832 register contents in the *ZX81*'s memory.

The OUT 04H, A instruction deselects

continued on page 98

INTERFACING

continued from page 56

the MSM5832, which resumes normal timing operations. Program execution then returns to the BASIC language routine. Line 870 defines the next MSM5832 register which is to be input. Line 880 defines the next storage location for that register's contents, and if all 13 registers in the MSM5832 have not been accessed, the machine-language program is called again. If all 13 registers of the MSM5832 have been accessed, program execution stops.

After program execution stops, the contents of all 13 registers of the MSM5832 will reside in memory, in the addresses listed in Table 12. Once the date and time information reside in memory, it is a simple matter to retrieve it and display it using the PEEK instruction. For instance, if you wish to retrieve the day of the week, write this simple program, you can enter the direct command:

PRINT PEEK 16546.

The number retrieved represents a day of the week as defined in Table 13. (Of course you could write a simple program to give you back the actual day instead of the number.)

If you want the time to be retrieved and displayed try the program in Table 14. And if you want to retrieve and display the entire date, try the program in Table 15.

Of course, using separate programs is not the most elegant way to use the clock/calendar circuit. You could put all of the simple BASIC routines in a loop and have the time information updated periodically. To demonstrate updating, make the following changes:

TABLE 14—TIME RETRIEVAL

```
10 DIM A(6)
20 LET  X  16540
30 FOR J=1 TO 6
40 LET A(J)=PEEK X
50 LET X = X +1
60 NEXT J
70 LET B$="AM"
75 IF A(6)>2 THEN LET B$="PM"
80 IF A(6)>2 THEN LET
   A(6)=A(6)-4
85 CLS
90 PRINT A(6);A(5);":";A(4);
   A(3);":"; A(2);A(1);" "B$
100 STOP
```

TABLE 15—DATE RETRIEVAL

```
10 DIM A(6)
20 LET X =16547
30 FOR J=1 TO 6
40 LET A(J)=PEEK X
50 LET X =X+1
60 NEXT J
70 LET B=19
80 PRINT
   A(4);A(3);"-";A(2);A(1);"-";
   B;A(6);A(5)
90 STOP
```

- Delete line 90 in the time-display program (Table 14).

TABLE 12—MEMORY LOCATIONS OF MSM5832 REGISTER VALUES

Hexadecimal	Decimal	Register
409C	16540	0—SECONDS
409D	16541	1—TENS OF SECONDS
409E	16542	2—MINUTES
409F	16543	3—TENS OF MINUTES
40A0	16544	4—HOURS
40A1	16545	5—TENS OF HOURS
40A2	16546	6—DAY OF WEEK
40A3	16547	7—DAYS
40A4	16548	8—TENS OF DAYS
40A5	16549	9—MONTHS
40A6	16550	10—TENS OF MONTHS
40A7	16551	11—YEARS
40A8	16552	12—TENS OF YEARS

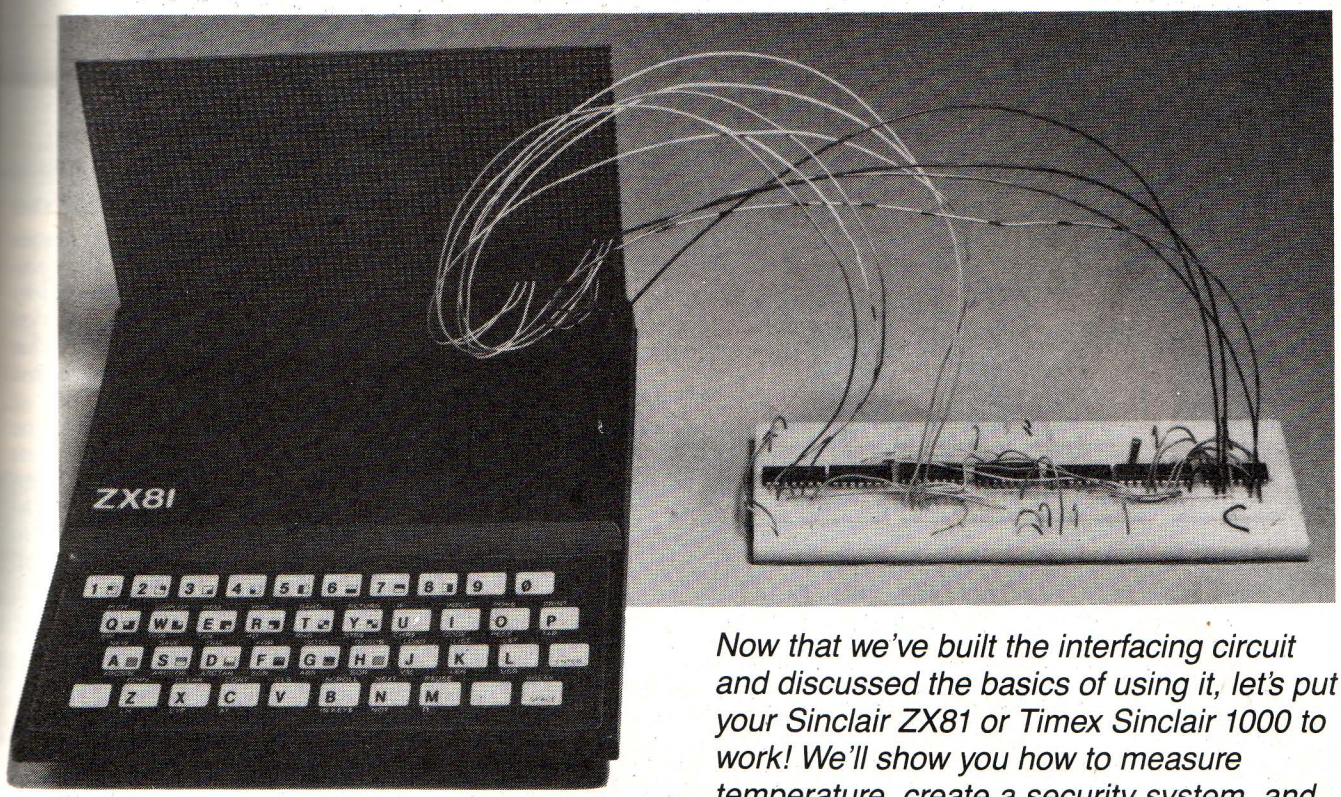
TABLE 13—DAY-OF-WEEK VALUES

0—Monday
1—Tuesday
2—Wednesday
3—Thursday
4—Friday
5—Saturday
6—Sunday

- Change line 900 in the retrieval routine (Table 8) to: 900 GOTO 20.

Those changes allow the time-display routine to print the time values (lines 10 through 80); go and get updated values from the MSM5832 (lines 800 through 900); and return to the time display routine. The program does that until a BREAK is entered.

We now have a good idea of how to control the clock/calendar IC. Controlling other external circuits and devices is not too much different. Next month, when we continue, we'll look at how to do that. When we're finished, you'll see how we can add an alarm function to the clock/calendar circuit. In addition to an alarm function you can use the MSM5832 in conjunction with the *ZX81* as a nighttime light controller and a date/time home security system. **R-E**



Now that we've built the interfacing circuit and discussed the basics of using it, let's put your Sinclair ZX81 or Timex Sinclair 1000 to work! We'll show you how to measure temperature, create a security system, and control high-power devices.

Interfacing the ZX81

NEIL BUNGARD

Part 3 THIS MONTH WE'LL use the principles established in the first two parts of this series to build some fun and useful microcomputer projects. They include a home-security system, a temperature-sensing circuit, and methods of controlling high-current devices with the microcomputer.

The number of things you can do with your microcomputer and interface are endless. In other words, the methods and principles that we'll use may be applied to any number of projects that you might dream up yourself. And you can combine all of the circuits we present to do numerous things. So don't be afraid to take what we start with and improve and expand on it. Your imagination is the only limit on what can be done.

TV interference

Before we get started building the add-on circuits, we should address a problem that is created by those circuits—TV-picture deterioration. As more integrated circuits are added to the Sinclair interface, you will notice interference on your TV picture. (Many *ZX81* users have trouble with screen interference even *without* hooking anything up to the computer!) You might want to try some of the following quick-fix tips to help clear up your picture.

One trick that sometimes works is to roll up the connecting cable that goes between the *ZX81* and the antenna/*ZX81* switching box. Another trick that helps a little is to move the Sinclair away from the TV set—elevating the TV set seems to have the most favorable results. Perhaps the easiest way to improve your picture is to ground the antenna switching box to the UHF-antenna input on the back of your TV set. That can be done by wrapping a copper wire around the switching box and attaching its end to the UHF antenna input. If you do those three simple things, you should be able to obtain a picture that's almost as clear as the picture without the interface circuit attached. Although shielding the interface and associated circuits in a metal case is a bit more work than the previous fixes, it will give better results.

A home-security system

The first interface add-on we'll look at is a home-security system. We'll present only a minimum-security system. But you can add as much sophistication as you want. The limiting factor is the amount of program memory available on your microcomputer.

Figure 7 shows the security system's schematic. To guard eight doors and/or windows, all you need are eight magnets, 8 reed switches, and one eight-bit three-state latch (IC11, a 74LS373). You could, of course, use other alarm switches (such as pressure mats, or door-mounted plunger switches). But we'll discuss reed switches because they're easy to mount.

You should mount the normally open reed switch so that the magnet holds the switch closed when the window or door is closed. Then, when the window or door is closed, the line associated with that window or door will be at a logic Ø. If the window or door is opened, the associated reed switch will also open, and IC11 will see a logic 1 (because of the pull-up resistor). To check the status of the eight windows or doors, all you have to do is have the *ZX81* periodically generate an "IN Ø5" device-code pulse. That inputs the window/door status information into the accumulator of the Z80 where the status of each bit can be checked. The flow

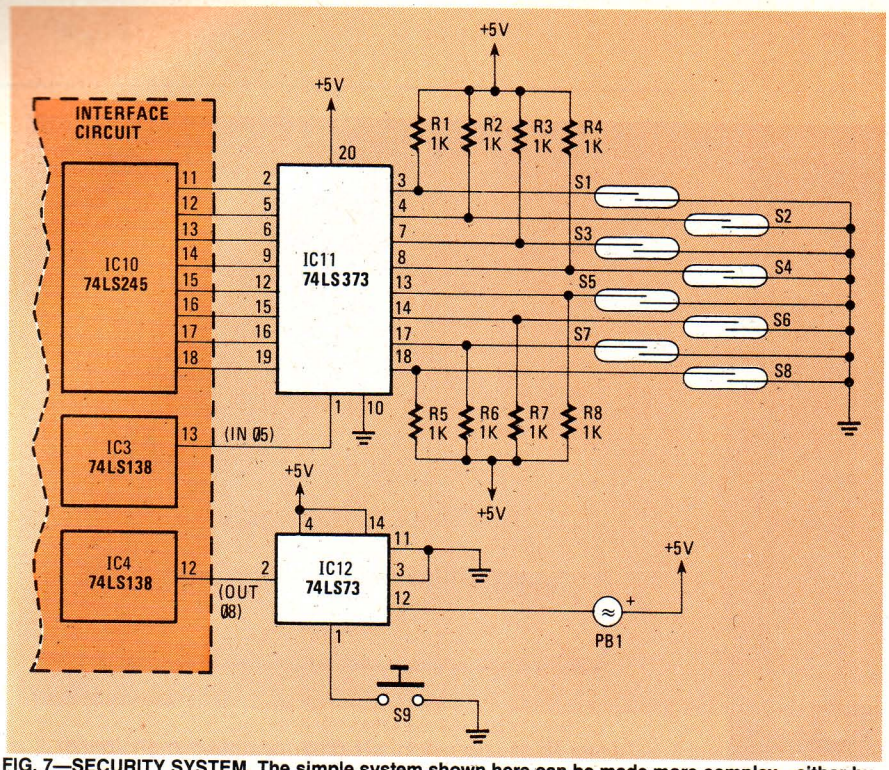


FIG. 7—SECURITY SYSTEM. The simple system shown here can be made more complex—either by adding more switches (or sensors) or by controlling some device other than a buzzer when a break-in is sensed.

chart in Fig. 8 describes the software required to operate the security system.

You can, of course, make the program a little more extensive. For example, you can check each bit of the status information to determine *which* window or door was opened. Or, if you want to note the time of the security breach (or to keep track of when your teenage daughter comes home) you can use the clock/calendar circuit developed over the last two months in conjunction with the security system.

But because we have so much to cover, we'll keep things simple. The machine-code and BASIC programs in Tables 16 and 17 will accomplish the task expressed by the flow chart in Fig. 8. To load the above programs, first type "10 REM 123456789012." Next use the machine-language entry program presented in Table 5 (Part 2) to load the machine-language subroutines into the REM statement. Once the machine language is loaded, erase the entry program and enter the remainder of the BASIC routine.

Now let's look at how the programs control the security-system circuit. Line 20 calls the machine-language routine located at memory address 16514. The first two instructions there define where the security information is going to be stored in memory once it is retrieved from the 74LS373. The command "IN A,Ø5" fetches the security information from the 74LS373. (Remember: You must use odd input-device codes or bits D6 and D7 will be masked out when you input. That's one

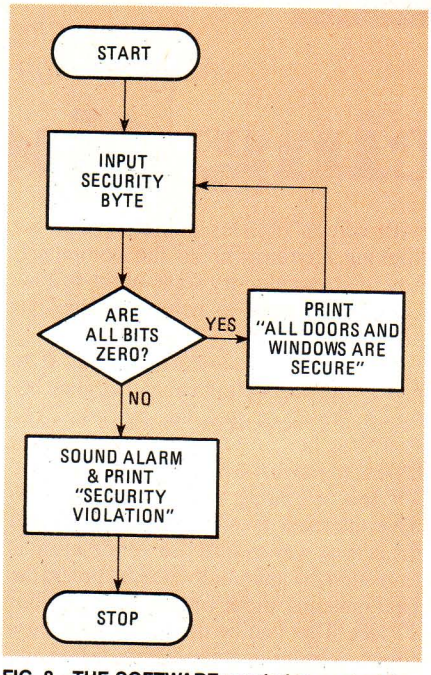


FIG. 8—THE SOFTWARE needed to control the basic security system can be as simple as this flowchart shows.

TABLE 16—SECURITY MONITORING

```
Ø6 4Ø      LD B,4Ø
ØE 8D      LD C,8D
DB Ø5      IN A,Ø5
Ø2         LD (BC),A
C9         RET
D3 Ø8      OUT Ø8,A
C9         RET
```

TABLE 17—SECURITY MONITORING

```
1Ø REM 123456789012
2Ø LET A=USR 16514
3Ø LET B=PEEK 16525
4Ø IF B=Ø THEN GOTO 8Ø
5Ø LET A=USR 16522
6Ø PRINT "SECURITY VIOLATION"
7Ø STOP
8Ø PRINT "ALL DOORS AND
   WINDOWS SECURE"
9Ø GOTO 2Ø
```

of those strange Sinclair idiosyncrasies that you have to get used to.) The command "LD (BC),A" places the security information into the memory location pointed to by the BC register pair. That storage location was defined in the first two machine-language instructions as 408DH (16525). Execution returns to line 30 of the BASIC program, which sets variable "B" equal to the security information just placed in memory by the machine-language routine. If "B" is equal to zero in line 40 (no security violation), program execution branches to line 80 where an "all secure" message is printed, and the entire process is repeated.

If, in line 40, "B" does not equal zero, then a security violation has occurred. In that case, program execution continues to line 50, where another short machine-code routine is called. That routine does one thing: It generates an "OUT Ø8" device-code pulse. As you can see in Fig. 7, that device-code pulse is fed to pin 2 of a 74LS73 flip-flop (IC12). When the 74LS73 sees a negative-going pulse on pin 2, it sets its output (pin 12) to a logic Ø. That causes the piezoelectric buzzer, PB1, to turn on. After the alarm circuit is turned on, program execution returns to line 60 in the BASIC program, where "security violation" is written to the TV screen. In line 70, program execution stops. When pin 1 of the 74LS73 (IC12) is grounded by pressing S9, the output of the 74LS73 is reset to a logic 1 and the alarm is turned off.

If you want to monitor more doors and windows (or pressure mats, etc.) you can add additional three-state devices to the circuit and read the states by using a different input device-code pulse.

Temperature sensing

The next circuit we'll discuss will allow you to monitor temperature. It has a measurement range from 0 to 100 degrees

PARTS LIST—SECURITY SYSTEM

Resistors ¼-watt, 5% unless otherwise noted
R1–R8—1000 ohms
Semiconductors
IC11—74LS373 octal D-type latch
IC12—74LS73 dual J-K flip-flop
Other components
S1–S8—Reed switches. See text
PB1—Piezoelectric buzzer

Celsius with an accuracy of a couple of degrees Celsius. You can see that it's not a precision thermometer, but you can use the circuit to measure ambient temperature or the temperature of a developer bath in a darkroom. Based on the temperature, you can have the computer turn heaters or fans on or off (using control circuits that we'll get to shortly).

Referring to Fig. 9, the thermometer circuit consists of National Semiconductor's LM335 temperature sensor and the ADC0804 single-channel, eight-bit analog-to-digital (A/D) converter.

The ADC0804 connects directly to the interface circuit developed in Part 1, and only two signals are necessary to control the circuit's operation. The ADC0804 control lines are connected to the two 74LS138's (IC's 3 and 4) of the Sinclair interface circuit. An "OUT ØC" device code pulse starts an analog-to-digital conversion of the LM335's output voltage. At the end of the conversion (about 100 microseconds), a digital value representing the sensor's temperature is input into the Z80's accumulator with an "IN Ø9" device-code pulse.

The temperature-sensing circuit can be adjusted via resistors R11 and R12 so that the temperature will be read directly in degrees Celsius. Potentiometers R11 and R12 should be ten-turn (at least) potentiometers so that a fine adjustment of the zero setpoint and the span can be accomplished.

The LM335 should be soldered to a pair of small-gauge, twisted wires and a small amount of silicone rubber should be placed on the leads to insulate them from liquids in which the sensor might be submerged.

Once you have the sensor prepared, you can calibrate the circuit. You'll need a glass of ice water, a glass of boiling water, and a Celsius thermometer that you know to be acc rate. You will also need some software.

Enter, for line 10, a 12-character REMark statement to reserve room for two machine-language subroutines. Use the machine-language entry program presented in Part 2, Table 5 to load the machine-language routines in Table 18. Then load the BASIC program in Table 19.

Line 20 of that program calls the first of the two machine-language subroutines—the instruction "OUT ØC,A," which generates an output device-code pulse that starts an analog-to-digital conversion by the ADC0804. Once the device-code pulse has been sent, program execution returns to line 30 of the BASIC program, which calls the second machine-language subroutine (at memory location 16517).

The first two instructions, "LD B,40" and "LD C,8D," set the memory location where the temperature information is to be stored. That location is 408DH or 16525. The third instruction, "IN A,09," reads the temperature information from the ADC0804 and "LD (BC),A" stores the temperature information in memory. Program execution returns in line 40 of the BASIC routine, which sets the variable "B" equal to the temperature information just placed in memory by the machine-language subroutine. Line 50 clears the screen and line 60 prints the new temperature. After a pause, the entire sequence is repeated.

To calibrate the sensing circuit, run the above program and place the LM335 into the glass of ice water along with the known-accurate thermometer. Allow the sensor and thermometer to sit for a few minutes and then adjust resistor R1 until the thermometer reading and the number printed on the screen are the same. That adjusts the zero setting. (If the temperature of the ice water is not zero, or very close to zero, suspect that something is wrong.)

Now place the LM335 and the mercury thermometer into the glass of boiling water. After you allow them to set for a few minutes, adjust R2 until the reading on the screen matches that of the mercury thermometer. That sets the span of the

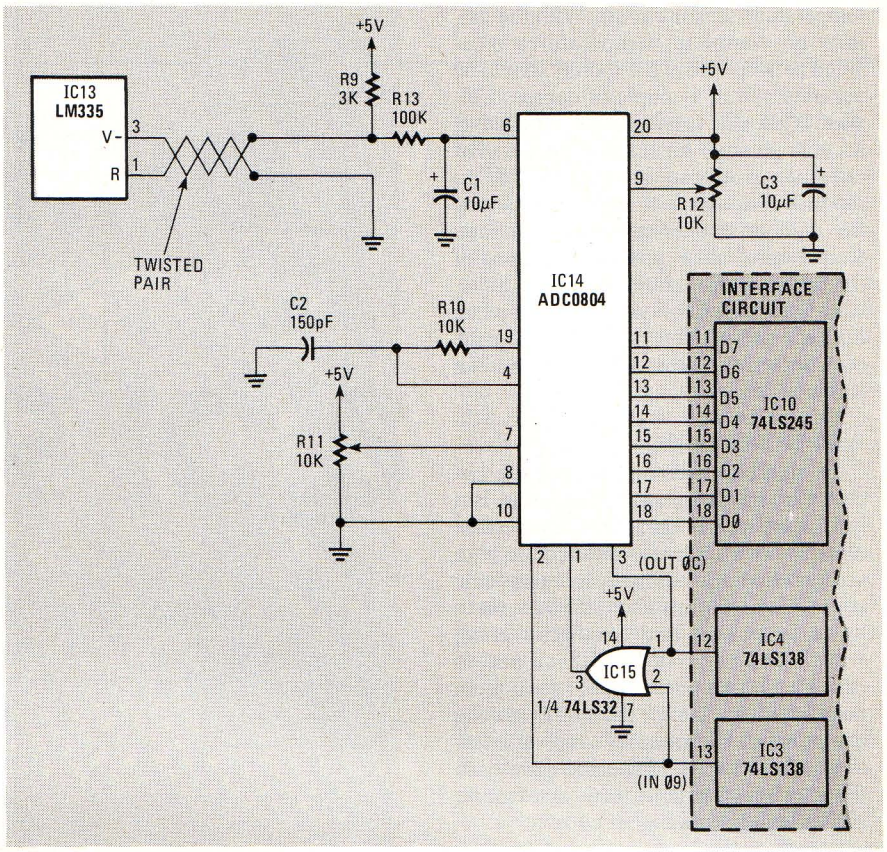


FIG. 9—TEMPERATURE SENSING CIRCUIT. Note that R11 and R12 should be multiturn-type potentiometers—it makes calibration easier.

TABLE 18—TEMPERATURE MEASURING

D3 ØC	OUT ØC,A
C9	RET
Ø6 4Ø	LD B,4Ø
ØE 8D	LD C,8D
DB Ø9	IN A,Ø9
Ø2	LD (BC),A
C9	RET

TABLE 19—TEMPERATURE MEASURING

```
2Ø LET A=USR 16514
3Ø LET A=USR 16517
4Ø LET B=PEEK 16525
5Ø CLS
6Ø PRINT "THE TEMPERATURE IS
   ";B;"CENTIGRADE"
7Ø PAUSE 1ØØ
8Ø GOTO 2Ø
```

PARTS LIST—TEMPERATURE SENSOR

Resistors ¼ watt, 5% unless otherwise noted
R9—3000 ohms
R10—10,000 ohms
R11, R12—10,000 ohms, multiturn potentiometer
R13—100,000 ohms
Capacitors
C1, C3—10-μF, 10 volts, electrolytic
C2—150 pF, ceramic disc
Semiconductors
IC13—LM335 temperature sensor
IC14—ADC0804 analog-to-digital converter
IC15—74LS32 quad OR gate

temperature-sensor circuit. With the calibration complete, temperatures measured between the two setpoints should be accurate within a couple of degrees Celsius. While that is not great accuracy, it is accurate enough for many purposes. For example, you could use the *ZX81* to monitor two different temperatures, say the outdoor temperature and your attic temperature. Then, you could use one of the power-control circuits (that we'll discuss next) to turn on your attic fan when the attic temperature is higher than the outside temperature. (But only in the summer, of course.)

A power controller

An interface that can control logic circuits certainly has many applications. But we're sure you'll agree that an interface that can control devices that require high voltage or current has many more possible applications. For instance, if you want to use a large siren and floodlights along with the security system, or if you want to activate a heater or fan, you'll need more power than the *ZX81* or the interface alone can provide. However, let's look at a few devices that can be connected directly to the interface to control loads that require higher voltage and higher current.

The first device we will consider for power handling is the mechanical relay. The relay is a simple device to use and can be used for either AC or DC loads. Figure 10-a shows a typical relay/microcomputer hookup. If a logic 1 is written to the latch (a 74LS373) through DØ of the microcomputers data bus, the relay will be de-energized. If a logic Ø is written to the latch, the relay will be energized and the load will be switched on. The disadvantage of this configuration is that mechanical relays that can be controlled with logic-level signals typically can only handle loads up to 0.5 amp at 120 volts.

Another kind of relay which has gained popularity in the past five years is the solid-state relay (SSR). Like its mechanical counterpart, the SSR can be activated with a logic-level signal, but it is generally used to switch AC loads. The new-generation solid-state relay can switch both AC and DC loads so that it is becoming a direct replacement for the mechanical relay. The real advantage of the SSR is that it can handle a great deal of power (for example 35 amps at 110 volts), and has no moving parts. That means that your *ZX81* can indirectly control a 3850-watt device—a formidable sized heater, for example. The interface circuit for the solid-state relay is the same as that for the mechanical relay. The solid-state relay's particular disadvantage is that it is relatively expensive (about $18.00 for a 120 volt AC, 20-amp model).

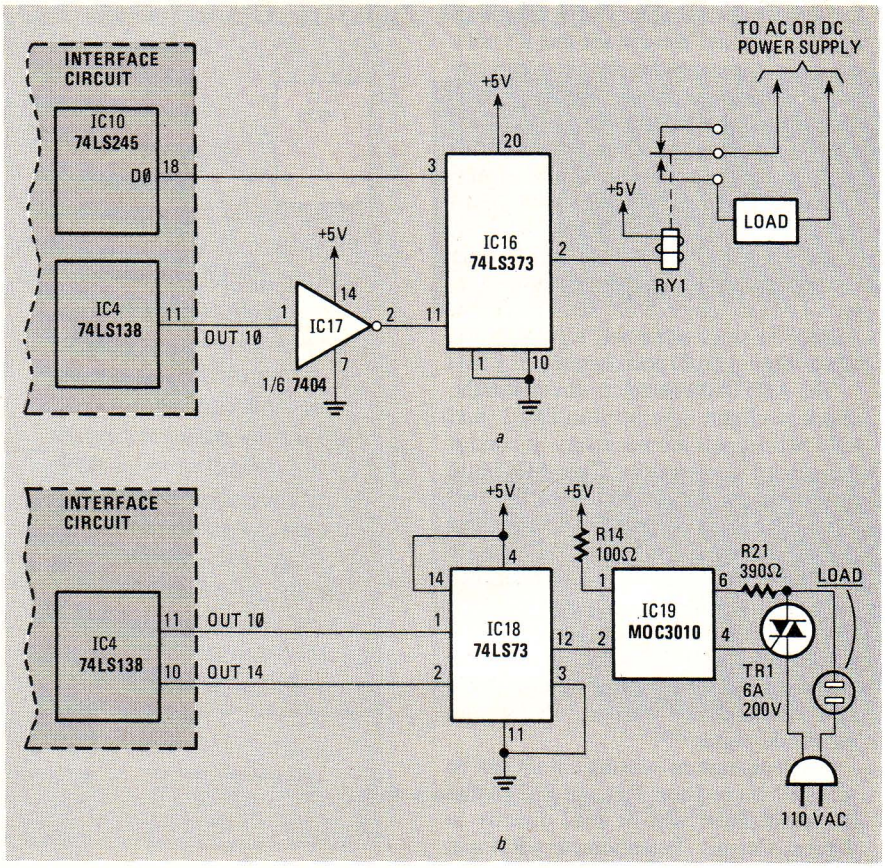


FIG. 10—YOU CAN CONTROL high-voltage circuits by either adding a solid-state or mechanical relay (as in *a*) or by using a triac driver and triac (as in *b*).

The software listed in Table 20 is to control the relay circuits. When the two routines are stored in a REM statement, they can be called with a simple "RAND USR *xxx*" statement, where *xxx* is the starting address of the routine.

The third power-handling device we'll look at is a solid-state device for controlling AC loads. The device is called a triac; the interface circuit is shown in Fig. 10-b. Connected to the gate of the triac is an optically-coupled triac driver (which allows a logic signal to control the triac). Since the driver is optically coupled to the triac, the power circuit is electrically isolated from the computer.

To control the circuit, instead of writing to a latch (as was done in the relay circuit), a set/reset flip-flop (a 74LS73) is used to generate the bistable logic-levels that control the power circuit. When an "OUT 10" pulse is generated, it sets the flip-flop's output to a logic 1, turning the triac off. When an "OUT 14" pulse is generated, it resets the flip-flop's output to a logic Ø and the triac is turned on. The triac circuit is best suited for AC loads up to 300 watts, and will not work on DC loads at all.

TABLE 20—RELAY CONTROL

3E	ØØ	LDA, ØØ	Relay
D3	1Ø	OUT 1Ø, A	on
C9		RET	
3E	Ø1	LD A, Ø1	Relay
D3	1Ø	OUT 1Ø, A	off
C9		RET	

The software to control the triac circuit is a little different from the relay-circuit software. The data bus is not needed for control, instead, two output device-codes are used. One device code turns the load circuit on and the other device code turns the load circuit off. The following machine language routine is all that is required to control the triac circuit:

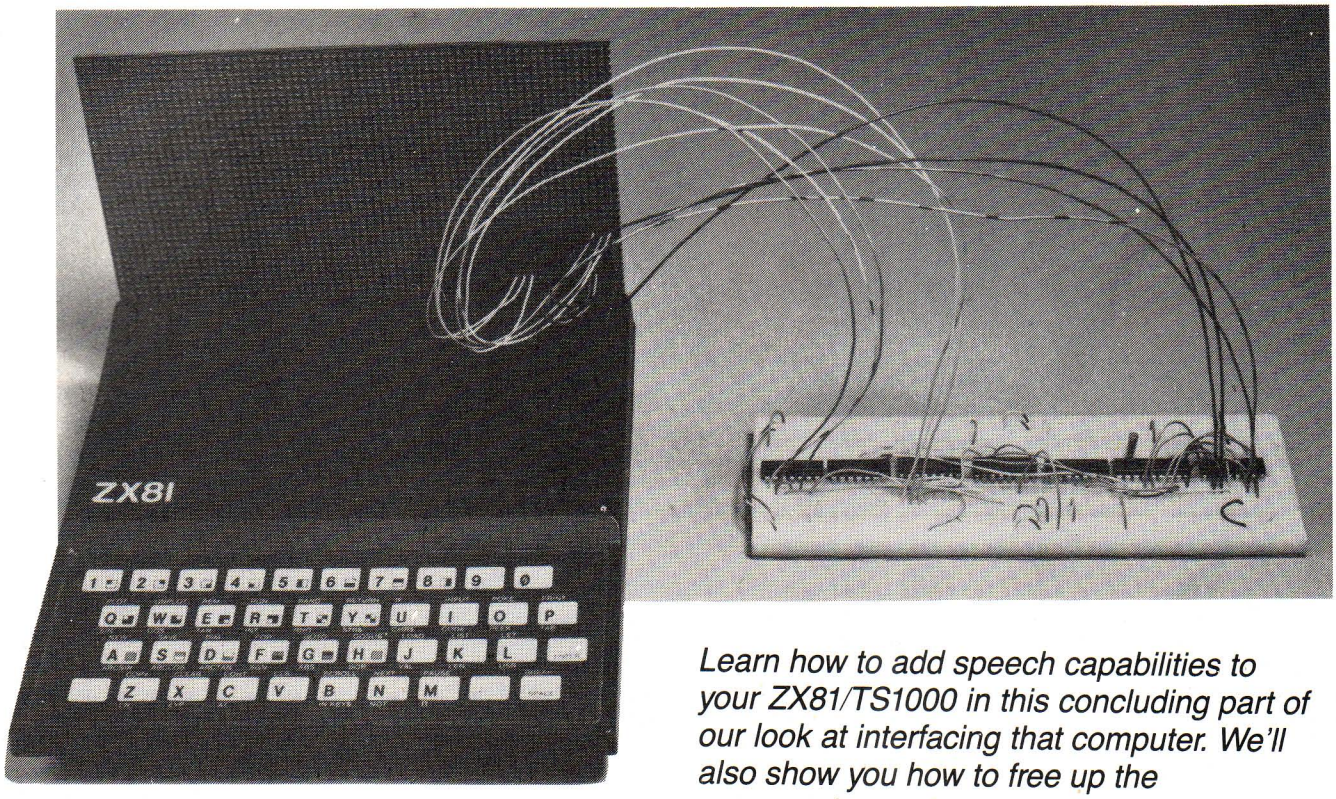
To turn the triac on: OUT 1Ø,A
To turn the triac off: OUT 14,A

Each of the three power-handling circuits has its own specific advantages, and depending upon your application you can choose the device that best suits your needs.

We've saved the best add-on—a speech synthesizer—for last. Unfortunately, that means that you'll have to wait until next month for its description. **R-E**

PARTS LIST—POWER CONTROLLER

Resistors ¼-watt, 5%
R14—100 ohms
R21—390 ohms
Semiconductors
IC16—74LS373 octal D-type latch
IC18—74LS73 D-type flip-flop
IC17—74LS04 hex inverter
IC19—MOC3010 optically coupled triac driver
TR1—triac, 6 amps, 200 volts
Other components
RY1—SPDT relay, 5-volt, 72 mA coil



Learn how to add speech capabilities to your ZX81/TS1000 in this concluding part of our look at interfacing that computer. We'll also show you how to free up the computer's port for other uses.

Interfacing the ZX81

NEIL BUNGARD

Part 4 THIS MONTH, WE'LL finish up our look at interfacing the *ZX81/TS1000* by showing you how you can add speech capabilities to that machine.

Adding speech capabilities

A *Digitalker* speech synthesizer is the last of the external circuits that we'll be adding to our interface. The *Digitalker*, which is manufactured by National Semiconductor, consists of a speech processor IC (SPC) and ROM that contains the synthesizer's vocabulary. (For more information on the *Digitalker*, see the July 1982 issue of **Radio-Electronics**.)

We'll be using a set of three IC's (the SPC and two 8 × 8K ROM IC's) called the DT1050. It sells for about $35 and is available from a number of suppliers, including Jameco Electronics.

The complete synthesizer is not difficult to build: All that's required is to add simple filter and amplifier circuits to the DT1050 IC set.

As we mentioned before, the synthesizer's vocabulary (which consists of 137 separate "words," 2 tones, and 5 different silence durations) is contained in two ROM IC's (MM52164SSR1 and MM52164SSR2). The vocabulary is listed in Table 21. (That is the most popular vocabulary set, although other ROM's with different, and larger, vocabularies are available.)

The synthesizer circuit, as shown in Fig. 11, consists of a digital and an analog section. The digital section consists of the speech-processor IC22, two ROM's, IC23 and IC24, and two 7400-series IC's for control-signal decoding.

The analog section of the circuit consists of IC25 and IC26. IC25 is an LM346 op-amp that is configured as a lowpass filter with a rolloff frequency of about 200 Hz. The filter is required to take out the high-frequency noise generated by the *Digitalker*'s speech-synthesis technique. IC26 is an LM386 audio-amplifier IC that is used to drive a small 8-ohm speaker.

To use the *Digitalker* with the *ZX81*, 8 data-bus and 3 control-line connections are required. The data bus carries information to the SPC (telling it which word to speak). The control signals are used to time the information transfer and start the speech sequence. Figure 12 shows the control-signal timing necessary to write information to the SPC.

That control-signal timing is accomplished with 3 output device-code pulses (18, 14, and 1C) and a IC20. When pin 2 of the IC20 sees a low-going pulse accomplished with an "OUT 14" device-code pulse, its output (pin 12) goes to a logic Ø. (That's the negative-going edge of the pulse generated on the CS input of the SPC in the waveform shown in Fig. 12.) Next an "OUT 18" device-code pulse is generated to the $\overline{WR}$ input of the SPC. That pulse initiates the data transfer into the SPC, and starts the speech sequence. The final device-code pulse that is generated is "OUT 1C", which sets the output of the 74LS73 (pin 12) back to a logic 1 and deselects the SPC. (That's the rising edge of the pulse in the top waveform in Fig. 12.)

As long as the *Digitalker* is speaking, its INTR output (pin 6) is at logic Ø. At the end of a speech sequence however, the INTR line goes to a logic 1.

The status of INTR is checked by using an "IN ØC" device-code pulse which is

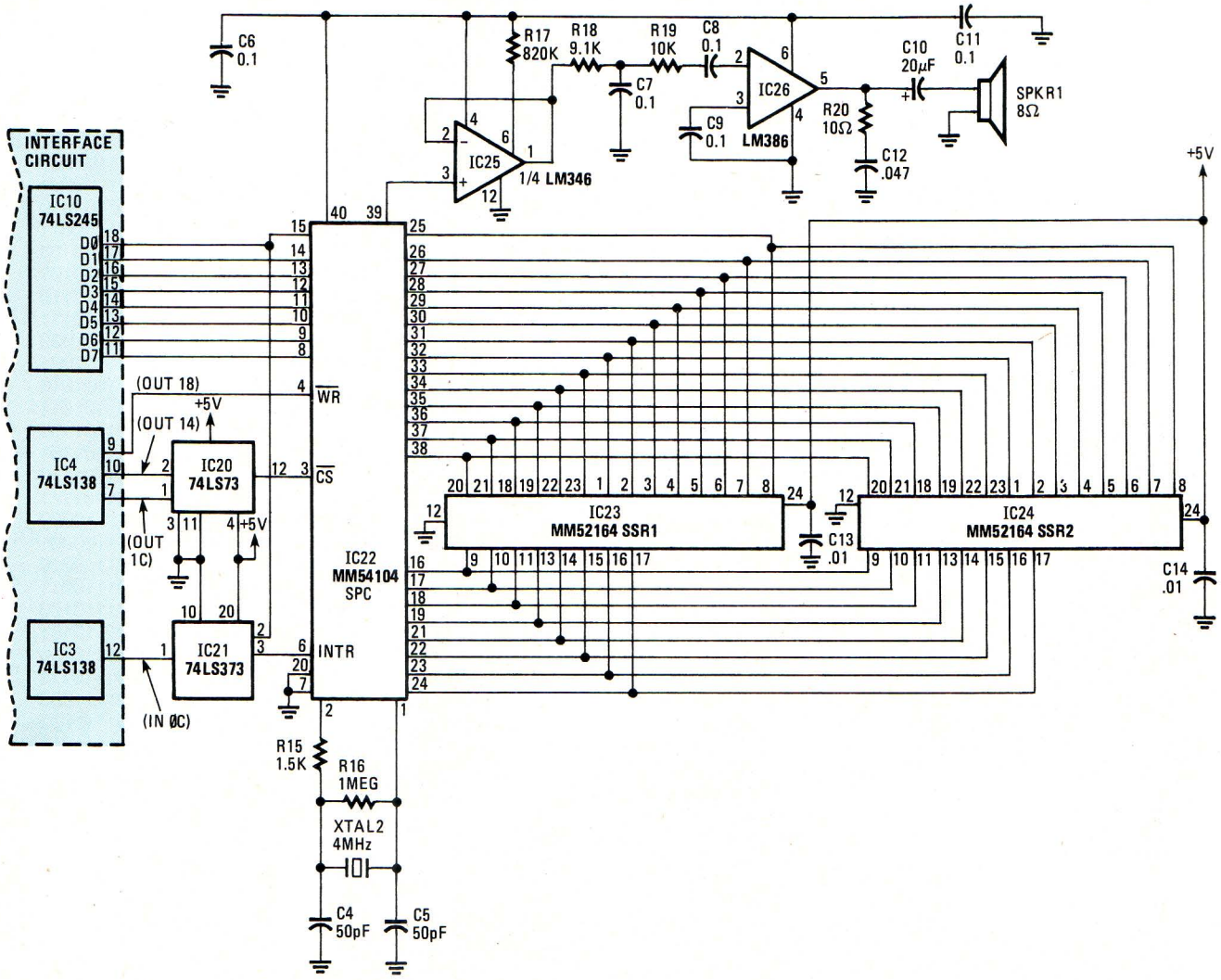


FIG. 11—THE SPEECH SYNTHESIZER circuit, thanks to large-scale integration, is not as complex as you might think. For best voice quality, don't use a very small speaker for SPKR1.

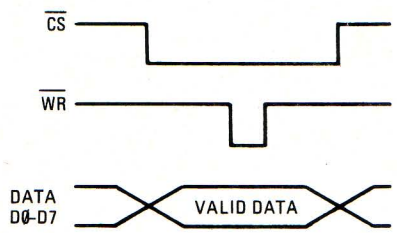


FIG. 12—THREE Z80 CONTROL SIGNALS are needed to generate the timing shown here.

connected to the ENABLE pin of IC21, a 74LS373 three-state latch. When IN ØC is generated, the status of the INTR output of the SPC is sent through IC10 to the Z80 accumulator on data-bus bit DØ. If DØ is a logic Ø, the speech sequence is still in progress. If DØ is a logic 1, then the speech sequence has ended, and the next word can be sent to the SPC.

We will demonstrate the capabilities of the *Digitalker* by using a machine-language routine and two BASIC programs. Remember that the programs we'll use are only a starting point. You are encouraged to modify the programs. For example, you might want to use the *Digitalker* with any of the other interface projects presented in this series.

To begin, first reserve the required space for the machine-language program which operates the *Digitalker*'s circuitry. A 20-character REM statement accomplishes that task. Then load the machine-language subroutine listed in Table 22. When the machine code has been entered, erase the loader program (if you used one) and load the BASIC program in Table 23.

Let's follow the two programs to see what they accomplish. Beginning in line 20 of the BASIC program, the variable "A" (which represents the numerical value of the word that the *Digitalker* will be asked to speak) is set to zero. (A complete listing of the *Digitalker*'s 144-word vocabularly and each word's associated numerical value is shown in Table 21.) Line 30 places the value of "A" into a memory location where it will later be retrieved by the machine-language subroutine that is called by line 40.

The first three instructions of the machine-language program repeatedly check the INTR output of the SPC to determine if a speech sequence has ended. As we mentioned previously, that's done by generating an input device-code pulse (IN ØC), setting all bits of the accumulator to a logic Ø except bit DØ (AND Ø1), and asking if DØ is a logic 1 or a logic Ø (JZ 82 40). If DØ is a logic Ø, program execution returns to the beginning of the machine-language routine. If DØ is a logic 1, which means the speech sequence has ended, the numerical value of the next word to be spoken is retrieved from memory location 4095H (16533), where it was previously stored by the BASIC routine. That's accomplished with the instructions: LD B,40, LD C,95, and LD A,(BC). Once the numerical value of the next word to be spoken is in the accumulator of the Z80, it is transferred to the SPC by a series of output instructions (OUT 14, OUT 18, and OUT 1C).

With the data transferred to the SPC, the speech sequence starts automatically. Program execution returns to the BASIC

TABLE 21—DIGITALKER VOCABULARY

Word	Decimal value	8-bit binary address S8 S1	Word	Decimal value	8-bit binary address S8 S1	Word	Decimal value	8-bit binary address S8 S1
THIS IS DIGITALKER	000	00000000	Q	048	00110000	IS	096	01100000
ONE	001	00000001	R	049	00110001	IT	097	01100001
TWO	002	00000010	S	050	00110010	KILO	098	01100010
THREE	003	00000011	T	051	00110011	LEFT	099	01100011
FOUR	004	00000100	U	052	00110100	LESS	100	01100100
FIVE	005	00000101	V	053	00110101	LESSER	101	01100101
SIX	006	00000110	W	054	00110110	LIMIT	102	01100110
SEVEN	007	00000111	X	055	00110111	LOW	103	01100111
EIGHT	008	00001000	Y	056	00111000	LOWER	104	01101000
NINE	009	00001001	Z	057	00111001	MARK	105	01101001
TEN	010	00001010	AGAIN	058	00111010	METER	106	01101010
ELEVEN	011	00001011	AMPERE	059	00111011	MILE	107	01101011
TWELVE	012	00001100	AND	060	00111100	MILLI	108	01101100
THIRTEEN	013	00001101	AT	061	00111101	MINUS	109	01101101
FOURTEEN	014	00001110	CANCEL	062	00111110	MINUTE	110	01101110
FIFTEEN	015	00001111	CASE	063	00111111	NEAR	111	01101111
SIXTEEN	016	00010000	CENT	064	01000000	NUMBER	112	01110000
SEVENTEEN	017	00010001	400HERTZ TONE	065	01000001	OF	113	01110001
EIGHTEEN	018	00010010	80HERTZ TONE	066	01000010	OFF	114	01110010
NINETEEN	019	00010011	20MS SILENCE	067	01000011	ON	115	01110011
TWENTY	020	00010100	40MS SILENCE	068	01000100	OUT	116	01110100
THIRTY	021	00010101	80MS SILENCE	069	01000101	OVER	117	01110101
FORTY	022	00010110	160MS SILENCE	070	01000110	PARENTHESIS	118	01110110
FIFTY	023	00010111	320MS SILENCE	071	01000111	PERCENT	119	01110111
SIXTY	024	00011000	CENTI	072	01001000	PLEASE	120	01111000
SEVENTY	025	00011001	CHECK	073	01001001	PLUS	121	01111001
EIGHTY	026	00011010	COMMA	074	01001010	POINT	122	01111010
NINETY	027	00011011	CONTROL	075	01001011	POUND	123	01111011
HUNDRED	028	00011100	DANGER	076	01001100	PULSES	124	01111100
THOUSAND	029	00011101	DEGREE	077	01001101	RATE	125	01111101
MILLION	030	00011110	DOLLAR	078	01001110	RE	126	01111110
ZERO	031	00011111	DOWN	079	01001111	READY	127	01111111
A	032	00100000	EQUAL	080	01010000	RIGHT	128	10000000
B	033	00100001	ERROR	081	01010001	SS	129	10000001
C	034	00100010	FEET	082	01010010	SECOND	130	10000010
D	035	00100011	FLOW	083	01010011	SET	131	10000011
E	036	00100100	FUEL	084	01010100	SPACE	132	10000100
F	037	00100101	GALLON	085	01010101	SPEED	133	10000101
G	038	00100110	GO	086	01010110	STAR	134	10000110
H	039	00100111	GRAM	087	01010111	START	135	10000111
I	040	00101000	GREAT	088	01011000	STOP	136	10001000
J	041	00101001	GREATER	089	01011001	THAN	137	10001001
K	042	00101010	HAVE	090	01011010	THE	138	10001010
L	043	00101011	HIGH	091	01011011	TIME	139	10001011
M	044	00101100	HIGHER	092	01011100	TRY	140	10001100
N	045	00101101	HOUR	093	01011101	UP	141	10001101
O	046	00101110	IN	094	01011110	VOLT	142	10001110
P	047	00101111	INCHES	095	01011111	WEIGHT	143	10001111

TABLE 22—DIGITALKER CONTROL

DB 0C	IN 0C
E6 01	AND 01
CA 82 01	JZ 82 40
06 40	LD B,40
0E 95	LD C,95
0A	LD A,(BC)
D3 14	OUT 14
D3 18	OUT 18
D3 1C	OUT 1C
C9	RET

program in line 50, which increments the variable "A" by 1. Line 60 sends program execution back to line 30 where the new value will be sent to the SPC and the next word in Table 21 will be spoken by the *Digitalker*.

When the synthesizer is controlled by the programs we have just discussed, it will speak, starting with the first word in the listing in Table 21, and say all of the words sequentially. You'll notice that although the last valid word has a numerical value of 143, the program sequences through 255—there are 112 undefined values which are spoken. Those undefined values generate some garbled sounds. But

continued on page 82

TABLE 23—AUTOMATIC DIGITALKER OPERATION

```
20 LET A=0
30 POKE 16533, A
40 LET B=USR 16514
50 LET A=A+1
60 GOTO 30
```

TABLE 24—MANUAL DIGITALKER OPERATION

```
20 LET A$=""
30 IF A$="" THEN INPUT A$
40 LET B$=A$(TO 3)
50 LET A$=A$(4 TO)
60 LET A=VAL B$
70 POKE 16514,A
80 LET B=USR 16514
90 GOTO 30
```

continued from page 72

try them anyway—we're sure you'll find some of them amusing.

For initial testing, you'll want the *Digitalker* to speak all of the words in its vocabulary. But if you want to use the synthesizer for serious applications, you'll need control over exactly what it says. The next program allows you to command the *Digitalker* to speak any sequence of words from the list in Table 21 so that you can program phrases or sentences. Enter the BASIC program in Table 24. Using that program, you can enter a word sequence as a series of three-digit values. All words must be represented by three digits: Any numerical value less than 100 must be padded with leading zeros. (For example, a 1 must be represented as a Ø Ø1.) The digits are entered in a series, and delimiters are not required between the three-digit groups. For example, if you want the *Digitalker* to say "The time is 1:30," the sequence you enter is:

138139096001021

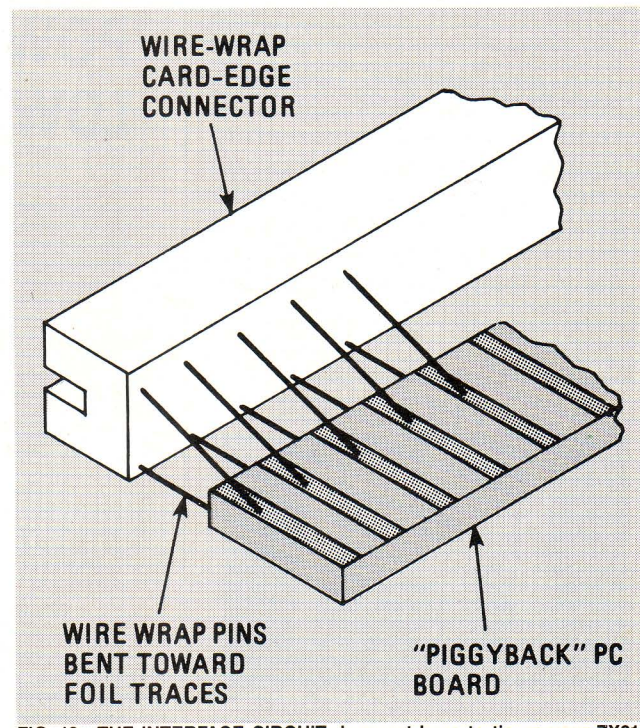
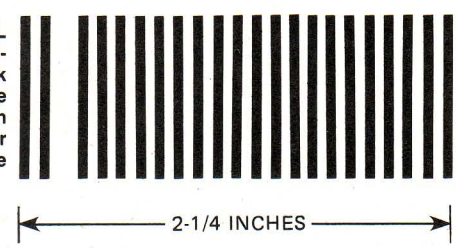


FIG. 13—THE INTERFACE CIRCUIT does not have to tie up your ZX81's port. An "extender" board will allow you to piggyback other devices to the interface.

FIG. 14—THE FOIL PATTERN for the double-sided piggyback extender board. Note that only one pattern is shown. The other side is, of course, the same.



The BASIC program divides the sequence into its three-digit word representations and sends them to the machine-language routine where the *Digitalker* is commanded to sequentially speak each word.

A piggyback connector

As you complete the interface projects we're sure that you'll discover that the 1K or 2K memory in your computer really limits

PARTS LIST—SPEECH SYNTHESIZER

Resistors, ¼-watt, 5% unless otherwise noted
R15—1500 ohms
R16—1 megohm
R17—820,000 ohms
R18—9100 ohms
R19—10,000 ohms
R20—10 ohms, ½ watt

Capacitors
C4, C5—50 pF, ceramic disc
C6–C9, C11—0.1 μF, ceramic disc
C10—20 μF, 10 volts, electrolytic
C12—0.047 μF, ceramic disc
C13, C14—.01 μF, ceramic disc

Semiconductors
IC20—74LS73 dual J-K flip-flop
IC21—74LS373 octal D-type latch
IC22—MM54104 speech processor IC
IC23—MM52164 SSR1 speech ROM
IC24—MM52164 SSR2 speech ROM
IC25—LM346 quad op-amp
IC26—LM386 audio amplifier

Other components
XTAL2—4 MHz
SPKR1—8 ohms

what you can do. To eliminate that problem, you might want to add a 16K or 64K memory pack. Both are commercially available for the Sinclair. The problem is that the interface circuit is plugged into the same slot that the memory pack must plug into. That problem can be solved by soldering a set of card-edge fingers onto the wire-wrap socket so that a memory pack can be piggybacked onto the interface circuit. Figure 13 shows how the card-edge fingers are attached to the wire-wrap socket. Figure 14 is the foil pattern for one side of the board. (The pattern for the other side is, of course, the same.)

The author of this four-part article would like to hear from any readers with interesting circuits to share, or anyone with questions regarding interfacing the *ZX81* or Timex Sinclair 1000. You can contact Neil Bungard directly at PO Box 493, Blacksburg, VA 24060 **R-E**

"We've added the accelerator card to the computer. Why do you ask?"