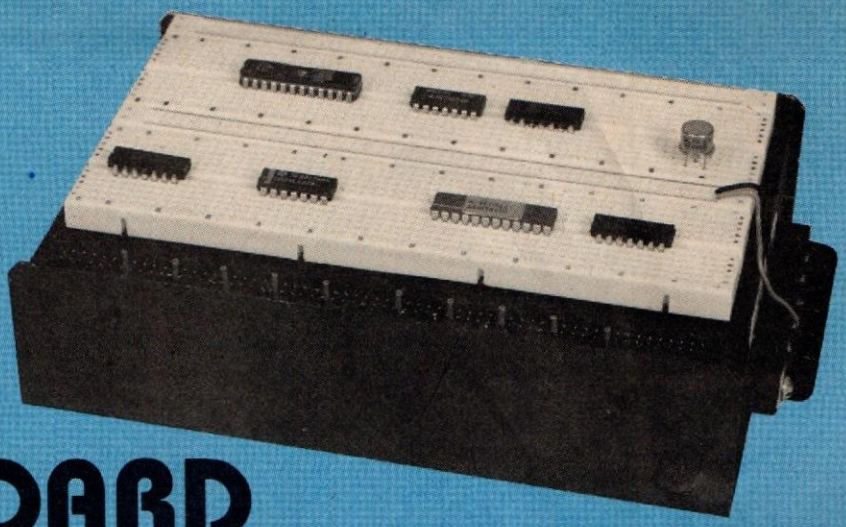
Putting last month's PC breadboard hardware to work.



BUILD THIS PC I/O BREADBOARD

DAVE DAGE

SOME PEOPLE MASTER HARDWARE, others master software—but few master both. Those who do, however, can expect great rewards. Last month's project was building the hardware: a PC interface card and a breadboard system with ten fully decoded and latched input/output (I/O) ports. This month's article is about the software required to make the hardware do something interesting. Along the way, you'll learn how to breadboard some hardware for demonstrating input and output capabilities, and you'll see software examples in three popular computer languages: BASIC, assembler, and C.

It will be helpful if you have had some experience with programming in at least one of those languages. But if not, don't worry—this presentation allows for inexperience. If you are just beginning to program, it's better to learn two—or more—languages at the same time. Most programming languages do the same things; learning several will help you place the statements and procedures of each in perspective.

Hardware setup

To begin, breadboard the circuit shown in Fig. 1. The schematic specifies the resistor networks, a DIP switch, and LED bargraph indicator, but any discrete components that are electrically equivalent can be substituted. Figure 2 shows how to mount the components on the breadboard.

The input circuit connects to port four, which appears at address 260 decimal (0104 hex) if you configured the interface card at the default base I/O port address. Resistors R2a–R2h pull all eight data lines low. By switching the poles of the DIP switch, you can pull each line high through resistors R3a–R3h.

The output circuit also connects to port four (260 decimal, 0104 hex). The output lines can drive each of the eight LEDs that are tied to ground through current-limiting resistors R1a–R1h. An LED will turn on when the associated data line goes high, and off when it is low.

Now let's see how to use the software to read the switches and light the lights. First I'll discuss the BASIC language, then assembler, and then C.

BASIC programming

BASIC, as supplied with DOS, is an interpreted language. When you run a BASIC program, you are running the large BASIC.EXE program, which takes each statement one at a time, "interpreting" what it means, and then executing it.

As with any programming language, BASIC has advantages and disadvantages. BASIC's advantages include simple setup; most housekeeping chores are handled for you automatically by the BASIC interpreter. In addition, testing and experimenting with BASIC is quick and easy because you can execute code in the immediate mode, rather than running "source code" through a program called a compiler.

BASIC also has disadvantages that include slow speed, awkward bit-level manipulation, and variables that are always global. An interpreted BASIC program is much slower than an equivalent compiled program in almost any other language. Bit-level manipulation can be troublesome for people who want to work directly with hardware. The problem with global variables is that simple typographical errors make it easy to create hard-to-debug problems, particularly in those large programs with lots of variables.

Despite those disadvantages, BASIC is ideal in situations where speed is not a requirement, the program is not large, and quick development and testing are paramount.

Listing 1 shows a simple BASIC program that will read the input port, transfer the data to the output port, and then repeat the process indefinitely. The overall effect of this program is that the switches directly control the LEDs. (Press Ctrl-Break when you've had enough.)

The objective of this program

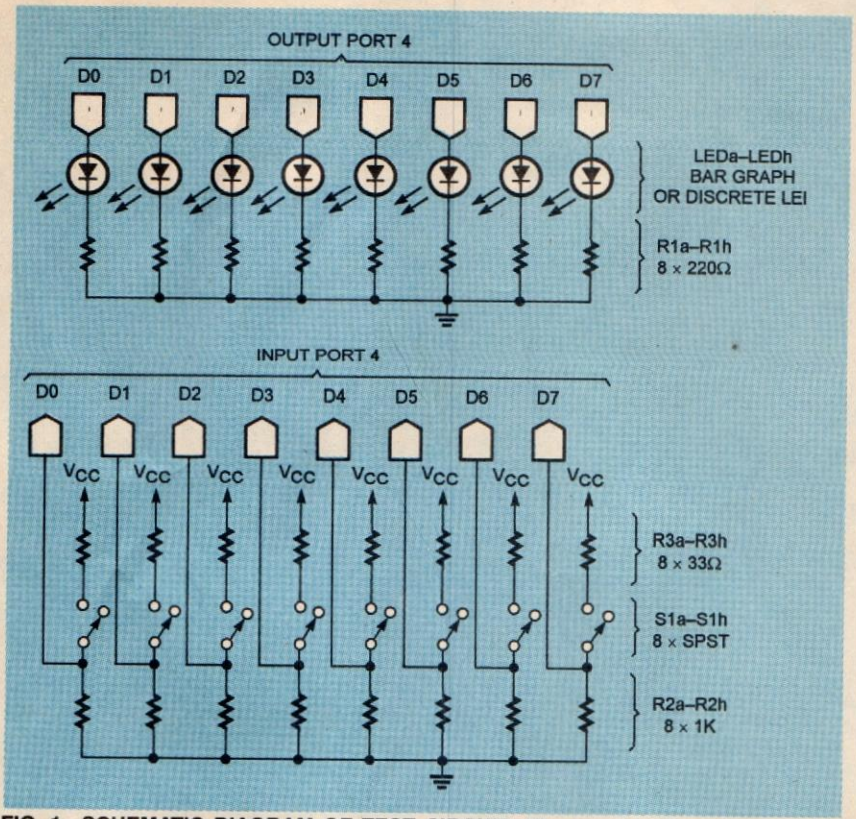


FIG. 1—SCHEMATIC DIAGRAM OF TEST CIRCUIT: The input port reads an eight-position DIP switch, and the output port drives an eight-LED bargraph.

is to show that the computer really controls the connection between the two ports. You could write a program that would switch a light on or off only after a password has been entered. Similarly, you could write a program so that the switch that controls a particular LED could be changed to control a completely different LED without rewiring the board. Try doing something like that without using software—it's next to impossible!

Another possibility would be to assign the input and output ports for completely separate purposes. For example, the DIP switch could serve as eight additional function keys for a special program, and the LEDs could function as a bargraph displaying, for example, the time remaining in some particular process.

Of course, it's also possible to have some fun with the circuit. Listing 2 provides one example. (Depending on the speed of your computer, you might have to adjust the value of the time delay in line 190.)

Assembly language programming

If BASIC represents one end of the programming-language spectrum, assembler or assembly language represents the other. Where BASIC is slow and cumbersome, assembler is quick and lithe. On the other hand, it's easy to perform tests and do experiments in BASIC,

LISTING 1—SIMPLE TEST PROGRAM

```
10 N = INP (260)
20 OUT 260, N
30 GOTO 10
```

but assembler requires careful forethought in planning those tasks. However, it's easy to control the hardware in assembler.

Every microprocessor has its own assembly language. Intel's 80X86 family has one, Motorola's 68XXX family has another, and so on. Moreover, within a given microprocessor family, successive additions to the family usually call for new instructions specific to that more advanced device.

Some microprocessors have a single address space that is occupied by both system random-access memory (RAM) and I/O ports. Other CPUs have separate locations for memory and I/O. Intel CPUs, for example, have separate memory and I/O locations. In this architecture, the CPU and I/O devices share common address and data buses; separate CPU control signals determine whether a given operation occurs in an address space or an I/O space.

In the 80X86 microprocessor family, different instructions allow the transfer of one, two, or four bytes of data simultaneously between an I/O port and the CPU's accumulator, or *A* register. The CPU can specify which port participates in a transfer in several ways. In assembly language, the port can be selected by an *immediate value*, which is actually part of the instruction the CPU executes. Another way to select a port is to preload the DX register with the address of the desired port, and then execute a slightly different instruction.

Figure 3 shows the register model for the 8086 CPU. Later members of the Intel family ex-

LISTING 2—LEDSWING.BAS

```
100 REM * for ST-1. Purpose: walk a single bit back and forth for display on
110 REM * a LED bar indicator.  Hardware: LED bar indicator, limit resistors.
120 REM * Note: TON (line 180) is loop delay, increase for faster cmptrs.
130 REM ************************************************************************
140 REM * AO address output port, TON time LED is on, TOFF time LED is off.
150 REM ************************************************************************
160 CLS:REM START
170 INPUT" Enter output port address in decimal ";AO
180 TON=500  : REM increase to slow cycling. Try 500 to 5000.
190 FOR C=1 TO 7
200 OUT AO,2^C: FOR T=1 TO TON:NEXT T
210 NEXT C
220 FOR C=6 TO 0 STEP -1
230 OUT AO,2^C: FOR T=1 TO TON:NEXT T
240 NEXT C
250 GOTO 190
```

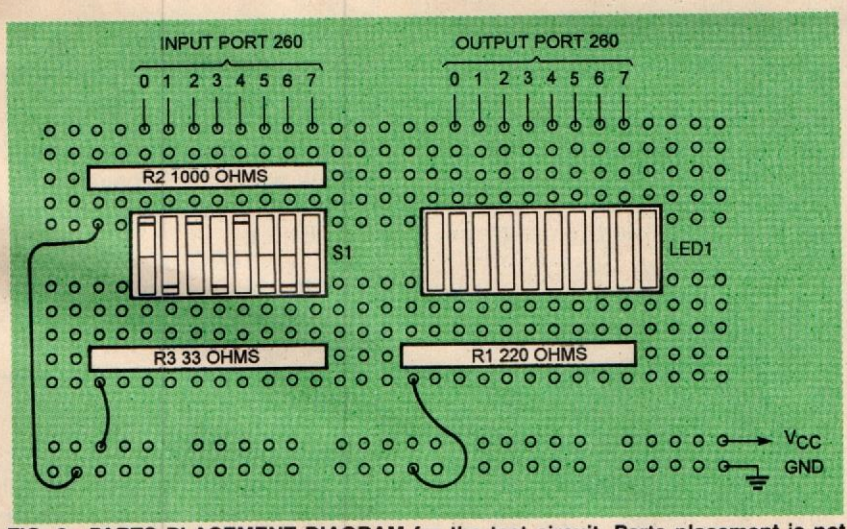


FIG. 2—PARTS PLACEMENT DIAGRAM for the test circuit. Parts placement is not critical; just be sure the power-supply polarity is not reversed.

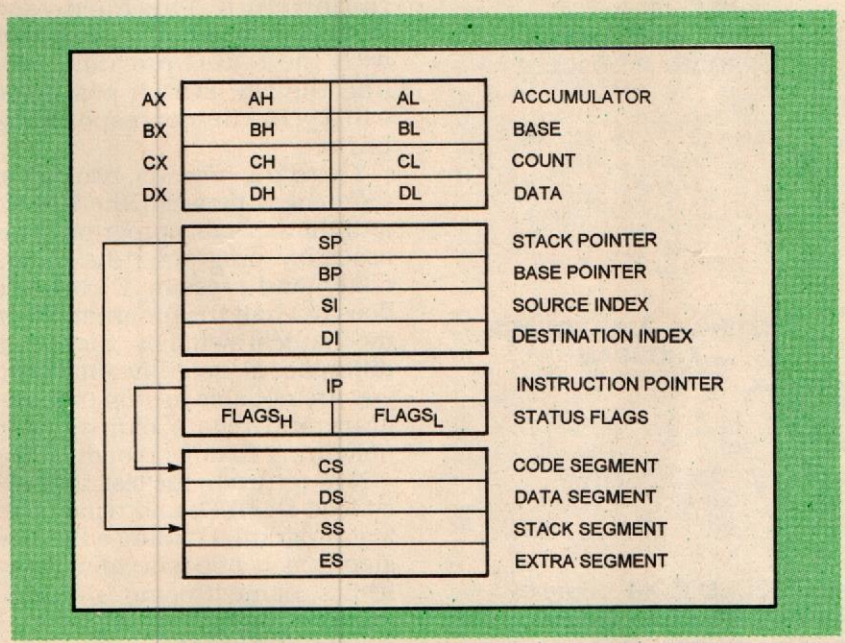


FIG. 3—INTEL MICROPROCESSOR REGISTER MODEL shows the 16-bit register format common to the 8088, 8086, 80186, and 80286 CPUs. The 386 and higher-level CPUs have 32-bit wide registers.

tend the architecture of the model, but all are backward compatible with it. Note that the main registers (AX, BX, CX, and DX) are 16 bits wide, but they can be addressed in 8-bit blocks: AL and AH (A low and A high), and so on. The 80386 and more powerful CPUs extend the concept to the 32-bit level.

The number of bits that can be transferred to or from an I/O port simultaneously depends on the CPU's capability, and on the interface hardware. This project's interface card works at the byte level, so only eight bits can be transferred at a time, through the lower half of the accumulator, designated AL.

The assembly-language instructions for reading and writing a port specified by the DX register are, respectively, IN AL, DX and OUT DX, AL. Those instructions assemble to hexadecimal machine-code values EC and EE, respectively. At this time you should know how to build an assembly-language version of our earlier BASIC test program. See Listing 3.

The only difference between the BASIC program and the assembly program is that the assembly program must preload the DX register with the desired hexadecimal port address. In Listing 3, the first instruction moves port address 0104h into the DX register. The second instruction transfers the data byte from the I/O port specified by DX into register AL. Then the CPU writes this same value back out to the same I/O port. Last, the program jumps back to line 2 to continue to the process indefinitely. Instead of using a GOTO statement with a line number, an assembly-language program uses labels such as "loop1" in the example.

To enter and execute an assembly-language program, you will need a program that can translate the assembler statements into machine code. This kind of program is called an assembler. A full-featured assembler such as MASM (Microsoft), TASM (Borland), or A86 (a shareware product by Eric Isaacson) will assign addresses to labels, keep track of data by name, and much more. However, for simple test programs, those programs are more comprehensive than necessary. For the purposes of this project, the DOS program DEBUG.COM will suffice.

Debug

A version of DEBUG is packaged with every copy of DOS. Unfortunately, it's one of the most user-*unfriendly* programs ever written. To make things easy on yourself, set up a separate directory (or use a separate floppy disk) just for DEBUG and the assembly programs that you will write. DEBUG is normally installed in your DOS directory, so you should be able to run it directly from your test directory.

Now execute the program. You should see its prompt, a simple hyphen. Press "A" followed by Enter. This puts DEBUG into its Assemble mode.

LISTING 3—ASSEMBLER TEST PROGRAM

```
        mov  dx, 104
loop1:  in   al, dx
        out  dx, al
        jmp  loop1
```

The program will display eight hexadecimal digits, divided into two groups of four separated by a colon. The value on the left is called the segment address, and the value on the right is called the offset address. The segment address, shown here as "xxxx" might vary from machine to machine; for small programs, its value doesn't matter. To determine the actual address associated with a segment:offset pair, multiply the segment by 16 (10 hex) and then add the offset. For example, hex address 2345:0006 = 23450 + 0006 = 23456.

The offset address, on the other hand, is critical. If you don't deliberately specify a value, DEBUG begins assembly at location 0100h (256 decimal). DOS allocates the first 256-byte block of memory within a given segment to store information about small COM programs of the kind being developed here. That block is a historical remnant from an earlier programming language, CP/M; it's officially called the PSP or Program Segment Prefix. Leave the PSP alone; don't try to place code or data there. (By the way. DOS's EXE file format does not use the PSP. In addition, COM programs are limited to 64K of code space, whereas EXE programs can be much larger.)

Intel CPUs have a set of registers called *segment registers:* code segment (CS), data segment (DS), stack segment (SS), and extra segment (ES). The segment registers typically function as indexes into various areas of memory. By default, DEBUG loads the current segment address into each of those registers.

Notice that the stack pointer (SP) is initialized at offset FFFE, and the instruction pointer (IP, also called the *program counter*) is set to offset 0100.

You now have a 64K block of memory ready for the entry of programs. Enter the code shown in Listing 4 at the DEBUG prompt.

Unfortunately, you cannot use symbolic addresses (e.g., the loop1 in the previous example) with DEBUG; you must enter the actual CPU offset address. That's why you see jmp 0103 at offset 0105. After entering the final instruction, press Enter again, and you'll return to the DEBUG prompt.

Before running the program, save it. First name the program with DEBUG's "n" command:

LISTING 4—DEBUG PROGRAM SW2LED

```
-a100 <Enter>
xxxx:0100 mov dx, 104 <Enter>
xxxx:0103 in al, dx <Enter>
xxxx:0104 out dx, al <Enter>
xxxx:0105 jmp 0103 <Enter>
xxxx:0107 <Enter>
-
```

LISTING 5—DEBUG PROGRAM SW2LED2

```
        mov dx, 104
loop1:   in al, dx
        out dx,al
        shl al
        jnc loop1
        mov ah,0
        int 21
```

LISTING 6—C LANGUAGE PROGRAM

```
#include <conio.h>
main()
{
    int inp(unsigned), outp(unsigned, int)
    while(1)
     {
        outp(260,inp(260));
        _asm mov ah, 11
        _asm int 33
     }
}
```

-n SW2LED.COM <enter>
Drive and path are optional, but for this project unnecessary, because the default directory has been specifically set aside for our test programs.

DEBUG's "w" command writes current memory contents to disk. However, you must specify the exact number of bytes in two registers: BX and CX. For anything less than 64 Kbytes, BX will contain zero and CX the remainder. The test program is only seven bytes long. So enter 0000 and 0007 into registers BX and CX as follows:
-r cx <enter>
CX 0000
:7 <enter>
-r bx <enter>
BX 0000
: <enter>
-

After you enter the first line ("r cx"), DEBUG displays the current value of CX and presents a colon (:) prompt. Enter 7 followed by Enter. Repeat the process with BX. To accept the currently displayed value, just press Enter.

The next step is to write the file to disk. Press "w" followed by Enter. Debug will respond with the number of bytes it is writing. Verify that the value displayed is correct:
-w <enter>
Writing 00007 bytes

Be certain the value is correct before proceeding—you'll see why in a moment. Now run the program by entering "g" (go) at the prompt. The hardware should respond just as it did with the BASIC program; cycling the DIP switch positions will cycle the corresponding LEDs.

There are, however, two major differences between the BASIC and the assembly program. The assembly program runs much faster, and pressing Ctrl-Break does not halt the program. With mechanical switches, the speed difference is insignificant, but if you are not able to stop the program and regain control, the only thing to do is reboot.

Now return to the test directory, load DEBUG, and reload the test program. Loading a file is a step that is opposite to writing it: first name it using "n," then load it using "l:"
-n SW2LED.COM <enter>
-l <enter>

How will you know that the correct program loaded? Try the unassemble command, "u." The assemble command used previously converts assembly-language instructions into hexadecimal bytes that are executed by the CPU, the unassemble command (sometimes called the *disassemble command*) converts hex bytes into assembly-language instructions that can be understood by people. Issue the following command to disassemble the test program:
-u 100 L7 <enter>

The "u" stands for unassemble, 100 is the starting address, and L7 instructs the program to

disassemble seven bytes beginning at that address.

DEBUG should display a list of program instructions identical to that in Listing 4.

Graceful ending

For the final assembly-language exercise, the program is made a little more "intelligent." The program can be allowed to end gracefully, so you don't have to reset the computer to halt the program. One way of doing it is to use DIP switch position 8 as a "break" key. After each pass through the read-switches, write-LEDs loop, check switch 8. If it's on, end the program; otherwise continue.

The shift instruction provides a simple way to do this in assembly language. You will shift bit 8 out of the accumulator and into a special register called the *carry flag*. As its name implies, the carry flag is normally used for arithmetic instructions. But it's also useful for determining the flow of a program based on the state of some condition—for example, the on or off state of a DIP switch.

Listing 5 shows how all these instructions tie together. As before, preload DX with 0104h, input the byte from the port at that address, then write it back out. Now for a surprise. The "shl al" instruction causes the CPU to shift the contents of its AL register left one bit position, moving the most-significant bit into the carry flag. The following instruction tells the CPU to jump back to the input instruction (at address loop1) if the carry bit is not set—i.e., if the switch is off. If this is not done, the program executes a special pair of instructions that will return control to the calling program.

Following the procedure previously outlined, enter the program in DEBUG, and save it to disk with a new name (SW2LED2.COM). Remember that you must enter the address of loop1 with the hex address, not the symbolic constant. Other than that, the program's operation is straightforward.

With this introduction, you should be able to read the documentation for DEBUG and learn how to use the rest of its commands. Although cryptic, DEBUG is a powerful tool for writing and debugging new programs and for exploring your computer system and its configuration. You might want to record all of the commands and parameters on a handy card for future reference.

The following information related to the speed difference between the assembler and BASIC versions of this program is interesting. On a standard 4.77-MHz PC, the BASIC program loop executes in about 2.25 milliseconds. By contrast, the assembler version takes about 9 microseconds. In other words, the assembly-language program ran about 250 times faster than the BASIC program!

GLOSSARY OF TERMS

Assembler—A computer program that converts or translates assembly language source code instructions into machine language.
Compiler—A computer program or circuitry that translates a high-level language into an executable program in a single operation. See *assembler* and *interpreter*.
Global variable—A variable in a computer program that can be shared by any object or subroutine within the program.
High-level language—An application-oriented programming language, as distinguished from a machine-oriented programming language. It is also termed a *computer language*. Examples are BASIC and C.
Interpreter—A computer executive routine that translates a program in high-level language or code into machine language or code. Unlike a compiler, the interpreter translates and executes one line at a time. See *assembler* and *compiler*.
Machine code—Instructions executed by a computer processor. It is also called *machine language*.
Machine instruction—An instruction written in a programming language that a computer can recognize and understand without translation.
Register—A circuit in computers or other digital circuitry that holds data in binary format for process or transfer.
Source language—The language in which a problem is programmed for a computer. It must be translated into an object program in machine language by an assembler, compiler, or interpreter.
Source program—A program that is written in source language or code.

C language programming

Earlier BASIC and assembler were defined as opposite ends of the programming-language spectrum. A broad range of other languages occupy positions all along that speed-performance spectrum. For example, C language has gained tremendous popularity during the past decade. Like BASIC, it is a high-level language, but it is really closer in many of its characteristics to assembly language. Indeed, C has been described as a "portable assembly language."

C is a compiled language like FORTRAN, COBOL, Pascal, and even some versions of BASIC. You start by writing source code that is similar to BASIC. However, C has no interpreter that runs it one line at a time. Instead, the source code is compiled into machine language that the CPU executes directly. As part of the compilation process, the compiler flags syntax errors (e.g., typos, undeclared variables, and misspelled language elements), that must be corrected before the machine code will be generated. Next, you have the option of linking the machine code with other predefined code libraries. When the process is complete, you will have a stand-alone file that will run from the DOS prompt.

Modern C development environments combine all the tools necessary for editing, compiling, and linking C code into a single, *integrated development environment* (IDE). The preeminent products in this category include Borland's Turbo C and Microsoft's Quick C. For entry-level programmers, an IDE is recommended. Even experienced programmers can significantly increase their programming productivity with an IDE.

Both Quick C and Turbo C include a special feature called *inline assembly*, which allows you to embed assembly-language programming instructions in the middle of a C program. Inline assembly thus gives you the best of both worlds: the low-level hardware

Continued on page 89

PC I/O BREADBOARD

continued from page 59

access features of assembly, with the high-level control and error checking features of C. (Note that Turbo C requires a separate product, Turbo Assembler, to make use of inline assembly; Quick C has everything you need built in.)

The final example, Listing 6, is a Quick C version of the switches-in, LEDs-out program. The first line of the program instructs the compiler to include a standard library of functions for controlling the keyboard and screen. Every C program has a *main* function, which begins on the next line. Within the main function, two integers are defined: inp and outp. They correspond to the input port and the output port, respectively. It will loop forever (or until someone presses Ctrl-Break) reading the input port, writing that value to the output port, and then checking for a key press entry through DOS.

The call to DOS has interesting features. The purpose of DOS function 11 (0Bh) is to report if a key press entry is waiting. When engaged in that function, it checks for a Ctrl-C or Ctrl-Break; either will terminate the program.

Wrapping up

The full potential of the computer (PC or mainframe) can be realized only with effective supporting software. Your success in any technical field today requires that you have a working knowledge of software. This has been a brief, and it is hoped, painless introduction to programming. You've seen that assembly language offers power and speed, but it requires a lot of planning and intimate knowledge of its host microprocessor to be used effectively. BASIC, on the other hand, is easy to learn and use, but it suffers from low speed, and is burdened with antiquated language constructs. For many people, C is the ideal compromise. It allows low-level access to the hardware when you need it, while simultaneously providing all the advantages of a high-level language. C is, however, more difficult to write (and read) than BASIC.

In the next part of this article, you will learn how to put the hardware and software to work in a practical project: a flexible, configurable EPROM programmer. The control software is in BASIC. Ω

ORDERING INFORMATION

The following items are available from DAGE SCIENTIFIC, P.O. Box 144, Valley Springs, CA 95252, (209) 772-2076:

- **Complete kit including manual and all parts (model ST-1)—$119**
- **Set of 2 PC boards and manual (model ST-2)—$40**

All orders add $3.95 shipping and handling. CA residents add sales tax.

WHAT'S NEWS

continued from page 85

3.8 volts, compared with the 1.2 volts of the other cells.

The solid lithium-ion cells can be stacked and connected together to produce batteries with higher voltages. Because the cells are thin and flexible, they can be formed into prismatic batteries of almost any shape needed. The battery form would not be restricted by the cylindrical shapes typical of other rechargeable cells. Bellcore says that its lithium-ion cells can be discharged and charged several hundred times with less capacity loss than the other rechargeable cells.

Lithium-ion batteries with liquid electrolytes have been on the market for only about two years, but they are still developmental models for limited applications. Japan started a national project to develop distributed lithium-ion battery storage technology in 1992.

Automated hotel check-ins

Hyatt Hotel Corporation expects that its "Touch and Go Check-In" machine will do for hotels what the automatic teller machine (ATM) did for banking. The check-in machine, which looks and works like an ATM, allows guests with reservations and credit cards to bypass lines at the front desk and check themselves into their rooms directly.

AN AUTOMATED HOTEL registration machine, part of Hyatt Hotel's Touch and Go Check-In system, speeds up guest registration and checkout.

Two machines are now being tested at the Hyatt Regency O'Hare hotel in a Chicago suburb and the Hyatt Regency Atlanta in Atlanta, with an eye toward their eventual use chain-wide.

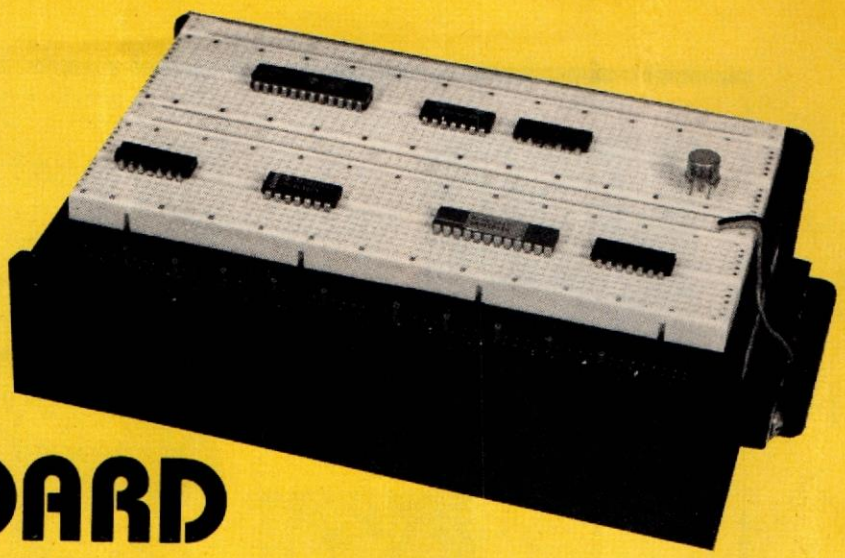
Upon arrival, a guest inserts his credit card into the machine. This causes his previous room, bed, and other check-in selections to appear on the machine's monitor for his approval. When this procedure is complete—typically in less than 90 seconds—the machine dispenses one or more room keys and a printed "passport" containing the room number.

At check-out time, the guest can use the machine to approve and pay for room charges that are displayed on screen. When that transaction is complete, the machine prints out a receipt.

Hyatt Hotel plans to expand its Touch and Go Check-In program to other hotels this year. A future service that could possibly be performed by the machine would be interactive participation in the selection and the making of reservations for local restaurants. Ω

Learn how fish with our EPROM programmer.

BUILD THIS PC I/O BREADBOARD



DAVE DAGE

GIVE A MAN A FISH AND YOU'LL FEED him for a day; teach a man to fish and you'll feed him for life. That old proverb can be applied to EPROMs, too. Give a man an EPROM programmer and he can program some devices. Teach him how to build a programmer, and he can program any EPROM he will encounter.

You can build a sophisticated EPROM programmer on your PC I/O breadboard with nothing more than a few jumper wires. The PC I/O breadboard was described in the June 1994 issue. An article in the July issue showed how to program it. The breadboard contains ten fully decoded and latched I/O ports, controlled by a simple PC interface card. This third and final installment shows that the breadboard is not just for building "toy" circuits, but can be put to real-world use as well. The EPROM programmer is fully functional, and can read, write, program, copy, and verify EPROMs ranging from a lowly 2716 all the way up to a 27512. The popular 27128 is used here as an example.

With what you'll learn here, adapting the program to new types of EPROMs will be simple. By the way, to make sure there are no misunderstandings, we'll use the expression *burn* to signify programming the EPROM, and *program* to refer to the BASIC software that controls the programmer.

Two notes about the software: 1. It will run under either GW-BASIC (supplied with MS-DOS prior to version 5) or QBASIC (supplied with DOS 5 and later versions). 2. The complete program is too long to print here. However, significant portions of the code will be explained, so that you'll know how it works. You can obtain the complete listing from the *Electronics Now BBS* (516-293-2283, V.32, V.42bis—the program is called EPROMBRN.BAS, and is part the PCIO.ZIP file), or with a kit of parts from the author.

Addressing the EPROM

First set up the hardware. Figure 1 shows the hookup between the ST-1 breadboard and the 27128 28-pin EPROM. The port numbers shown in the figure and in the program listings represent the decimal values of the default base I/O port (260) provided by the breadboard. If the board is not set to the default address, substitute the values in all diagrams and software. (Part 1 of this series detailed base I/O port selection.)

The 27128 has 128K bits of memory, accessible as 16,384 (16K) eight-bit bytes. Addressing 16K of memory requires 14 address lines ($2^{14} = 16,384$). Output port 260 drives the eight low-order address lines (A0–A7), and output port 261 drives the six high-order address lines (A8–A13), leaving the two highest bits of output port 261 unconnected.

Each successive byte in the EPROM can be addressed, starting at 0, as follows: Assume all 14 address lines are low. Increment the value at port 260 until all eight bits are high (i.e., the counter hits 255). Then put a 1 in 261, and a 0 in 260. Again increment through all values at 260, increment 261, and again put a 0 in 260. Continue in this fashion until the counter for 261 reaches 64, at which point you're accessing the highest address in the EPROM ($64 \times 256 = 16,384$). Then reset both counters to 0 and start over.

The way to do this in software is with a pair of nested FOR/NEXT loops, as shown in Listing 1. The inner loop, which addresses port 260, counts from 0 to 255; the outer loop counts from 0 to 63, addressing port 261.

This simple routine can be used for reading, verifying, burning, or copying the EPROM, by placing corresponding functions within the innermost loop (between lines 230 and 400). Note that the number at the head of each line is required only with GW-BASIC; neither compiled BASIC nor QBASIC requires line numbers.

Data flow

Addressing each location in the EPROM is easy. The next question is how to get data into and out of a specified location. A mini data bus is formed by connecting the EPROM's eight data lines to output port 262 and in-

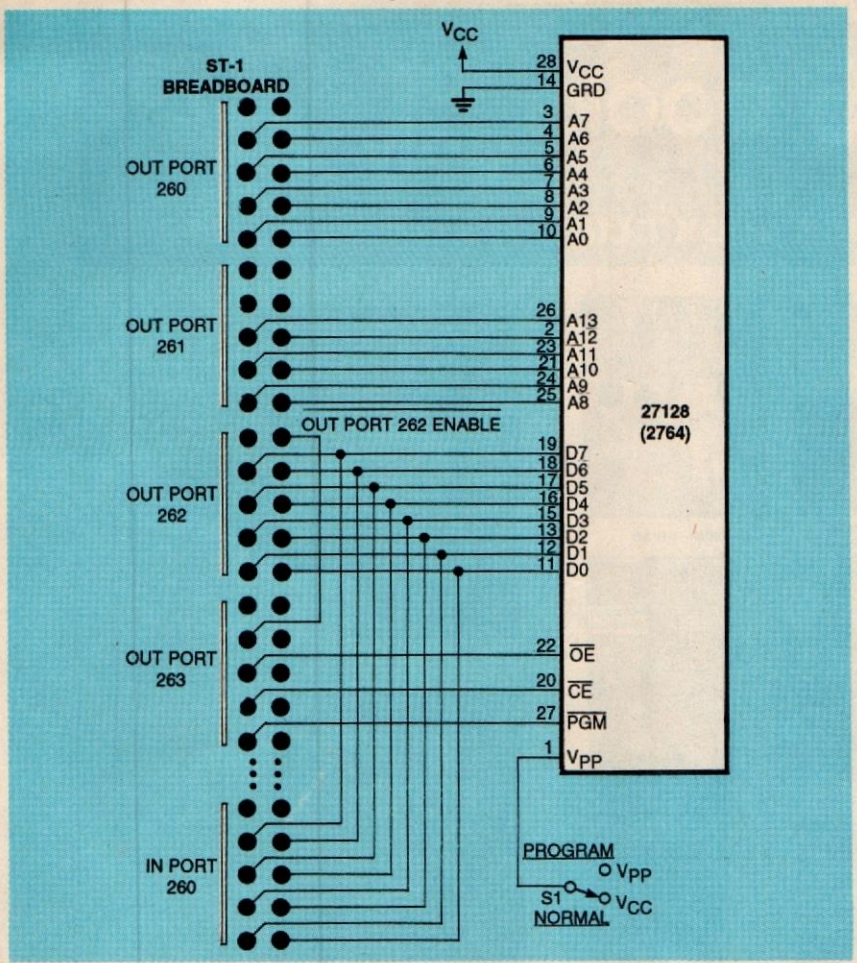


FIG. 1—27128 EPROM HOOKUP to the ST-1 breadboard appears here. The same hookup works for a 2764.

put port 260. The only "trick" is that the EPROM's outputs must never be activated at the same time as those of port 262. Fortunately, data flow can be controlled with just two pins: the EPROM's output-enable ($\overline{OE}$) line and the breadboard's enable line.

To read data from the EPROM, you must enable its output by bringing pin 22 low, while at the same time disabling port 262 by bringing its control pin high. Then BASIC's input port instruction (INP) can read data from the EPROM into variable V, as follows:

```
300 V = inp(260)
```

Conversely, output variable V to the EPROM like this:

```
310 out 262, V
```

Control lines

Of course, an OUT instruction by itself will not burn any data into an EPROM; the technique for doing so will be described momentarily. For now, note that port 263 drives the EPROM's output enable ($\overline{OE}$, pin 22), chip enable ($\overline{CE}$, pin 20), and program pulse ($\overline{PGM}$, pin 27), as shown in Fig. 1. Port 263 also controls port 262's enable line. The odd-numbered bits of 263 are configured as follows: D1 ($\overline{PGM}$), D3 ($\overline{CE}$), D5 ($\overline{OE}$), and D7 ($\overline{PORT\ 262\ ENABLE}$). All four signals are active-low. Bits D0, D2, D4, and D6 are not used, so they can be set to any arbitrary value. You might want to make them the complements of D1, D3, D5, and D7. Then, if extra hardware is ever needed, both the active signals and their complements will be available.

In the inactive state, all four control signals must be high. To determine what value to send to port 263, place 1's in bit positions 1, 3, 5, and 7, and 0's in positions 0, 2, 4, and 6. Convert the resulting eight-bit binary number (1010 1010) to decimal (170); that value must be sent to port 263 before the EPROM is installed. In fact, whenever a procedure completes, you should return output 263 to the inactive state, as follows:

```
100 out 263, 170
```

To read a byte from the EPROM, output enable ($\overline{OE}$) and chip enable ($\overline{CE}$) must be activated (pulled low). Plugging the two 0's in along with their complements gives 10010110 (150). Thus, to activate read mode:

```
150 out 263, 150
```

To burn a byte into the EPROM, the program prompts the user to apply the programming voltage, and then waits for a response. After receiving the user's response, the program enables port 262 and the EPROM's chip-enable line. Then (and only then) can it apply the 50-millisecond burn pulse.

Activating port 262 and the EPROM's chip-enable line equates to 01100110 (102). To perform the burn, that value

LISTING 1—EPROM ADDRESSING

```
200 for HI = 0 to 63
210  out 261, HI
220  for LO = 0 to 255
230       out 260, LO
...       some useful function
400  next LO
410 next HI
```

LISTING 2—EPROM BURNING

```
340 print "APPLY PROGRAMMING VOLTAGE NOW"
 ... wait for positive response
350 out 262, V rem out value to burn
360 out 263, 102 rem get ready to burn
370 out 263, 101 rem turn on PGM (burn EPROM)
 ... wait for 50 milliseconds
380 out 263, 102 rem shut off PGM
```

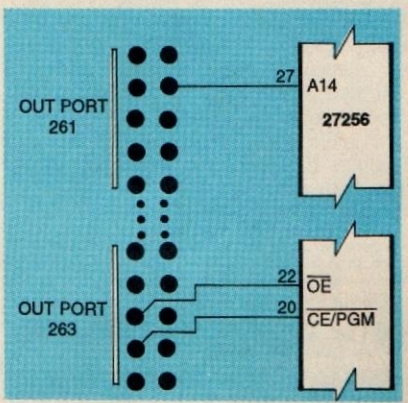


FIG. 2—27256 EPROM HOOKUP: Only the differences between the 27128 and the 27256 are shown.

must be changed to 01100101 (101) for 50 milliseconds, then back to 102. Listing 2 shows the entire sequence, except the 50-millisecond delay, which will be described next. For now, note than V_{PP} can remain on, and the actual value of the currently addressed byte in the EPROM can be read as described previously. If the written value equals the current value, programming was therefore successful, so the next address can be selected.

The 50-millisecond delay

The most difficult part of this whole project is generating an accurate time delay. The problem is how to guarantee the accuracy of the generated timing pulses. The simplest kind of delay is a do-nothing loop that increments a counter to some value. By adjusting the value, the delay can be made longer or shorter. That approach is not the best solution because it is CPU-dependent. That is, it depends on the type of CPU and its clock speed. Thus a program that runs fine on a 4.7-MHz PC might not do so well on a 66-MHz 486.

DOS interrupts are an additional complication. If an interrupt (e.g., for disk access) occurs during the loop, the time would increase unpredictably. The bottom line is that, as good as it is for other things, BASIC is just not suitable for generating accurate time delays. However, BASIC provides a fairly clean way of incorporating short assembly-language programs that can accomplish that type of task.

The assembly-language routine shown in Listing 3 combines line 370, the 50-millisecond pulse, and line 380 from the previous listing. First it turns off all interrupts, then it outputs the burn command (65

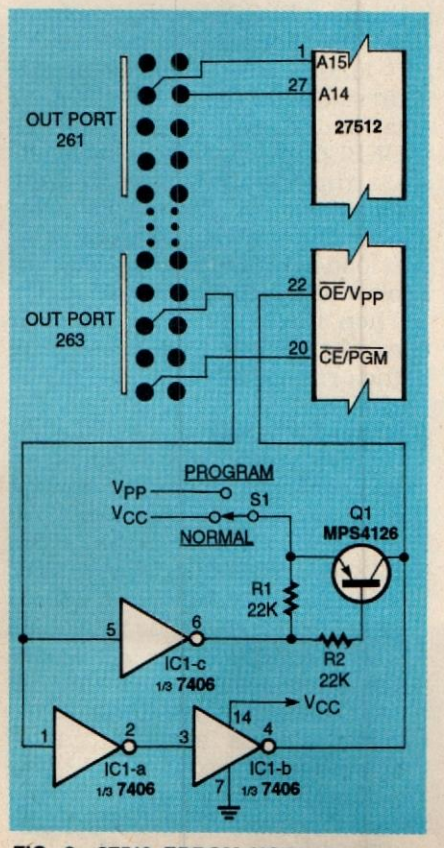


FIG. 3—27512 EPROM HOOKUP: Only the differences between the 27128 and the 27512 are shown.

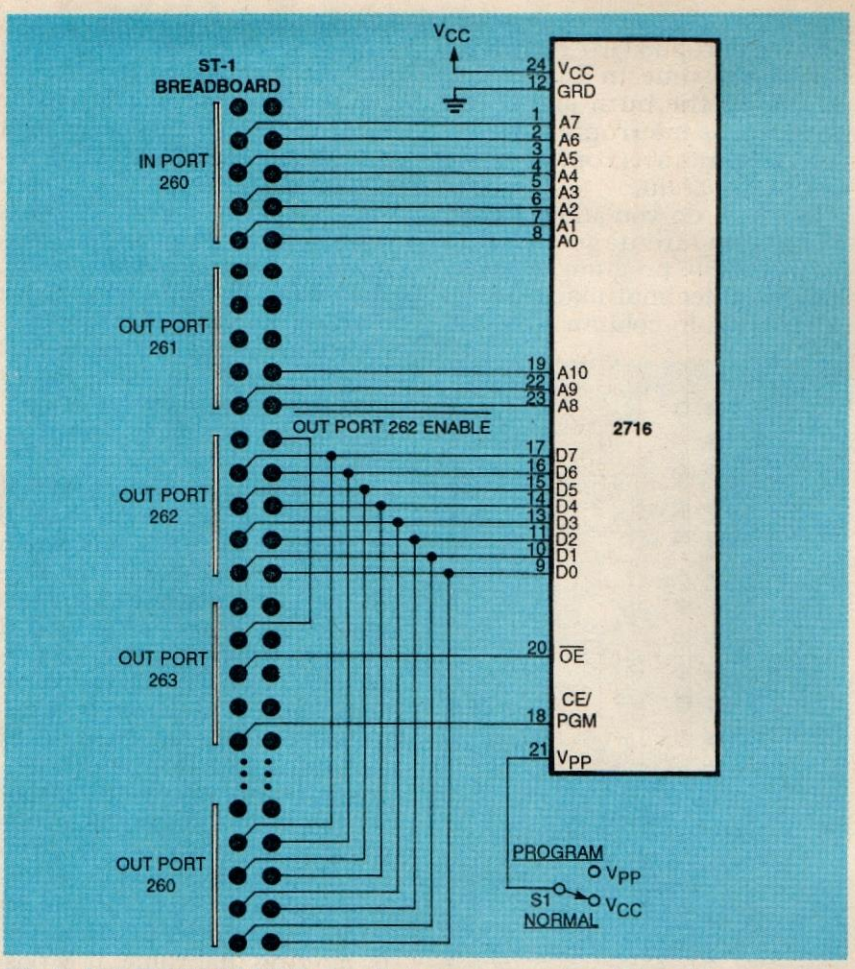


FIG. 4—2716 EPROM HOOKUP: Note that power moves to pin 24, and fewer address lines are needed.

LISTING 3—ASSEMBLY LANGUAGE DELAY ROUTINE

```
FA                  CLI             ; stop interrupts
BA0701              MOV DX,0107     ; control address in hex
B065                MOV AL,65       ; load control for pulse
EE                  OUT DX,AL       ; start PGM pulse
B81000              MOV AX,0010     ; outer loop start value
B90000    OUTLOOP:  MOV CX,0000     ; max value on inner loop
49        INLOOP:   DEC CX          ; start inner loop
75FD                JNZ INLOOP
48                  DEC AX          ; start outer loop
75F7                JNZ OUTLOOP
B066                MOV AL,66       ; load off pulse value
EE                  OUT DX,AL       ; turn off pulse
FB                  STI             ; allow interrupts
CB                  RETF            ; return
```

hex) to port 263 (107 hex). Next it wastes time in two loops, shuts off the burn signal (66 hex), turns interrupts back on, and then returns control to the BASIC program.

So how do you integrate an assembly-language program with a BASIC program? Convert the hexadecimal machine instructions in column 1 of Listing 3 to decimal, and then include them in the BASIC program via DATA statements. Then read the data into a pseudo variable, MC$. Then assign the variable BASE to where MC$ starts in memory. To execute the program, all you need to do is execute a CALL BASE statement. Listing 4 shows how it all works. Note that the loop values are defaults; the BASIC program pokes updated values into the appropriate locations, depending on the desired pulse length.

As the routine stands now, it will generate a delay of 5 to 6 seconds on an 8088, and much shorter delays on faster machines. The program calculates how many loops per second it takes to produce a five- to ten-second time delay on any specific machine. Then from that value, the program calculates constant LM (loops/millisecond). Do not alter your "turbo" switch after the program has calculated that value. The actual pulse length (in milliseconds) is stored in constant PL, and defaults to a value of 50. However, by altering that value, it is possible to generate any time delay of about 0.1 milliseconds or longer.

The only down side to that delay method is that your computer is almost entirely consumed with pulse generation. DMA is still active, but under normal circumstances, the only DMA channel that should be operating is that controlling memory refresh. At worst, memory refresh causes a one- to two-microsecond jitter in the delay pulse. Although it's viewable on a scope, that jitter is not a concern when burning EPROMs.

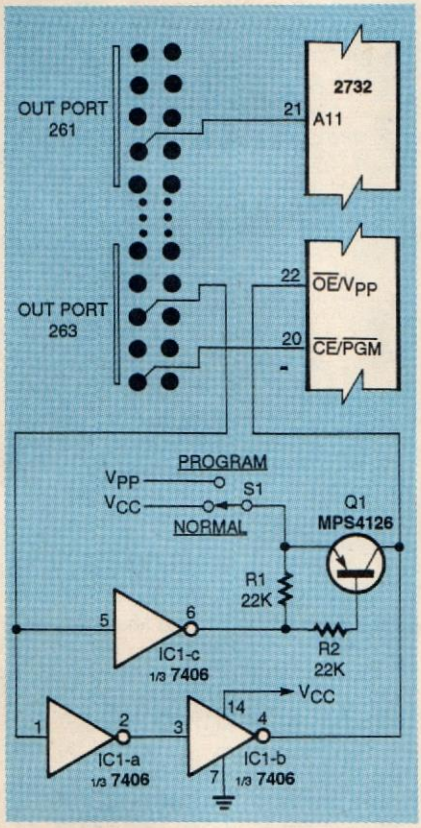


FIG. 5—2732 EPROM HOOKUP: Only the differences between the 2716 and the 2732 are shown.

LISTING 4—INTEGRATING THE ASSEMBLY LANGUAGE ROUTINE

```
7050 FOR X=1 TO 30 '# of code bytes to read
7060   READ N
7070   MC$=MC$+CHR$(N) 'generates string of machine code
7080 NEXT X
7090 PTR=VARPTR(MC$) 'use to find address of code
7100 BASE=PEEK(PTR+1)+PEEK(PTR+2)*256 'start address of assembly program
7110 NOINT=BASE+1 'skip 1st instr which disables interrupts
...
7140 CALL NOINT
...
7610 DATA 250 , 186 , 7 , 1 , 176 , 101 , 238 , 184 , 16 , 0
7620 DATA 185 , 0 , 0 , 73 , 117 , 253 , 72 , 117 , 247 , 176
7630 DATA 102 , 238 , 251 , 203 , 244 , 244 , 42 , 60 , 52 , 0
```

LISTING 5—ACCESSING DATA

```
180 def seg = &H60000 rem initialize segment pointer
190 out 262, 150 rem initialize to read EPROM
...  then "inside the address loops"
240 V = inp(260) rem read value
250 poke (HI*256 + LO), V rem stores value to memory
```

Data storage

Data that will be put into or taken out of an EPROM must be stored in memory—but where? BASIC doesn't permit dynamic memory allocation as C or Pascal do. So we have arbitrarily chosen absolute address 60000 hex. If you run BASIC on a machine without a lot of TSRs and device drivers installed, it will probably reside well below 50000 hex, including its stack. As long as you have at least 512K of memory, the area from 60000 to 70000 hex should be safe for storage. Admittedly, this is not the most elegant memory-allocation scheme, but it does work.

Data can come from many sources; however, the only source discussed here will be another EPROM. If you want to burn an EPROM with data from another source, you have to get it loaded into memory at address 60000 hex. After the data is loaded, you can use DOS's DEBUG program to modify it. Listing 5 shows how to copy data from an existing EPROM. After the contents of an EPROM is loaded into memory, it's a simple matter to save it as a file. Then next time it's needed, it can be reloaded without using the original EPROM.

Other EPROMs

Until now, this discussion has centered on the 27128. The good news is that the same principles also apply to the 2764, 27256, and 27512 devices, all of which are packaged in a 28-pin DIP. The 2764 can be viewed as half of a 27128. The 2764 uses the same wiring diagram; the only difference being that high-order address line A13 (pin 26) has no function.

On the other hand, moving up to the next largest device (the 27256) requires another address pin. Since all 28 pins are already used, the 27256 multi-

plexes the chip-enable line (pin 20) with the program pulse, assigning address A14 to pin 27. The only change to the program is that the initialization constant for a read (as shown in Listing 5) changes from 150 to 149. Figure 2 shows how to update the wiring diagram originally shown in Fig. 1 for a 27256.

Moving up to the 27512 requires yet another pin. This one combines output enable (pin 22) with V_{PP} and assigns address line A15 to pin 1. Pin 22 must now cycle between zero volts (output enable) and V_{PP}. To provide program control over that capability, additional components must be added, as shown in Fig 3. Four parts are required: an open-collector hex inverter (provided by a 7406), two 22K resistors, and a switching transistor. The EPROM read procedure is identical to that for a 27256, but the verify-during-burn procedure must be changed to first lower V_{PP} to zero. The verify sequence is as follows: 170, 150, 149, 150, 170. The burn sequence remains the same for all EPROMs.

The less-dense 24-pin EPROMs follow the same pin usage as outlined above, except that four fewer address lines are needed. The 2716 (shown in Fig. 4) follows the 27256, while the 2732 (shown in Fig. 5) follows the 27512. In addition, the 2732 requires extra control circuitry like the 27512. The only software difference lies in the address loop counters.

The complete program

Now you know how to "fish." But in case you're looking for a "canned" solution, a full-function BASIC program has been written for burning all the following EPROM types: 2716, 2732, 2764, 27128, 27256, and 27512. As with many programs, much of the code is concerned with the user interface. The program is menu-driven; it incorporates many error-trapping routines. You can download a copy of the program from the *Electronics Now BBS* as part of the PCIO.ZIP file. Try it, modify it, improve it.

Before starting the program, wire the ST-1 breadboard for the type of EPROM to be burned, but do not install the EPROM yet. Make a special directory for EPROM burning, and copy EPROMBRN.BAS, DEBUG.EXE, and GWBASIC.EXE into that directory. Make this your current directory and then run GWBASIC EPROMBRN from the DOS prompt.

The program begins by calculating loops/millisecond for your machine. This may take up to 20 seconds, so be patient. Remember not to change your turbo switch after the speed has been calculated. The program then presents you with a menu for choosing the type (size) of EPROM, and the burn voltage. The value you enter for size will affect the address loop constants, but the burn voltage value is displayed only as a reference. Note that EPROMs requiring a 12.5-volt burn voltage should work fine with the 12 volts available on the breadboard.

Next you'll see the main menu, which will allow you to copy, load, save, verify, and burn an EPROM. An additional option allows you to run DEBUG from within the programmer, for purposes of altering memory contents. Press the first letter of the desired command, and you'll enter the appropriate submenu. At this level, you must follow all entries with a carriage return. Typing an incorrect value or no value at all will return you to the main menu.

As discussed above, all data must be stored in memory at address 60000 hex. The typical method of obtaining that data would be to copy it from an existing EPROM using the copy command. Follow the screen commands by installing the source EPROM, then select Go. The entire EPROM will be copied into memory, and a simple arithmetic checksum will be displayed in both decimal and hex.

You can modify the EPROM values in that memory image using DEBUG. Return to the main menu and press D. The program will pass control to DEBUG. Now you can examine and change memory, using the procedures described in your DOS manual. When done, press Q followed by Return, and you'll fall back into the main menu.

Data stored in memory can be saved to a file by selecting Save EPROM from the main menu. Enter an eight-character filename with no extension; the program automatically adds the extension ROM. Regarding the save, you should know that BASIC can save a maximum of only 64K bytes of data, several bytes of which are reserved for header information. Hence when saving a 27512 EPROM (64K), the program saves the file in two pieces, one with the extension ROM, and the other with the extension RON.

After data has been saved to a file, it can be recalled with the LOAD EPROM command. Enter the filename (with the .ROM extension), or press L to list all (*.ROM) files in the current directory. If a program has been split, enter only the filename with the ROM extension; both parts will be loaded. Whenever the main menu is displayed, you can obtain a checksum of either the currently installed EPROM or the current memory data by selecting Verify.

With a valid data file in memory, you can burn an EPROM by selecting Burn EPROM from the main menu. The program will optionally verify that a device has been erased, then beep and request that you apply the selected burn voltage. Press B to burn the data into the device. The program burns the first byte and checks for success. If successful, the program continues on to the next byte, and so on until all bytes in that segment have been burned. Then

ORDERING INFORMATION

The following items are available from DAGE SCIENTIFIC, P.O. Box 144, Valley Springs, CA 95252, (209) 772-2076:

- Complete kit including manual and all parts (model ST-1)—$119
- Set of 2 PC boards and manual (model ST-2)—$40

All orders add $3.95 shipping and handling. CA residents add sales tax.

the segment number is incremented and the process repeats. After burning each segment, the program displays the remaining number of segments on the screen. That gives an indication that everything is working; it also helps you gauge how long the entire process will take. You can stop the program between segments by pressing S. After completely burning the EPROM (or after you press S), the program will beep to remind you to remove the burn voltage. Additional EPROMs may be burned using only this menu selection.

What next?

Although it's simple, the 50-millisecond programming method has its down side. It could take seven minutes to burn a 2764, and as long as an hour for a 27512. That may be acceptable for occasional use, but you may get impatient.

Most manufactures have developed fast programming algorithms, which typical commercial EPROM burners call High Performance Programming. This type of algorithm delivers a variable number of short (0.1–1.0 millisecond) pulses, with the number of pulses depending on when the to-be-programmed value changes. Higher voltages and closer tolerances are required for V_{CC} and V_{PP}. Writing the necessary loops is easy using GW-BASIC, as is generating the short pulses. The limiting factor is the overall slow-running interpreted GW-BASIC. Maybe now it's time to change to C. Ω

AMATEUR TV

continued from page 52

Next set both C19 and C26 on the power amplifier to mid positions (capacitor plates should be halfway meshed). Apply 13.8 volts DC to the transmitter. First adjust C19 and C26 quickly for maximum power output, and then adjust C13 and C14. Repeat these adjustments for maximum RF output. Repeat this with the higher crystal frequencies (434 or 439.25 MHz). If no output is seen at first, adjust C13 and C14 before readjusting C19 and C26.

At least 5 watts output should be obtained at each crystal frequency after their final adjustments. The power amplifier will draw about 700 milliamperes under these conditions. If 5 watts cannot be obtained, change the sizes of L8 and L9. Cut the loops with diagonal cutters, overlap the cut ends, and solder them together. If 5 watts is still not obtained, check C20 to C23, and all components in the collector circuit of Q9 for correct values. Expect the heatsink to become warm after about 5 minutes operation, but not hot enough to make it painful to touch. Vary R18; it should control the RF output smoothly from 0 to 5 watts.

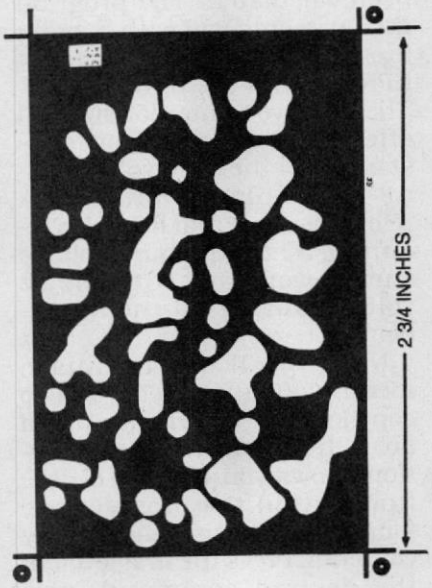
Note: You can skip this section if you are either building the 5-watt transmitter or any of the ¾-watt versions.

To check out the downconverter, first connect an RF probe, RF millivoltmeter, TV receiver, or monitor to the downconverter that will respond to 61.25 MHz (CH 3) or 67.25 MHz (CH 4). Next, apply +13.8 volts to both the junction of R43, C55, and R52, and the free end of R40. Verify the voltages given in Table 1. Set R50 so that about +5.5 to +6 volts is present on the wiper of R50.

Next, adjust C59 for a 378-MHz reading on the counter. Verify that R50 will vary the frequency from about 367 MHz to 382 MHz if channel-3 intermediate frequency (IF) is used. If a channel-4 IF is desired, set C59 for a frequency range of 361 to 376 MHz. A 15-MHz change in local-oscillator frequency is desirable, although 13 MHz will be acceptable. If you cannot obtain this reading, C 58 can be replaced with a 6.8 pF capacitor if a wider tuning range is needed.

Next, set trimmer capacitors C43, C46, and C51 to midway (capacitor plates half meshed). With the L.O. set at 378 MHz (channel 3 IF) or 372 MHz (channel 4 IF), apply a 439.25-MHz signal (±2 MHz) to the antenna lead. The signal level should initially be about 20 millivolts. Tune C43, C46, and C41 for maximum IF output at 61 or 67 MHz as seen on the indicator connected to the converter output. Reduce the input signal to the converter as required. Next, peak the slug in L20 for maximum output. This is a broad adjustment. You should obtain

Continued on page 74



COMPONENT SIDE of the Mini ATV board.

SOLDER SIDE of the Mini ATV board.