OMNIPROM

Bob Hart
2946 Merriman Road
Medford, OR 97501

Most early EPROM programmers were designed for one job and one type of EPROM. Even when the programmer evolved into a plug-in accessory to a microcomputer, it was seldom possible to program more than a couple of types. Now, because of the similarity of the available devices, the programmer itself can easily be programmed to accommodate most varieties of EPROMs.

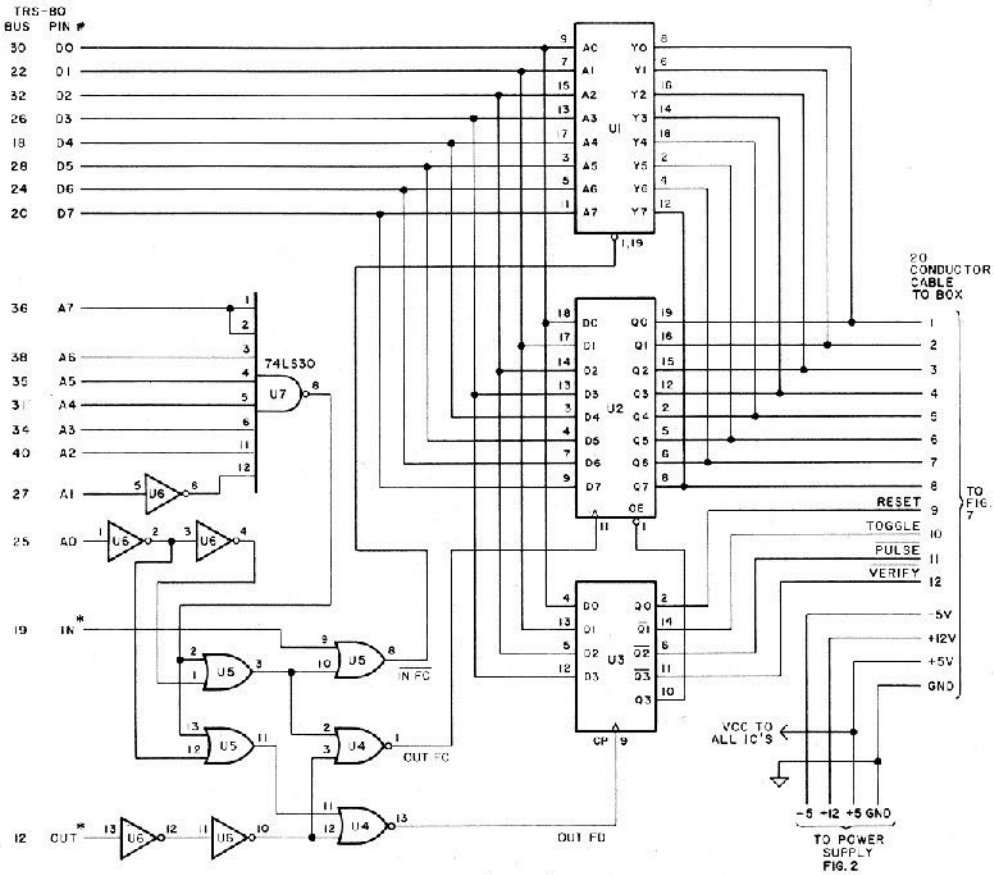
The System

OMNIPROM consists of three parts: a Z80-based computer, a control program, and a hardware interface for the EPROM. It can program 2704, 2708, 2716 and 2732 types, and should also work on 2764, TMS2716, 2532 and 2564 types. (I have not tried those.)

The system will verify that the EPROM is erased, copy data from another ROM or EPROM, program from any memory location, verify programming, and program the above chips. To connect the device to your computer you need one 8-bit input port, an 8-bit output port, and a 4-bit output port. These ports connect to a programming box that contains a 25-volt power converter (12 volts in, 25 volts out), an address counter, various control circuits, and a socket for the EPROM. The separate programming box allows you to work where it is convenient rather than trying to grope behind the computer to wherever the port interface is plugged in.

Input and Output

To write to the EPROM, eight bits of data have to be extracted from the computer's bus and presented to the EPROM's data pins. Alternately, to read the chip, the data on those same pins must be made available to the computer bus. For these



# TYPE	5V. PIN	GROUND
U1 - 81LS97	20	10
U2 - 74LS374	20	10
U3 - 74LS175	16	8
U4 - 74LS02	14	7
U5 - 74LS32	14	7
U6 - 74LS14	14	7
U7 - 74LS30	14	7

* INPUT AND OUTPUT - BYPASS +5 VOLTS WITH 0.1 μF DISK CAPACITORS.

Fig. 1. Input and Output

functions we need an 8-bit output port (for writing) and an 8-bit input port (for reading).

The largest EPROMs use 13 address lines. Rather than run all 13 back to the computer bus (through two additional output ports), I use a 12-stage counter in the programming box to provide address information. Only two controls are needed for this counter: Reset (start at address zero) and Toggle (go to next address). The toggle line provides the thirteenth address. Two additional controls complete the system: Program/Verify and Pulse. A 4-bit output port handles all control functions.

Circuit details for the interface board are shown in Fig. 1. Rather than stealing power from the computer, I provided the interface with its own source of +12 volts, +5 volts and −5 volts. The 8-bit input port (for reading) is U1 and the 8-bit output port is U2 (for writing). The control port is implemented using U3. The rest of the circuit is used for decoding the port addresses. In this case, the 8-bit input/output port is at address 252 (FC hex) and the control port is at 253 (FD hex). All connections to the programming box are made through a 20-conductor ribbon cable. This cable carries power as well as data to the programming box.

Programming Box

The other end of that cable connects to a small plastic box (see the photo and Fig. 7) containing the rest of the circuitry for EPROM reading and programming. All components are attached to the cover, allowing easy disassembly and repair. (You might want a slightly larger box—the parts are crowded into this one.)

Programming EPROMs requires a 25 or 26-volt power supply. To avoid building one more ac supply, I decided to construct a dc-to-dc converter. It runs on +12 volts and puts out a regulated +25 volts. From experience with other low power dc-dc converters, I was prepared for marginal operation at best. I was surprised. This circuit, adapted from National Semiconductor Application Note AN-183, is a winner. Using an inductor in a flyback circuit, as much as 60 volts at 100 ma can be produced. That's why a regulator circuit holds the output to the desired 25 volts. Although I am using 12 volts as the input to the converter, 5 volts would also work. This is especially handy if you use nothing but 5-volt supply EPROMs (anything but 2704, 2708 and TMS2716). In that case, only 5 volts need be supplied to the programming box.

The address data for the EPROM is supplied by a 12-stage CMOS counter (U10) and the Toggle input from the port board. Every time the Toggle signal returns to "one," the counter advances by one. This combination gives a total of 8,192 different addresses, sufficient for the largest EPROMs now available.

The rest of the contents of the box can be classified as control circuitry. A transistor switch controls the 25-volt programming signal to the EPROM (Q2-4). Control signals from the port board are buffered by Schmitt input inverters (U8) to help prevent noise problems. The buffered control signals are sent to a 16-pin IC socket. This socket is not for the EPROM but for a module that programs the programmer for a particular EPROM type. The modules, diagrammed in Fig. 3, are built on 16-pin headers. You don't have to build all of them. Find out which EPROMs you will be programming and assemble only the ones you need. I'll go over the design of new modules in a later section.

There are two switches and two indicator lamps on the box. The first switch (S1) controls the low voltage power to the programming socket. The second (S2) enables the high voltage (25 volts). Both prevent damage to the EPROM as you insert and remove it from its socket. The indicator LEDs show which switches are on.

The last item on the box is a zero insertion-force socket. A 49-cent socket would probably work in its place, but would make it easier to bend a pin or break an expensive EPROM. The more expensive socket (about $10) is a pleasure to use. You just drop the chip in place and flip a lever to make a secure connection to all 24 pins. Moving the lever back up allows you to lift the device out—no strain on your nerves or on that EPROM.

Testing

Once you have the programmer assembled, check it for correct operation. Since this is a computer peripheral, the computer can do most of the checking. First, apply power to the unit (don't connect it to the computer yet) and make all the usual power supply and mis-wiring checks. When everything appears fine, connect the programmer to the expansion bus and run the program in Listing 1.

The program first checks the address counter. Turn on the programming box power switch. The power indicator should be on (see Fig. 4). A logic probe or scope should show all address lines toggling from logic high to logic low with each succeeding line switching at a lower rate (A0 is fastest and A12 is slowest). If there is no change on any line, check the interface address decoder and the wiring to the address counter. If some appear out of sequence, the fault is in the wiring from the EPROM socket to the counter. Press X and the program resets all address lines to zero (check them) and proceeds with a test of the data lines.

The computer is merely reading port 252 and displaying the decimal equivalent on the screen. With all data lines floating, the screen fills with 255s. Connect one end of a jumper to ground (EPROM socket pin 12) and the other end to each data line in sequence. If everything is normal, the screen displays the numbers in Table 1.

If you see nothing but 255, check that the jumper wire is tied to ground. It could also mean the address decoding circuit or the input latch (U3) is not operating. If the displayed num-

bers change but are incorrect, check the data line wiring at the input gate and to the programming box for shorts, opens or transpositions. Press X when you are satisfied everything is all right.

The next test checks data output. The computer is writing to port 252 (output) and reading port 252 (input). If it reads what it wrote, it keeps on checking. This automatic test stops if it finds an error. If you put your logic probe on the EPROM socket data lines you will see all data lines switching (01 fastest and 08 slowest). Hold X to abort this test.

Ground Pin	Display
9 (D0)	254
10 (D1)	253
11 (D2)	251
13 (D3)	247
14 (D4)	239
15 (D5)	223
16 (D6)	191
17 (D7)	127

Table 1

The last test sequence verifies the operation of the rest of the control signals (half were checked when we clocked and reset the address counter). Prepare a programming module for a 2708. Plug it into the programming box and turn on both switches. Both indicator LEDs should be on. Using a voltmeter with a 25-volt range, measure pin 7 on the module. It should be 12 volts. Press and hold 3 (verify) on the keyboard and the voltage should drop to zero. The same signal should be present on pin 20 of the EPROM socket.

Now shift the voltmeter probe to pin 5 of the module. It should read zero. Press and hold 2. The voltage should rise to 25 volts. Adjust R1 if this terminal is not at 25 volts. The signal is also present at pin 18 of the EPROM socket. For the last test, press 2 and 3 at the same time. Both pin 5 and pin 7 on the module should be at zero volts. You might check the rest of the EPROM socket pins for the correct supply voltages for a 2708 type. Any discrepancy would indicate a wiring error from the module socket to the EPROM socket.

If you've made it this far, you have a potentially functional EPROM programmer. For my peace of mind, I usually run the tests before each programming session. All tests are done to make sure the programming software can properly control the programmer. Without control, the hardware has the capability of turning an EPROM into slag. For this reason, do not plug a device into the programming socket unless the control software is running and the programmer box checks out.

Line Numbers	Function	USR Calls
10	Jump to machine-language loader	
20-320	Initialize memory and variables	
330-440	Function menu	
450-510	Type selection	
520-650	Read EPROM	USR0: uses Basic variables S and L
660-760	Move memory	USR1: var. S, E, F
770-830	Verify EPROM	USR2: var. L
840-950	Error codes	
960-1020	Check for erased	USR3: var. L
1030-1090	Program EPROM	USR4: var. L
1100-1110	DOS exit	
1120-1210	Hex to decimal	
1220-1310	Decimal to hex	
1350-1460	Modify memory	
1470-1560	Disk read	USR5: var. S, E, F
60000-end	Machine-language loader and data	

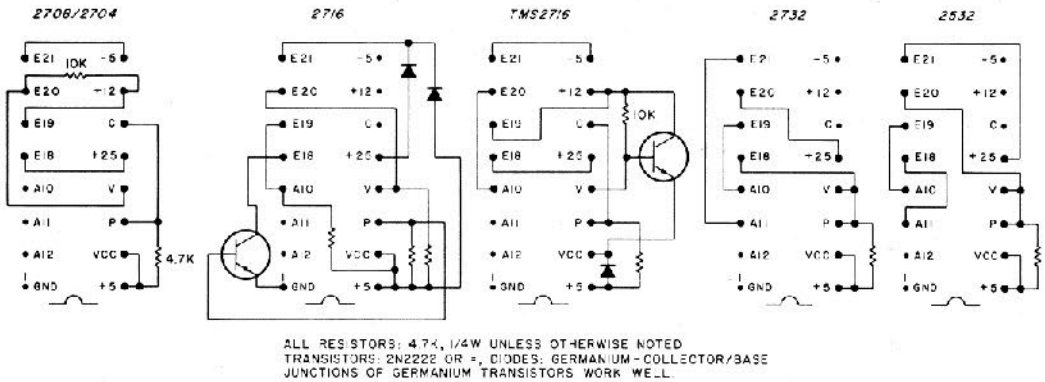
Table 2. Basic Line Number Map

Controlling the Box

I started writing the control program in machine language, but quickly came to the conclusion that most functions could be more easily and efficiently done in Basic. So the program has two parts: a Basic program that interacts with the user, and a machine-language program that interacts with the EPROM programmer hardware. The machine-language part, shown in Program Listing 2, is only there if you care to find out how it runs. The Basic program (Listings 3 and 4) actually contains the machine-language code and will load it in high memory and set memory size to protect it. Program Listing 3 starts with line 60000 and is the relocating machine-language loader. Type Listing 3 into the computer first. Use the same line numbers as listed and run it once. If all is normal, a message directs you to delete several lines. After you have done this, immediately save this program segment. Do not save it in ASCII form; a regular disk save will do just fine. This procedure is necessary to initialize the relocating machine-language loader and will only be done once. Now you are free to add Program Listing 4 to what is left of Listing 3. The only restriction is that line 20 has to remain as line 20. The rest of the line numbers can be anything less than 60000. (Lines greater than 60000 are deleted by the loader.)

Program Features

The data statements starting at line 210 define each type of EPROM to the software. The first statement tells the program how many types have been de-



ALL RESISTORS: 4.7K, 1/4W UNLESS OTHERWISE NOTED
TRANSISTORS: 2N2222 OR =, DIODES: GERMANIUM - COLLECTOR/BASE JUNCTIONS OF GERMANIUM TRANSISTORS WORK WELL

Fig. 3. Programming Modules for OMNIPROM

fined. Set it accordingly.

The first data entry is the type name (2708 or TMS2716, for example). The next number is the size of the EPROM in bytes minus one. Following that is the length of the programming pulse in 500-microsecond increments. Last is the loop count, the number of times the system will program each EPROM byte (more on this later). By adding data statements (and also constructing a new programming module) new types can be easily introduced.

All EPROMs are programmed from a fixed-location buffer in RAM memory. Included in the controlling software are functions to move information to this buffer. The size of the buffer is great enough to allow the largest EPROM to be programmed in one pass.

There is some disagreement between EPROM manufacturers about the length of the programming pulse. For the 2732 in particular, Intel claims a 50-ms pulse is required for each address. Motorola says only 2 ms are necessary. I have compromised. My programming algorithm uses the shorter pulse width, but then does a data verify after all bytes have been programmed. If the verify fails, the EPROM is programmed again (and verified). This continues until the EPROM is correctly programmed or the total pulse time for each address is greater than the longer specified time. If the latter is the case, an error is returned (Won't Program). Using this technique, I have programmed various types much quicker than normal.

I have done some tests on data retention and found there is no difference in erase times whether the short or the long programming pulse is used. To set up the system for short-pulse interactive programming (of the 2732), set the pulse width to four and the loop count to 25. If you desire long-pulse programming, set pulse width to 100 and loop count to one. In any case, the product of the two numbers should equal twice the maximum pulse width in milliseconds. Caution: Some EPROM types (for example,

2708) require multiple passes. Long-pulse, single-pass programming will destroy these types!

The verify function makes two different checks of data validity. It first makes the usual comparison check of RAM data to EPROM data. If that works out, a checksum is done over the entire contents of the EPROM. The checksum is then compared to one originally made when data was last moved to the buffer. If there is a problem with the checksum, one or more of the bytes in the buffer changed and you must reload the buffer and reprogram the EPROM.

"Some EPROM types require multiple passes. Long-pulse, single-pass programming will destroy these types!"

A map of the Basic program (Table 2) gives the range of line numbers for various functions and also shows the USR links to the machine-language portion.

When the program starts, it displays a menu of functions. When any function is completed, you are prompted to return to the menu. The functions in order of their appearance are:

- Set Type—Until you specify the type of EPROM, none of the other functions will operate. It's just as well. Without knowing the type, the program doesn't know what to do.
- Erased—Reads EPROM and checks that each location is 0FFH. You usually want to start with an erased chip before programming. They don't always come that way from the factory.
- Read PROM to Buffer—This function reads the data from the device plugged into the box and places it into memory. The program will ask for the memory location of the buffer. Answer with a hex address or just hit enter. If you do the latter,

programming and verifying will be done from the standard buffer. If you specify an address, it must be within the limits displayed on the screen.

- Move Memory to Buffer—specify the start and end locations in hex. The EPROM offset allows you to modify the buffer starting at any location. Zero offset loads to the beginning of the buffer. The offset can also be used to string data together from several memory locations. Move one batch at a time, setting the offset for the next group to one greater than the length of the previous group. Most of the time you'll be providing all of the data from one location, so the offset will usually be zero.

- Modify Memory—Usually used to manually enter data to the buffer. Specify a start location (in hex) and press enter. The memory location is displayed (in decimal) followed by the byte at that location (in hex). Enter the new byte value in hex and terminate the entry with enter or skip over the location by just pressing enter. Exit this mode by pressing X. If you want hex addresses and are not worried by how long it takes, hold the shift key while you press enter.

- Disk Read—This function reads an object file from disk and loads it to any place in memory. Be extremely careful about the destination in memory. The free areas are displayed but there is nothing to prevent you from loading to an otherwise occupied area (you could overwrite DOS or Basic). Both starting and ending addresses have to be entered in case the disk file is longer than you expected. If there is an error in accessing the disk, a number is displayed. That number is the DOS error code listed in the TRSDOS manual.

- Program—When all data in the buffer is correct, get your EPROM ready and follow the prompts. During the programming routine, a symbol in the upper right of the screen will change. This lets you know that something is going on. After the function is complete you can program another EPROM without reloading the buffer, since nothing in the buffer is changed by the program function.

● Verify EPROM—Compares data in buffer with EPROM data. If they don't match an error message is returned. If all data matches the routine returns with "Function Complete." This routine is performed automatically after (and during) programming.

● Exit to DOS—You might want to change this to exit to your favorite machine-language monitor. I wanted quick access to DOS to do dumps of EPROM data. To return to the EPROM programmer, load and run the software again.

"I wanted quick access to DOS..."

I have skipped cassette data sources. If you need one, first transfer it to a disk file. If you need that function, it could be added—four USR calls are left.

Transferring data from one type of EPROM or ROM to another can also be done. The easiest method is to set the type for the first ROM and then read it into the standard buffer. Reset the type to the other and program. Resetting the type will not affect the data in the buffer, but you will get a checksum error after programming if the EPROM lengths are different. To avoid the error, load the source EPROM to free memory and then change the type. Do a memory move to the standard buffer and program the destination EPROM.

Most of OMNIPROM's functions are self-prompting and allow you to bail out to the menu at the last moment. To reduce the effects of the sweaty palm syndrome, let your first EPROM be an inexpensive one (2708s are good).

Programming the Programmer

One problem associated with building a piece of equipment such as this is the certainty of obsolescence. The EPROM itself is now semi-obsolete since new Read Mostly Memories can be electrically written in circuit and retain data with the power off. CMOS RAMs can be kept alive for months with a 2-volt battery. Despite these newcomers, the EPROM is still the simplest (at least in the read mode) and least expensive way to retain semi-permanent data. Any programmer for EPROMs must be flexible enough to forestall the day when it too must be thrown in the junk bin. How useful do you think 1702 programmers are today? Programming the programmer is a way to gain this flexibility.

Programming is done by plugging a specially wired DIP (Dual Inline Package) header into the module socket. You can wire one of those in Fig. 3 or design your own. The design requires that you know the function of the module pins and the requirements of your EPROM. Let's cover the module pins first.

Most pins to all 25xx and 27xx series EPROMs have the same functions. The five pins that have different uses on different devices are brought out to five pins on the programming module (E18, E19, E20, E21 and E24). The rest of the module pins provide control signals and power supply voltages. All this is shown in detail in Table 3.

The V pin is a transistor switch that goes to ground

Pin Number	Name	Explanation
1	Gnd	Signal and power ground
2	A12	Address 12 from counter
3	A11	Address 11 from counter
4	A10	Address 10 from counter
5	E18	From pin 18 of EPROM socket
6	E19	From pin 19 of EPROM socket
7	E20	From pin 20 of EPROM socket
8	E21	From pin 21 of EPROM socket
9	−5	Negative 5 volt source
10	+12	Positive 12 volt source
11	C	Ground this pin to turn on 25 volts
12	+25	Programming voltage source
13	V	Goes low for Verify
14	P	Goes low to program
15	Vcc	To pin 24 of EPROM socket—usually Vcc
16	+5	Positive 5 volt source

Table 3. Programming Module Pins

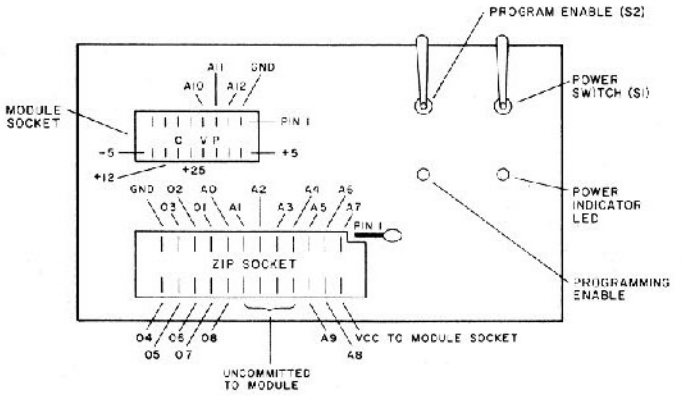


Fig. 4. Programming Layout and Permanent Pin Wiring

		PIN 18	PIN 19	PIN 20	PIN 21	POWER	NOTE
2704 (512 × 8)	READ	0	+12	0	-5	±5, +2	1
2708 (1K × 8)	WRITE	+26 500μS 0	+12	+12	-5		
2716 (2K × 8)	R	0 - $\overline{CE}$	A10	0 - $\overline{OE}$	+5	+5	1
	W	+5 2-50mS 0	A10	+5	+25		
TMS2716 (2K × 8)	R	0 - $\overline{CE}$	+12	A10	-5	±5, +12	2
	W	T25 500μS -0	+12	A10	-5	+12 TO VCC PIN	
2532 (4K × 8)	R	A11	A10	0 - $\overline{OE}$	0	-5	2
	W	A11	A10	+5 2-50mS 0	+25		
2732 (4K × 8)	R	0 - $\overline{CE}$	A10	0 - $\overline{OE}$	A11	+5	1
	W	+5 2-50mS 0	A10	+25	A11		
2764 (8K × 8)	R	A11	A10	0 - $\overline{CE}$	A12	+5	2
	W	A11	A10	+25 2mS +5	A12		
2564 (8K × 8)	R	INFO NOT AVAILABLE					
	W						
2758 (1K × 8)	R	0 - $\overline{CE}$	0	0 - $\overline{OE}$	+5		2
	W	+5 50mS 0	0	+5	+25		

NOTES 1 - DEVICE HAS BEEN PROGRAMMED ON OMNIPROM USING THIS DATA

2 - INFO EXTRACTED FROM VARIOUS MANUFACTURES SPEC SHEETS - HAS NOT BEEN VERIFIED IN USE.

Fig. 5. EPROM Parameters

whenever the control program sets verify true (we want to read the EPROM). The P pin is a similar switch that goes to ground whenever the computer requires a program pulse. If the C pin is pulled to ground (perhaps by the P pin) and at the same time verify is off (we want to write), 25 volts are switched to the 25V pin. The rest of the module pins include three additional address sources (A10, A11 and A12) and the three power supplies (+5 volts, +12 volts and −5 volts). The only problem remaining is how to interconnect all this stuff.

Pin E24 is the easy one. In most cases, it is the Vcc supply pin and will connect to +5 volts. In the TMS2716, +12 volts is connected there during programming. Connecting the other four E pins is somewhat more complicated. To start, prepare a chart similar to Fig. 5 listing the state of pins 18–21 in the read and program modes, the power requirements, and any other relevant information. It's a good idea to use the manufacturer's spec sheet for each

Continues on p. 278

Continued from p. 274

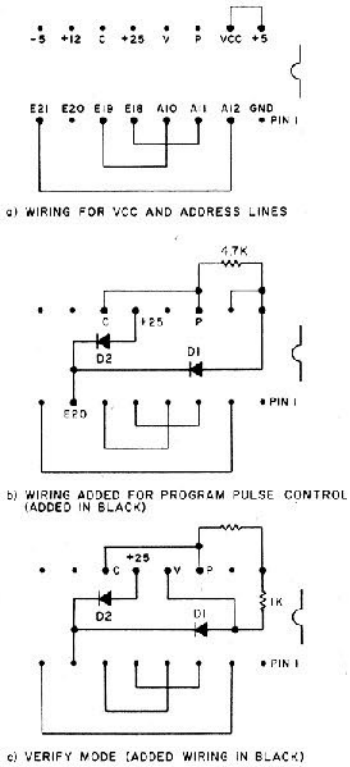


Fig. 6. Developing a programming module for the 2764. Use only after checking with manufacturer's specs. This circuit has not been tested.

EPROM to prepare the chart. Once you have the information and you know it is accurate, you can make up the module wiring configuration. To show you how this is done, let's take the information on the 2764 (see Fig. 5) and design a programming module.

The 2764 is an 8K by 8 memory device and as such requires three more address lines (the programmer's permanently connected address lines will only address 1K). From Fig. 5 you see that A10, A11 and A12 are connected to pins 19, 18 and 21 respectively. So on the module, connect A10 to E19, A11 to E18, and A12 to E21. Also since this chip uses 5 volts for power, connect +5 to Vcc.

See Fig. 6—it appears all controls have to act on pin 20. To read the chip, pin 20 must be at ground (0 volts). To program a location, a +25 volt pulse is applied to the same pin. Between programming pulses, pin 20 must be at +5 volts to keep it in the write mode.

We should first generate that 25-volt programming pulse. Remember, when the control program requests a program pulse, pin P on the module goes to ground. By connecting P to C, that ground signal will cause 25 volts to appear at pin 25V. To keep EPROM pin 20 at +5 volts between pulses, a diode connects it (pin 20) to +5 volts. Another diode is used to isolate the 25-volt switch. As it is right now, when the program commands a program pulse, 25 volts appears at 25V, goes through D2, and appears at pin 20. When there is no program pulse, 5 volts is applied through D1 and appears on the same pin (see Fig. 6b).

"The only problem remaining is how to interconnect all this stuff."

To read the chip, we need some way to pull pin 20 to ground. The verify command causes pin V to go to ground. So we connect V to E20, but if V goes to ground it shorts the 5-volt power supply through D1. To avoid such complications as extra switching transistors, relays, or other gimmicks, I'll just put a 1k ohm resistor in series with D1 to limit the short-circuit current. And there you have it (Fig. 6c). Because the diode is in series with the programming voltage, adjust the 25-volt supply for 25 volts at E20.

The Control Program

Add a new data statement af-

ter line 250: DATA"2764",8191,4, 25. Increment the number in line 210. We have just told the program that type 2764 has been added and that it is 8,191 bytes long (this is one less than the actual number of bytes—the computer starts from zero). The last two numbers indicate that the system should try to program each location a maximum of 25 times using 2-millisecond pulses (four times 500 microseconds). Incrementing the number at line 210 tells the system that there is one more device added to its repertoire.

Other EPROMs are more difficult to adapt to than the 2764. By comparing the data in Fig. 5 with the module schematics in Fig. 3, you should be able to see methods for producing most kinds of control signals.

An especially tricky device was the 2732. The only difference between programming and reading is the 25 volts applied to pin 20 during a program cycle. To switch from program to read without destroying the 2732, the 25 volts had to be removed before the chip enable pin was brought low. Because of this, the programmer box was modified to always shut off the 25 volts just before the V pin goes low. If the control program requests a read (verify true and pin V at ground), the 25-volt programming supply is always shut off. While this is a requirement with the 2732, it makes for safer operation with other types.

Before using the system on a real live EPROM, test the signals at the EPROM socket pins with the programming module in place. Use an oscilloscope to verify that all signals are within specification and that the control program works properly. This tests the new programming module and verifies that the entire system works as well.

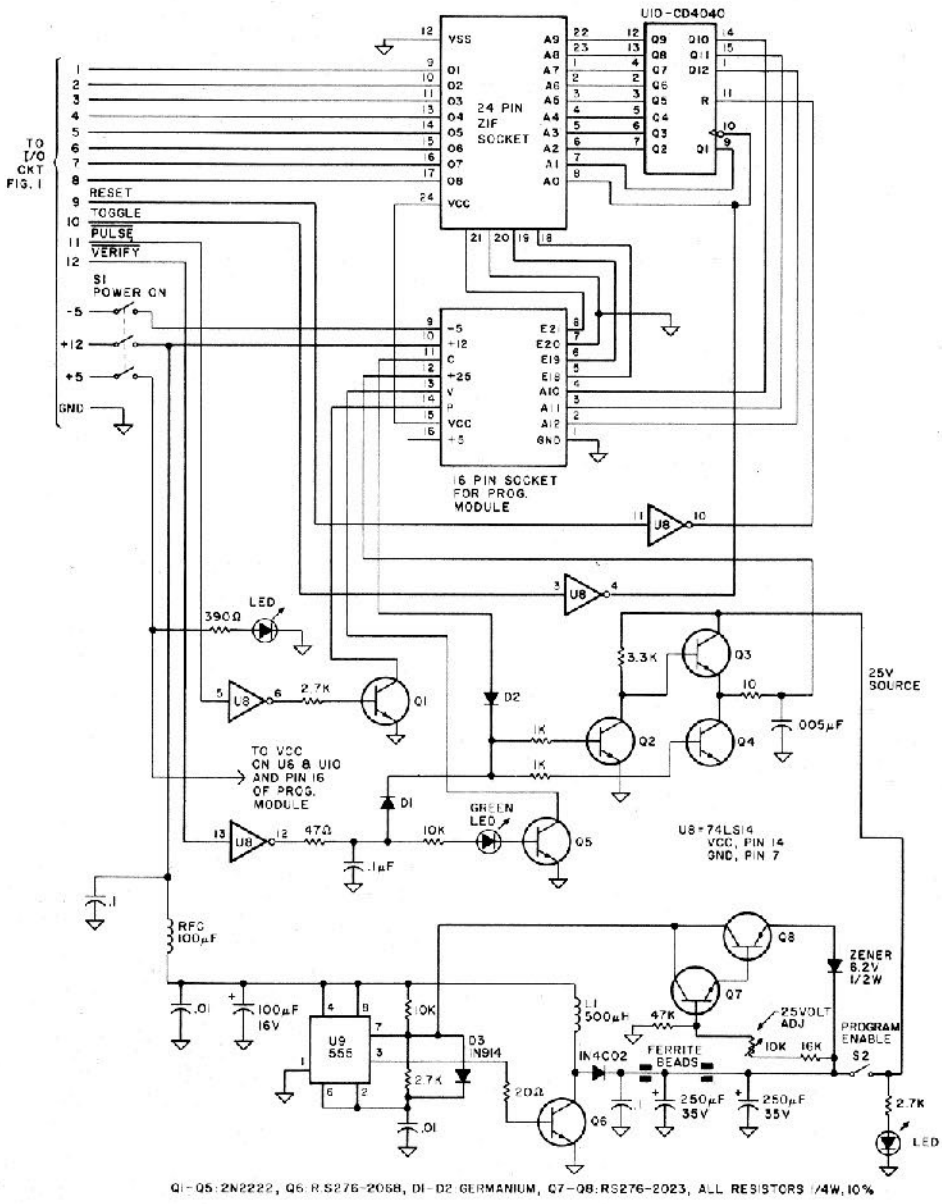


Fig. 7. Programming Box

Now What Do I Do With It?

If you've made it this far you probably have some pretty good ideas about uses for EPROMs. One of the most popular notions is to connect an EPROM containing a machine-language monitor or printer driver at the unused address space just above the Level II ROMs. Anoth-

er use is to provide the program for dedicated microprocessor controllers (burglar alarms, model railroad controllers, printers, solar heat control, electronic mail box, packet radio controllers). If you are tired of the character set on the screen of your computer, you could make a new character-generator ROM from a 2708 or 2716. And have you ever thought about modifying Basic in ROM? The old three-chip set for the Model I used 32K ROMs that are pin-compatible with 2532 EPROMs. Armed with enough information and stamina you might be able to convert your faithful Model I to a Model III (or better).

No matter what your EPROM needs are, OMNIPROM fits the bill. If you have any questions or comments I would be glad to hear from you. Please include a stamped self-addressed envelope with your letter. ■