Popular Electronics
SEPTEMBER 1981

BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 1: The Basic System

With the widely used 8080 as a model, the basic features of a central processing system are explored

IN ADDITION to its obvious application as the central processing unit (CPU) of a computer system, the microprocessor has found its way into a variety of products ranging from kitchen equipment to sophisticated laboratory data-acquisition systems. The key to this widespread utility is flexibility, which in turn comes from the microprocessor's unique ability to alter its internal logic in response to an external program. Since the response to inputs from the program is extremely rapid—on the order of a few microseconds—the processor can change its electrical configuration practically instantaneously, usually fast enough to convince a human correspondent that it is performing several activities simultaneously.

Given the speed and flexibility of microprocessors, and the fact that they are available at very reasonable prices, it is often economical to use a single processor rather than a great many simpler chips to synthesize logic functions, act as a controller, or the like. To accomplish this, however, it is necessary to understand the architecture of the processor, its needs in terms of support circuitry, how to program it, and how to interface it with the "outside world." Development of the necessary understanding is the goal of this multipart series.

Microprocessors vary in design, with each design programmable only via its own set of instructions. The unit that will be covered in detail in this series is the 8080. Since this CPU is the grandfather of a growing family of processors, including the Z80, 8048, and 8085, all with a common internal programming language, most of the information will apply to the entire family as well. Instructions not used by the 8080 will not, however, be covered.

The Basic System. Like many processors and logic elements, the 8080 requires a small number of support ICs in order to function. An 8080 along with its support chips is called a *CPU module.*

The program that determines the internal states taken on by the CPU is supplied to it in the form of electrical signals. To generate these signals as required and in the proper order, the program must be stored in some form of "memory" device. These devices represent the binary digits (1, 0) by means of "on-off" switching devices or analogous circuit elements. The binary code in which program instructions are expressed is called *machine language.* Each microprocessor (or microprocessor family) has its own machine language.

Binary instructions or data that are not subject to change can be stored permanently in ROM (read-only memory). Elements that are variable must be stored in RAM (random-access memory), which can be written, erased, and rewritten by the CPU.

To affect or control devices that interact with the outside world, the processor must deliver signals to them. It does this by means of an I/O (input/output) port. As the name implies, an I/O port can also deliver signals to the CPU from devices that sense external parameters.

Electrical signals representing data, instructions, and addresses (the locations of particular items in memory) pass between the CPU, memory devices, and I/O ports via a set of dedicated lines known collectively as *buses.* A typical bus (Fig. 1) also supplies dc operating power to the elements of the system.

Bus System. There are many versions of the bus system currently used, with the S-100 and SS50 being two of the most common. Although different mechanically, they all contain three major elements: the address bus, the data bus, and the control bus. (Figure 1 does not show the power supply lines and common ground usually carried on the bus system.)

In most systems, there are 16 lines in the address bus, thus enabling 2^{16} or 65,536 (64K) unique addresses. In an 8-bit system, there are 8 lines on the data

bus, allowing 2^8 or 256 data combinations. The control bus carries all system synchronization signals including the "clock" that keeps all CPU module events in step.

Memory. A computer memory is formed from a large array of semiconductor elements, each capable of storing a single binary 1 or 0, organized into groups of *bits* (short for "binary digits") often called *words*. The number of bits in each word is determined by the size of the CPU *registers* (storage locations internal to the microprocessor) and the number of data lines. A typical RAM arrangement is shown in Fig. 2. A memory word of eight bits is often referred to as a *byte*. Each byte represents one of 2^8 or 256 unique values (0-255). As the 8080 microprocessor uses this memory structure, it is considered a byte-oriented device.

Each memory location contains one word of memory bits, and is identified by a unique number, or *address*, assigned to it. The CPU gains access to the contents of any memory location by means of its address. A memory word may represent the encoded form of an instruction, or may be data to be processed by the CPU.

The CPU has control of memory in the sense that it can read data and instructions from memory and write data back into memory. Only when the CPU receives a direct memory access (DMA) signal via the control bus does it relinquish control of memory. DMA allows a high-speed device such as a magnetic disk to gain access to memory and control it. As noted earlier, memory that can be read and written or altered is termed read/write memory, or random-access memory (RAM). Memory that can be read, but not altered by writing, is termed read-only memory (ROM).

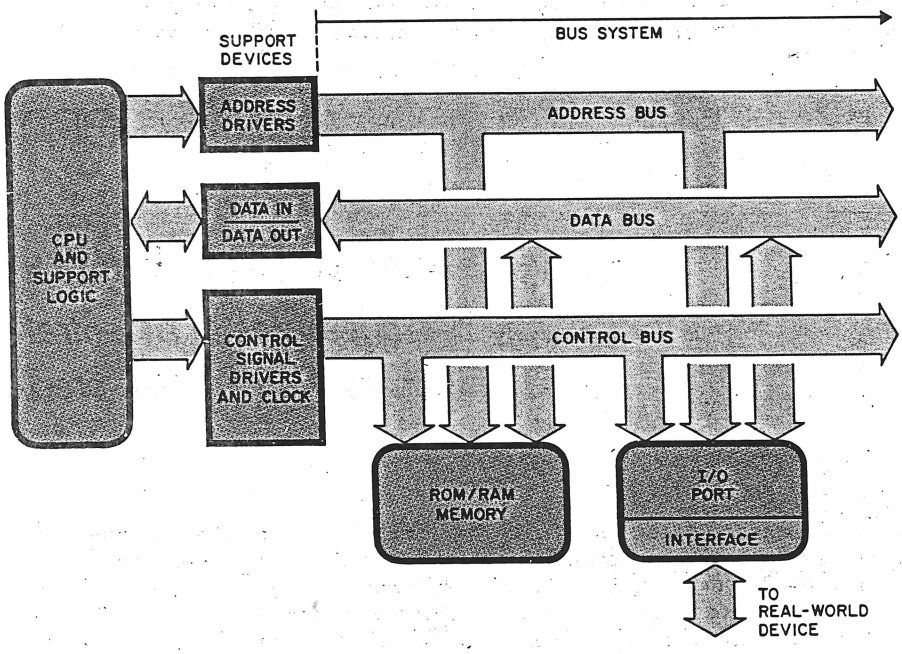


Fig. 1. A typical bus system contains three major elements: address bus, data bus, and control bus.

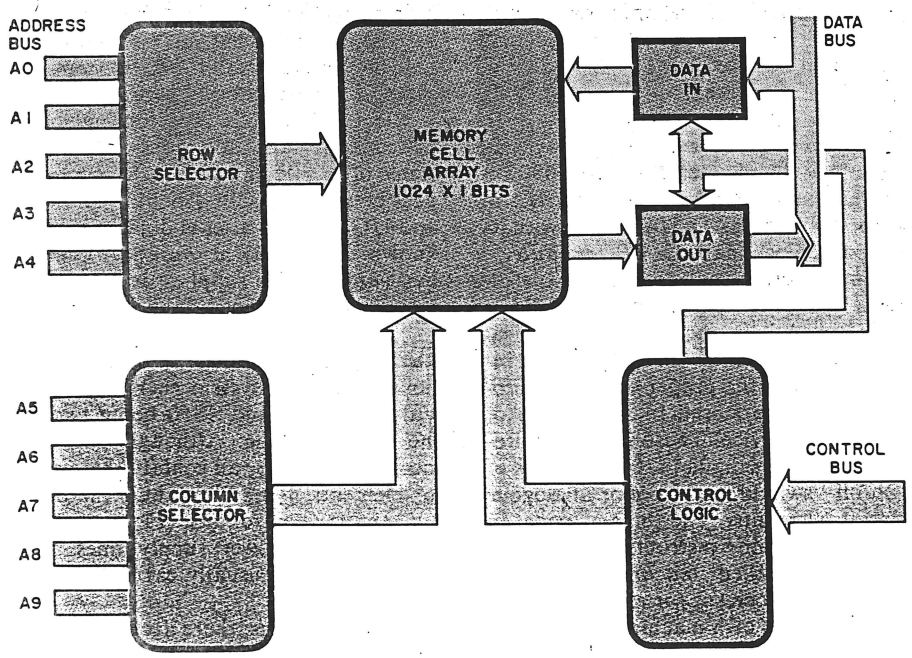


Fig. 2. Arrangement of a 2102 random-access memory. Eight of these are needed for 1024 by 8 bits.

Input/Output. To the 8080, the outside world may consist of up to 256 input and 256 output devices. These are usually referred to as peripherals, and may include keyboards, printers, displays, etc. Each peripheral communicates with the CPU by exchange of data bytes sent via its associated I/O port and the data bus (Fig. 3). Each peripheral is assigned an addresss from 0 to 255, much as each memory location is assigned an address. The portion of the I/O system that actually conditions data for input and output is known as the *interface* and generally there is one interface for each peripheral. The use of a port for input or output is done under program control.

Communication between the computer and a peripheral is done in one of two formats—serial or parallel. In parallel data transfer, all eight bits of the data byte are handled simultaneously. This permits rapid movement of data. In serial transfer, data is handled bit-by-bit instead of a byte at a time. This is slower, but has the advantage of using simple hardware (for example, a two-conductor cable or a telephone circuit instead of a multiconductor bus). When two computers exchange data via, say, an intercom line, the parallel data from buses of both computers is converted to serial form and transmitted bit-by-bit down the cable. The IC that performs the conversion from parallel to bit-serial form (and vice versa) belongs to a family of components known as UARTs (Universal Asynchronous Receiver-Transmitter). If used, the UART is part of the computer's I/O interface since it is used for conditioning data for input and output.

There are two basic types of serial communication—RS232 and what is called the 20-mA current loop. Basically, RS232 is a voltage circuit where a logic 0 is a positive voltage, and a logic 1 is a negative voltage. The newest version

of this voltage interface is RS422—which uses balanced transmission lines and differential current sensing to eliminate noise. The other commonly used serial port is the 20-mA current loop in which a flow of 20 mA in the series circuit produces a logic 1 while an absence of current denotes a logic 0. Both of these serial ports are controlled by a baud rate generator that "clocks" the operational speed of the port. Most peripherals use either the RS232 or 20-mA loops for communication.

Program Interrupt I/O improves the efficiency of CPU operation while data is being transferred to or from a peripheral that is many times slower than the CPU itself. Consider a computer processing large amounts of data, portions of which are to be output to a printer. When the peripheral is ready for data, it signals the CPU through a program interrupt. When the CPU acknowledges the interrupt, it completes the current instruction being executed in the main program and then automatically branches to a routine that will output the next data byte. After the byte is output to the printer, the CPU returns to where it left off in the main program. The 8080 is capable of handling up to eight interrupts from eight I/O devices using a special instruction of its instruction set. Data input is similarly handled.

Three-State Logic. There can be many peripherals connected to, and communicating along, the same bus lines. Thus, unless some form of "traffic control" is used, confusion can reign. Keeping order is the purpose of the three-state devices, shown in Fig. 4.

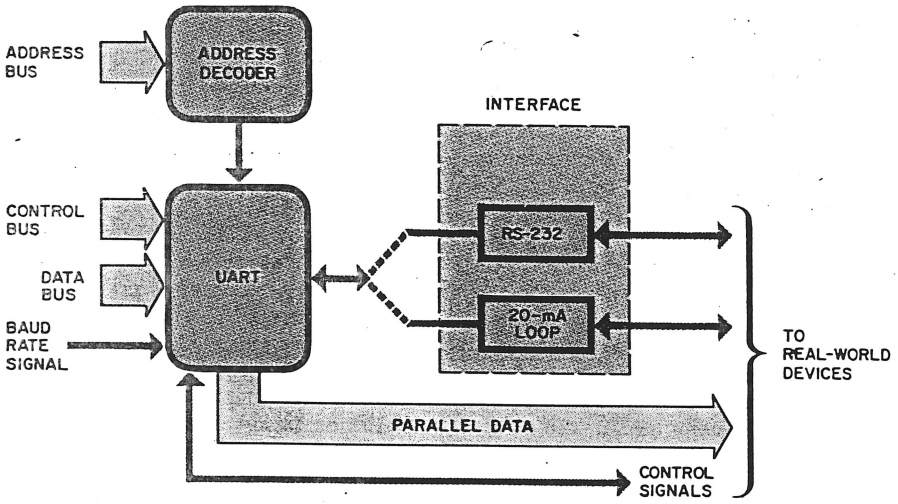


Fig. 3. Each peripheral communicates with the CPU through an associated I/O port and data bus.

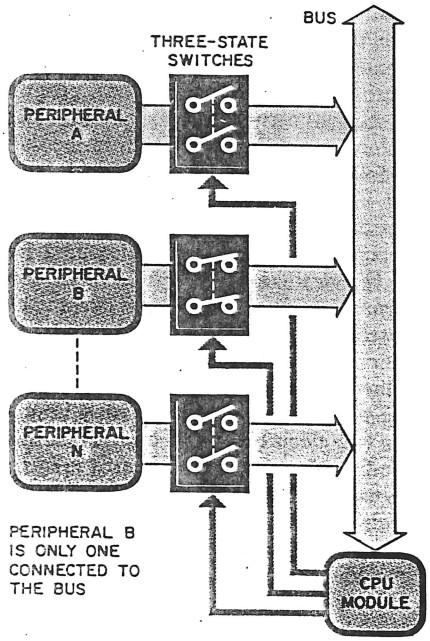


Fig. 4. A three-state device is an electronic switch connected between each bus line and associated logic.

Simply, a three-state device can be thought of as an electronic switch connected between each bus line and its associated logic. When the switch is closed, the associated logic can accept or deliver signals to the bus. But, when the switch is open, the bus does not "see" the logic—in effect, the logic does not exist for the bus.

Programming. A program for a computer or processor consists of a sequence of operational instructions stored in memory. Each instruction enables a single elementary operation such as the movement of a data byte, an arithmetic or logical operation on a data byte, or a change in instruction execution sequence. The set of all instructions common to a given CPU is referred to as its *instruction set*. The size of the instruction set is a measure of the CPU's capabilities. Another such measure is the length of the binary words the CPU can work with. Generally speaking, the larger the instruction set, or word size, the more powerful the CPU. The 8080 (an 8-bit CPU with 72 instructions) is thus more powerful than the 4040 (a 4-bit CPU with 60 instructions). Some microprocessor instruction sets may approach 200 instructions in length.

A program is stored in memory (RAM or ROM) as a sequence of bytes that represent the instructions. The memory address of the next instruction to be executed is held in an internal register of the CPU called the *Program Counter*. Early in the execution phase of each instruction, the program counter is automatically advanced to the address of the next sequential instruction in memory. Thus, program execution proceeds sequentially (i.e. memory location 213 is executed after location 212 is executed, etc.) unless a transfer-of-control, or BRANCH instruction (8080 JUMP, CALL, or RETURN) is executed, which causes the program counter to be set to a specified memory address. Program execution would then continue sequentially from this new memory location. The JUMP instruction specifies the address to be jumped to, which can be anywhere in memory. During execution of a JUMP, the CPU replaces the contents of the Program Counter with the address contained in the JUMP instruction.

Subroutines. A special type of jump occurs when the stored program CALLS, or accesses, a subroutine (a program within a program). Usually, a subroutine is a set of instructions that must be executed repeatedly in the course of running the main program. Algorithms that calculate mathematical functions and routines to input or output data to a peripheral device are often programmed as subroutines. The subroutine type of

jump requires the CPU to store the contents of the program counter at the time the jump occurs (when the CALL instruction is executed). This enables the processor to resume execution of the main program after the last instruction of the subroutine has been executed.

The processor has a special method of handling subroutines to insure an orderly return to the main program. When the CPU receives a CALL instruction from memory, it advances the Program Counter to the address of the next sequential instruction, and saves the Counter's contents in a special memory area known as the *Stack*. The latter holds the memory address of the instruction to be executed after the subroutine is completed. The processor then loads the address specified in the CALL instruction into its Program Counter. Consequently, the next instruction that is to be executed will be the first step of the subroutine.

Normally the last step of any subroutine is a RETURN instruction. When the processor executes the RETURN instruction, it replaces the current contents of the Program Counter with the address contained on the "top" (last entry) of the Stack. Since this address was the one originally saved by the CALL instruction, the processor will resume execution of the calling (main) program at the point immediately following the original CALL instruction. Note that this operation is very similar to executing a JUMP instruction, the difference being that the JUMP address is contained in the Stack area rather than in the JUMP instruction itself.

A subroutine may CALL another subroutine. This is called "nesting subroutines." If the microprocessor being used has a Stack for storing RETURN addresses, the maximum depth of nesting subroutines is determined solely by the depth of the Stack itself. So if the Stack has space for saving five return addresses, then five levels of subroutines can be accommodated.

Microprocessors have different methods of maintaining their Stack. Some store the RETURN addresses within registers in the processor, but this limits the levels of subroutine nesting. Others, such as the 8080, use a reserved area of RAM for the Stack and maintain a *Stack Pointer* (an internal register of the CPU) which contains the address of the most recent Stack entry; *i.e.*, the Stack Pointer always "points" to the top of the Stack. This type of Stack may be looked upon as a last-in-first-out (LIFO) memory, and allows virtually unlimited subroutine nesting.

Flags. The CPU has a set of flags, or internal flip-flops that are set or cleared (i.e., set to a logic 1 or 0, respectively) depending upon the results of certain instructions as they are executed. Two flags of the 8080 are: The "Zero Flag," which is set if the accumulator is 0 (actually 00000000 binary), and the "Carry Flag," which may be set when an arithmetic instruction causes the accumulator to overflow (*i.e.*, carry or borrow from an addition or subtraction). In most microprocessors there are other flags besides these. The 8080 has a total of five.

Most processors have instructions available that will store the accumulator and other general-purpose registers and flags on the Stack temporarily. Likewise, there are instructions available to reload the general-purpose registers and flags with data contained on the top of the Stack. This allows the contents of the registers and flags to be saved so that they may be used in another activity, as for example, a subroutine. Just before returning to the main program from a subroutine, the subroutine will restore the registers and flags it used (assuming, of course, that the same registers and flags were saved on the Stack prior to using them in the subroutine).

Let's go over one last concept of a CPU's instruction set, which gives the computer its "decision-making" power. This is a special set of transfer-of-control instructions that transfer program execution to another portion of memory if the condition specified in the instruction is met. An example is the 8080 instruction JUMP-IF-ZERO.

If the processor encounters a conditional transfer-of-control (or "conditional branch") instruction, it checks to see if the specified condition is met. The "condition" is always related to one of the flags. In the case of JUMP-IF-ZERO, program execution is transferred to the JUMP address contained in the instruction in the same manner as the unconditional JUMP *if* the Zero Flag is set. If the Zero Flag is not set (cleared), program execution assumes its sequential flow and executes the instruction immediately following the JUMP-IF-ZERO. A processor usually has a set of "Compare" instructions, that set and/or clear flags depending upon the result of comparison of two data words (the 8080 can compare two registers, or a register and the contents of a memory location). A conditional branch instruction will often follow a Compare instruction, so that the proper execution path may be chosen (the decision) based on the results of the flags from the Compare. It is in this manner that the CPU makes its "logical decisions." The 8080 also has various conditional calls and conditional returns in addition to the conditional JUMPS in its instruction set. ◇

(To be continued next month)

BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 2: The CPU Module

A practical system and how to connect it to the outside world

IN Part 1 of this series, we discussed the basic features of a central processing system, using the 8080 as an example. Included were descriptions of how such features as the memory, input/output devices, and programming work. Now we will examine how to design a CPU module based on the 8080. The schematic of such a module is shown in Figs. 5 through 7.

In the design of this module, one of the objectives was to keep it as simple as possible while retaining versatility in interfacing and expansion. The module incorporates 1K bytes (1024) of RAM and 2K bytes of EPROM (erasable programmable read only memory) which should be ample memory for most control applications.

Most of the signals found in the CPU module are available at the Bus Interface of Fig. 7. The others, denoted by an asterisk, are for interfacing the CPU module to a Program Development board that is to be presented in Part 3 of the series. These signals will otherwise normally be of no concern and should be left open-circuited.

Circuit Description. The 8080 microprocessor, (*IC1* of Fig. 5) initiates and directs all operations between itself, the memory, and the I/O units. Crystal-controlled clock generator *IC3* provides two nonoverlapping clock phases ($\phi 1$ and $\phi 2$) derived from the 18-MHz crystal. The clock also generates a status strobe, $\overline{\text{STSTB}}$, at pin 7 for use in *IC2* to provide the control bus signals. Other functions of *IC3* include providing a synchronized RESET signal (pin 1) to *IC1* in response to an external asynchronous $\overline{\text{RESIN}}$ signal (pin 2) and a synchronized READY signal (pin 4) in response to an external RDYIN signal (pin 3). The network consisting of *R1* and *C1* provides a power-on-reset to *IC1* through *IC3* when the module is powered up. Program execution begins immediately at memory location zero after power-up (unless the RDYIN input is low, in which case the CPU remains idle after reset until it is brought high). The RUN status of the CPU is indicated by *LED1*. Besides generating the control bus signals, *IC2* buffers the bidirectional data bus. The need for a separate negative power supply is obviated by *IC4*, which generates -5 V from the $+5$-V supply.

The microprocessor operating program is stored in EPROM *IC5* of Fig. 6. Pin 8 of *IC10A* is low for all addresses between hexadecimal 0000 and 07FF, which "turns on" *IC5*. This corresponds to 2048 unique memory locations, which is exactly the number of bytes of memory in *IC5*. The eight outputs (constituting one byte) of *IC5* are logically connected to the data bus when the output enable, $\overline{\text{OE}}$, on pin 20 is driven low by the control bus signal $\overline{\text{MEMR}}$ from pin 24 of *IC2*. When asserted, this signal is the CPU's way of notifying the system that it is ready to accept a byte of information from memory. Inputs A0 through A10 of *IC5* determine which of the 2048 internal bytes will be presented at its outputs (when enabled).

System RAM is formed by *IC7* and *IC8* (Fig. 6) and its operation is similar to that of EPROM *IC5*. The RAM does not normally contain the CPU's program since, unlike an EPROM, it is volatile in nature. That is, the RAM powers up into a random logic state, which is of no value to the CPU. However, the RAM may be used as a temporary data "scratchpad" since CPU data may be readily stored in it and retrieved later. The Stack area for the CPU will exist somewhere in the RAM.

Pin 11 of *IC10C* is low for all memory read and write operations between addresses 0800 and 0BFF (1024 unique locations), which "turns on" the RAM, containing 1024 bytes of memory. The difference in operation between the EPROM and the RAM is in the write-enable, $\overline{\text{WE}}$, input at pins 10 of *IC7* and

IC8. The state of this input determines the mode of operation of the RAM (read or write) when it is being accessed by the CPU (that is, when pin 11 of *IC10C* is low). When the write-enable input is high, the I/O lines of *IC7* and *IC8* are in the output mode and operation is similar to that of the EPROM. When low, the I/O lines are in the input mode and data on the data bus is stored in the addressed memory location. Note that the control bus signal $\overline{\text{MEMW}}$ at pin 26 of *IC2* drives the write-enable input of *IC7* and *IC8*. (The assertion of $\overline{\text{MEMW}}$ tells the memory that the CPU is attempting to write data into it, from the data bus). Inputs A0 through A9 determine which of the 1024 internal memory bytes will be read from or written into. The high-order bits of the address bus, which control the selection of *IC5*, *IC7*, and *IC8*, are decoded by *IC9* and *IC10*.

Ins and Outs of the CPU Module. Now that we have the basic CPU module, how do we enable it to communicate with the outside world? Suppose we want to monitor temperatures from sensors installed in various rooms of a house. How would we go about connecting the temperature sensors to the CPU? Or, suppose we want an alarm to sound if a forced entry is detected in the house. How is the alarm told to sound when the system detects an intruder? These are examples of the type of problem we'll be investigating—how to interface a digital computer to an analog world. We will approach it in a generalized manner so that a neophyte can design interfaces for his applications.

Once we learn how to interface external devices to the CPU module and how to program the module, applications will be limited only by the experimenter's imagination. For instance, once we have temperature sensors interfaced to the module it is a simple matter to program it to detect if the temperature is rising or falling (and how fast), to sound an alarm (or take other appropriate action) if a temperature limit has been exceeded, to record maximum and minimum temperatures with their corresponding dates and times, etc. The CPU module could easily handle this task and at the same time act as watch dog over the premises. Want to play a game with the system or have it wake you up in the morning while it's finishing brewing a fresh pot of hot coffee? It's simply a matter of connecting the appropriate peripherals (coffee pot and alarm) and their interfaces to the CPU module and plugging an EPROM with an appropriate program into the module.

To complete the hardware, let's look at how we would go about designing a parallel output interface. In the following discussion, remember $\overline{\text{I/O W}}$ means that the CPU is "outputting" a data byte. However, this data byte is present on the data bus for only about one microsecond, too short a time for humans to even notice. One could bring the RDYIN line low during the output instruction's execution, which would prolong the time the output data byte was available. Since the CPU is stalled as long as RDYIN is held low, this would tend to make the CPU very inefficient. A better method would be to somehow "snatch" the byte from the data bus and store it externally for as long as we please, while allowing the CPU to hum along at full speed. Figure 8 shows how this can be implemented.

Since the 8080 is capable of handling 256 output ports, the interface must have some means of determining if it is the one to receive the data byte. The Output Port Select in Fig. 8 accomplishes this by giving a true output for one unique address out of the 256 possible I/O port addresses. This circuit may consist of an 8-input NAND gate, an 8-bit comparator, or a decoder (1-of-8 or 1-of-16) chip as shown in Fig. 9. The selection device used is connected to

PARTS LIST

C1,C2,C3—10-μF, 10-V tantalum capacitor
C4,C5—2.2-μF, 15-V tantalum capacitor
D1—Germanium diode (1N270 or similar)
IC1—8080A microprocessor
IC2—8228 system controller
IC3—8224 clock generator and driver
IC4—ICL7660 voltage inverter
IC5—2716 EPROM
IC6—74LS368 hex inverting tri-state bus driver
IC7,IC8—2114L 1024x4 RAM
IC9—74LS33 quad 2-input NOR buffer
IC10—74LS00 quad 2-input NAND
IC11,IC12—74LS244 noninverting tri-state buffer
LED1—Red light emitting diode
P1,P2,P3—16-pin DIP socket
Q1—2N2907 or 2N3906 transistor
R1—10-kΩ, ¼-W 10% resistor
R2—330-Ω, ¼-W, 10% resistor
R3—20-kΩ, ¼-W, 10% resistor
R4,R5,R6,R11—3.3-Ω, ¼-W, 10% resistor
R7—1-kΩ, ¼-W, 10% resistor
R8,R9,R10—39-kΩ, ¼-W, 10% resistor
XTAL—18.000-MHz quartz crystal (Cystek CY19A or similar
Misc.—Sockets for ICs (must be provided for IC5), perf or pc board, 0.01-μF disc ceramic bypass capacitors distributed near ICs, +5-V, 500-mA and 12-V, 60-mA power supplies, wire-wrap wire or solder, etc.

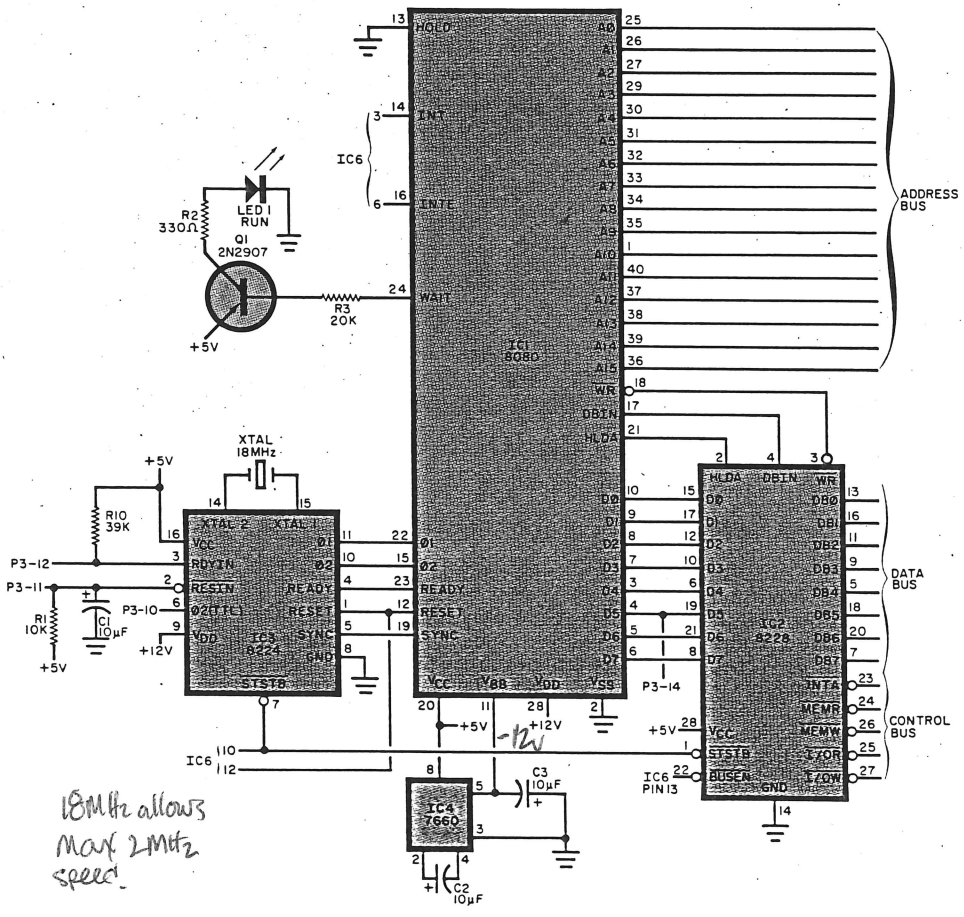


Fig. 5. Schematic of the microprocessor, clock generator (IC3) and control signal generator (IC2).

18MHz allows max 2MHz speed.

6.75MHz = 750KHz CPU FRQ

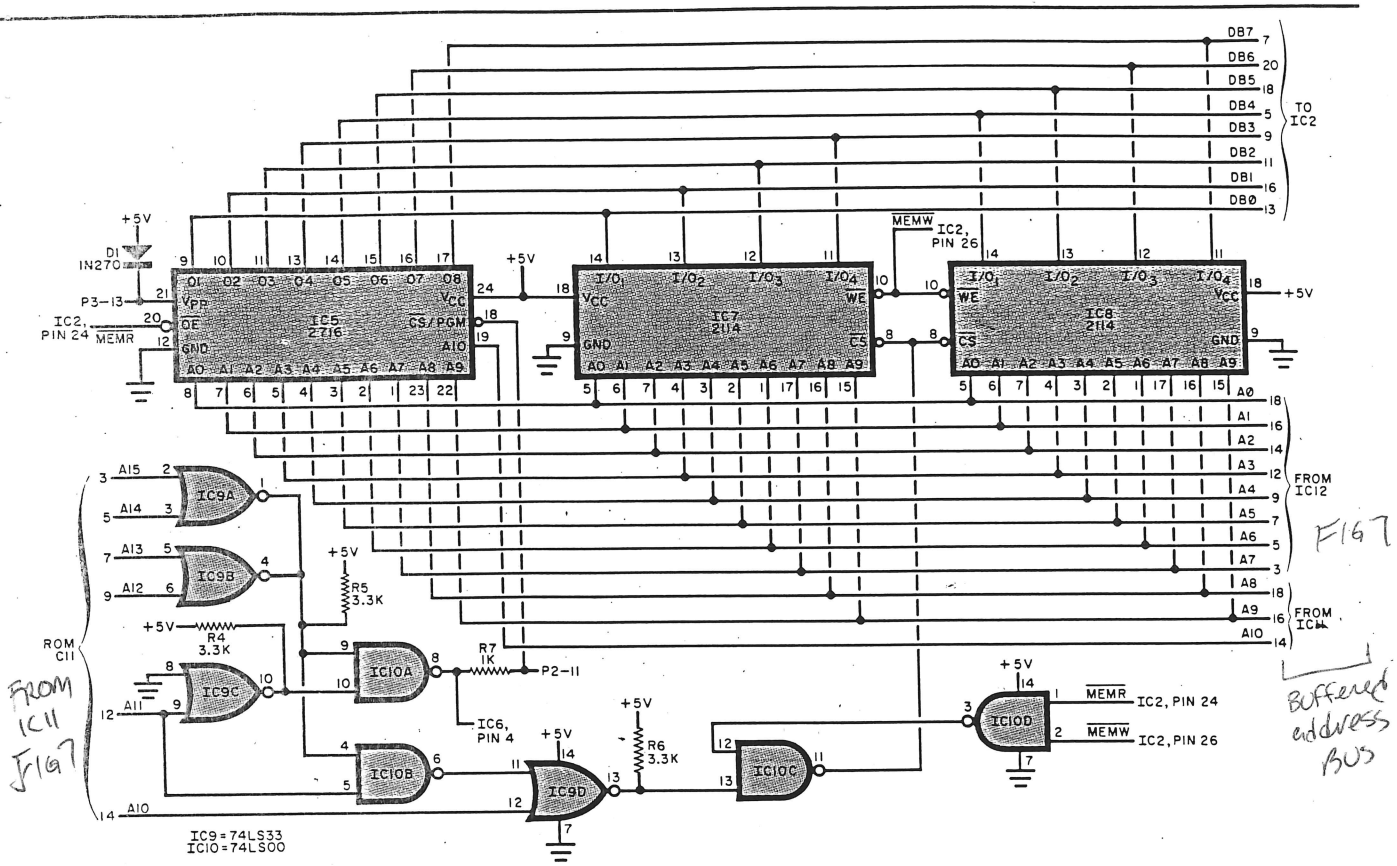


Fig. 6. Memory circuits contain the EPROM (IC5) and RAM composed of IC7 and IC8. Control logic is in IC9 and IC10.

either the high- or low-order byte of the address bus (both of which carry the I/O port address). We will use the high-order byte in the examples.

In Fig. 9A, the NAND gate approach, inverters can be used to create the desired port address. Here the port address is E8. The 1-of-8 decoder approach is shown in Fig. 8B. This method is particularly attractive when more than one output port is needed. A 1-of-16 decoder can be used when working with more address lines. The comparator approach, Fig. 8C, uses exclusive-NOR gates whose output goes high only when the same logic signal is applied to both inputs. By using open-collector gates as shown here, the outputs may be hard-wired together (wire ANDed) so as to produce a high output only when all the gate outputs are high. Using jumpers, port addresses are easily changed.

We now know how to determine *who* the CPU is communicating with, but now how do we actually "store" the output data byte? It just so happens (by no coincidence) that $\overline{I/O\ W}$ goes true (low) shortly after the output data has had time to stabilize on the data bus, and goes false (high) just before the data byte disappears. This translates to a low-going pulse on the order of half a microsecond in length, which is suitable for most digital IC's. By using this pulse to clock a latch (a temporary storage register), we will have succeeded in snatching and storing this data byte.

The AND gate in Fig. 8 tells the output latch to latch the contents of the data bus (which contains the data byte) at the proper time *only* when the CPU is making reference (outputting) to that particular latch (output port number). The eight outputs of the latch hold the data byte, which may be used for driving LED's, a printer, or turning on the coffee pot. One of the outputs may be connected to a relay or SCR to turn on the coffee pot, another output may drive an alarm, while yet another may turn on an air conditioner (via a relay, or SCR of course). It is evident from these examples that one output port can control a variety of peripherals by selectively setting and clearing the appropriate control bits at the latch output. This is easily done in the computer's program, which will be discussed in Part 3.

A parallel input interface is almost identical to a parallel output interface. The only difference is the direction of flow on the data bus. During the execution of an "input" instruction a "window" of only about half a microsecond exists in which input data can be placed on the data bus. This cannot be done at any other time or conflict may occur, resulting in a system "crash."

It is therefore essential that the input data be gated onto the data bus at the proper time. Fortunately, this strict timing requirement can be easily satisfied by use of the CPU generated $\overline{I/O\ R}$ signal. As the CPU executes an input instruction, it generates $\overline{I/O\ R}$ to inform external logic that input data can be placed on the data bus. This signal is usually AND'ed with an "Input Port Select" signal which is then connected to the enable input of three-state buffers as shown in Fig. 10. Note the similarity to the parallel output interface (Fig. 8). During the final execution phase of an input instruction (when $\overline{I/O\ R}$ is active), the input data is "latched" inside the CPU (transferred to the accumulator); therefore an external latch is not required as in the output interface.

In the I/O port decoder examples of Fig. 9 , the address bus (A8-A15) in itself does not tell us whether we are referencing a memory location, an input port, or an output port. Consequently , the Port Select signal will be true whenever the high-order byte of the address bus contains E8 (E8 through EF in Fig. 9B), regardless of the type of reference being made. This "ambiguity" may be put to advantage because it then makes it possible to use an Output Port Select signal also as an Input Port Select signal. In other words, the Port Selects for

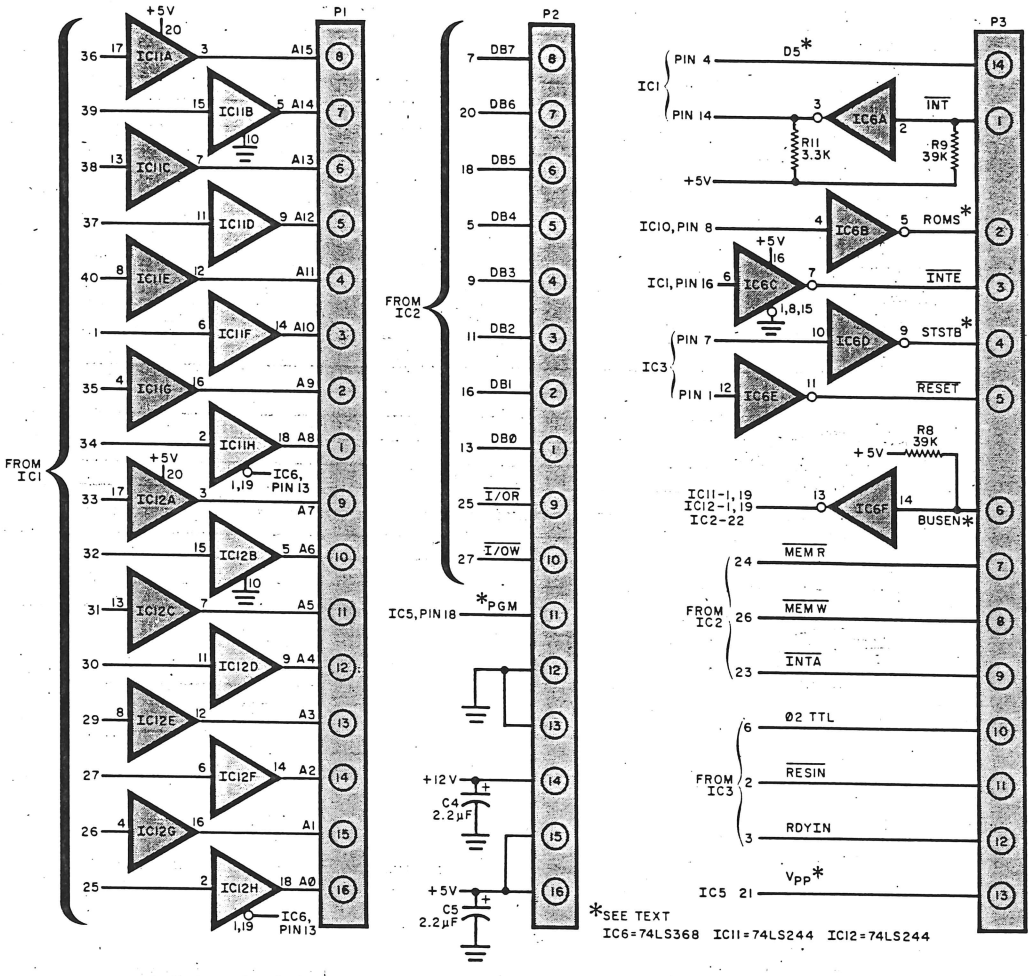


Fig. 7. Bus interface for the CPU module shows connections to the outside world. Signals marked with an asterisk are for interfacing the CPU module to a Program Development-Debugging board to be described in Part 3.

Fig. 8 and Fig. 10 may share the same Port Select circuit. (The control bus resolves this ambiguity by specifying the type of reference the address bus is making.) If the input and output port numbers are not equal, then two separate Port Select circuits will be required. The conrol bus signals $\overline{I/O\,R}$ and $\overline{I/O\,W}$ differentiate the input and output operations, as may be observed by comparing Figs. 8 and 10.

Figure 11A shows an output latch. The CPU data bus is connected to an octal latch which is clocked by the coincidence of Port Select and I/O Write signals. The latch outputs can be used to drive relays, LEDs, a printer, D/A converter, etc. The latch is cleared when the CPU is reset. In the typical parallel input interface circuit shown in Fig. 11B, data is buffered via the three-state

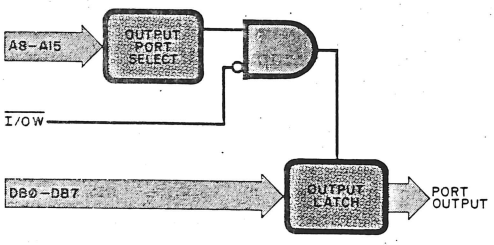
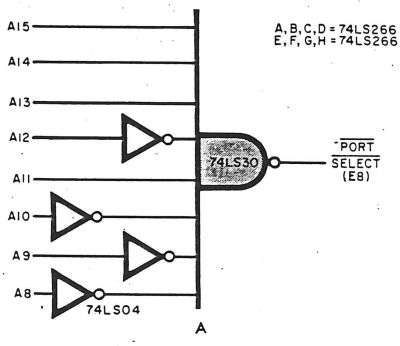
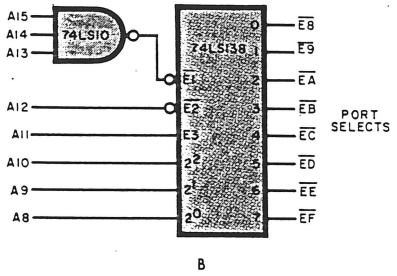


Fig. 8. Parallel output interface block diagram.





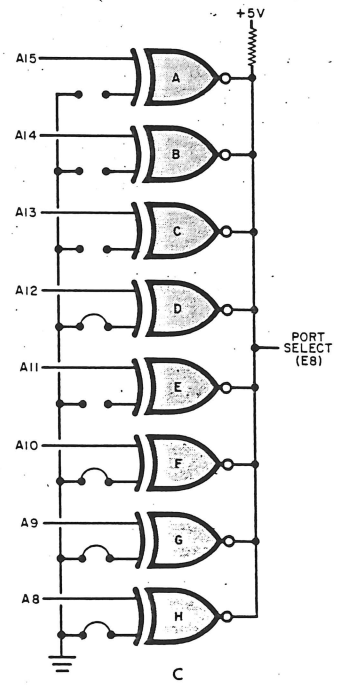


Fig. 9. Three ways to generate the port select signal: (A) with a NAND gate; (B) with a 1-of-8 decoder; and (C) with a comparator.

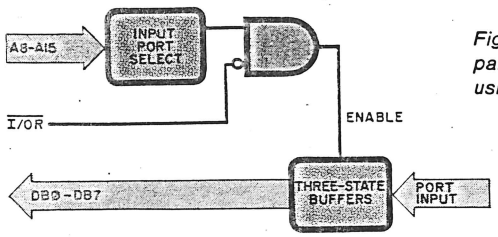


Fig. 10. Block diagram of parallel input interface using three-state buffers.

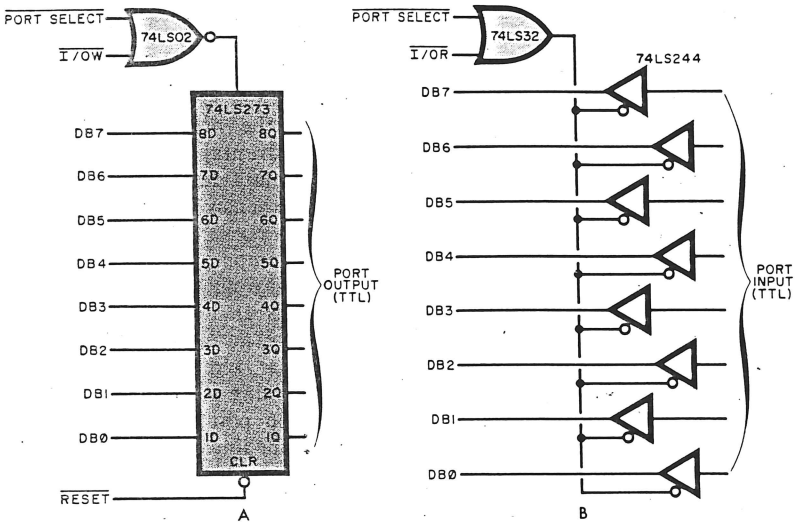


Fig. 11. At (A) is an output latch: (B) is a parallel input interface circuit.

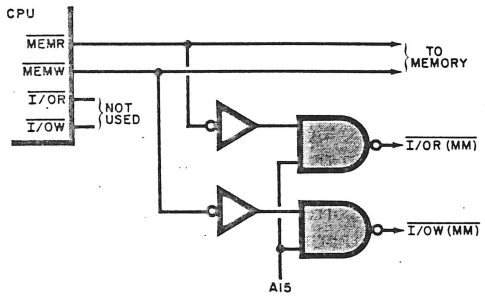
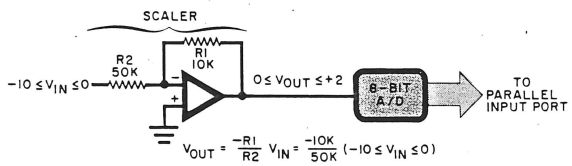


Fig. 12. Circuit for implementing memory-mapped input/output.

Fig. 13. Scaling circuit for example used in text.



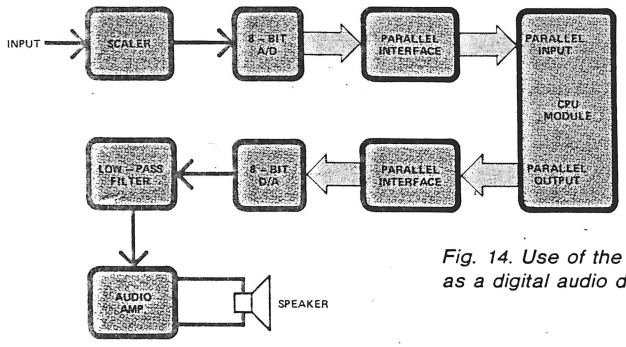


Fig. 14. Use of the CPU module as a digital audio delay line.

device to allow the data to be gated onto the data bus at the proper time. The Port Select signal can be derived from any of the previously discussed Port Select circuits. The input and output interfaces can share the same Port Select circuit if their port numbers are equal.

Note the similarity between $\overline{\text{MEMR}}$ and $\overline{\text{I/O R}}$ and also $\overline{\text{MEMW}}$ and $\overline{\text{I/O W}}$. In fact, the only reason the CPU generates $\overline{\text{I/O R}}$ and $\overline{\text{I/O W}}$ for input and output is to islolate memory from the I/O ports (by using the 8080 input and output instructions). Since the I/O structure may be viewed as an array of 256 single-byte memory locations (and therefore read and written), there is really no reason why $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ cannot also be used for I/O. An I/O of this type is called *memory-mapped I/O* (as compared to *isolated I/O* where the input and output instructions are exclusively used for input and output). If the full 8080 address space (64K bytes) is not used by memory, then memory-mapped I/O can be implementd.

Let's assume, for example, that we will never use any memory locations above hexadecimal address 7FFF. If we gate address bus bit A15 (which goes high for all address locations above 7FFF) with the $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ signals (Fig. 12), we may address up to 32,768 (2^{15}) input and 32,768 output devices! These new I/O control signals—$\overline{\text{I/O R (MM)}}$ (mm=memory mapped) and $\overline{\text{I/O W (MM)}}$—connect in exactly the same manner as the isolated control signals $\overline{\text{I/O R}}$ and $\overline{\text{I/O W}}$. The address bus now activates *memory* if A15 is a logic 0 and activates *I/O* if A15 is a logic 1. The I/O devices are still considered addressed ports, but instead of the accumulator being the only transfer medium, any of the 8080 registers can be used. All of the 8080 instructions that operate on memory locations can also be used in memory-mapped I/O. So by allocating an area of memory address space as I/O, we can create many new I/O "instructions" in the 8080 instruction set.

Some Applications. Note that data to be input in Fig. 11B must be in digital form. However, very few things in our world are digital in nature; they usually appear in analog form (voltages, currents, temperatures, sound waves, etc.). It is therefore inevitable that more circuitry will be required to complete the input interface. Before we discuss some typical examples, let's introduce the key element to be used—the analog-to-digital (A/D) converter.

The A/D converter is a versatile device widely used in computer applications. Its function is just what its name implies: to convert an analog (real-world) signal into digital form. A typical 8-bit A/D might accept an analog input voltage between 0 and +2 volts and represent this voltage by an 8-bit number at its output. In this case, an input voltage of +2 V would be represented by 255 (hexadecimal FF) at the output, 0 V by 0, +1 V by 127 (hexadecimal 7F), etc. The process of converting an analog signal to a digital number is called *quanti-*

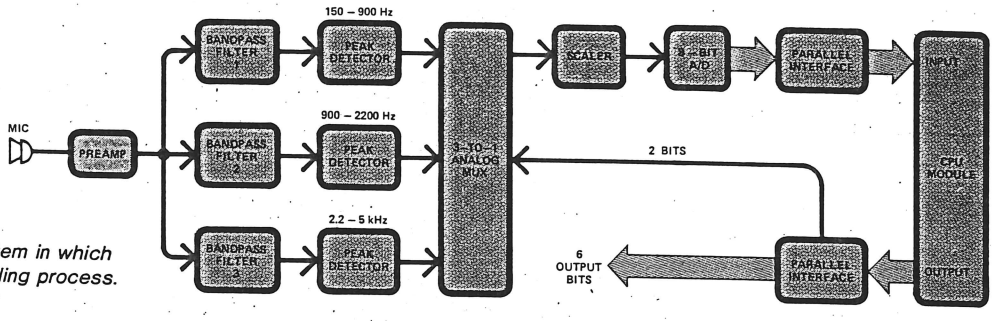


Fig. 15. A speech recognition system in which A/D conversion is used in a sampling process.

zation, and a variety of devices is available to perform this operation.

Since a typical A/D converter generally operates only over a small range of input voltages, what if we want to quantize a signal that varies from −10 V to 0 V, and the A/D can only convert voltages in the range of 0 to +2 volts? Figure 13 illustrates one possible solution. In this circuit, an input of −10 V will produce 255 (hex FF) at the A/D converter output. The process of conditioning an analog signal in order that it may be presented to an A/D in its operating range is called *scaling*. Note that if we built a variety of scaling circuits (to handle a wide range of input voltages) we would have the makings of a digital voltmeter. If we also converted currents and resistances into voltages within the range of the A/D, we might make our CPU function as a DMM, simply by connecting the A/D converter output to a parallel input port and writing a suitable program.

By connecting a digital-to-analog converter (D/A) to a parallel output port, we provide many more applications of the CPU module. For example, the module can be used as a digital audio delay line (Fig. 14) by "shifting" the quantized signal through the CPU's RAM. By varying the amount of delayed signal that is recombined with the original undelayed signal (either externally or in the CPU), and by varying the delay time, the CPU can create the effects of flanging, echo, phase shifting, compression (sustain), vibrato, harmonizing, etc. The delay time is easily controlled in the CPU's program by varying the rate at which the quantized music samples are shifted through the CPU's RAM. All of the signal characteristics—amplitude, frequency, and phase—can be easily manipulated once the quantized signal is in the CPU's memory. The real beauty of this approach is that all of the effects can be implemented with the same piece of hardware. Each special effect can be represented by a program routine in the CPU's EPROM memory, which is individually "called into action" via switches from an input port (or other means).

Another application of the A/D converter is in speech recognition. As shown in Fig. 15, bandpass filters are connected between a microphone and the A/D converter, a suitable speech-recognition program can be written to control various output devices (lights, locks, heaters, etc.) upon receipt of specific verbal commands. The peak detectors at the bandpass filter outputs have a sufficiently long time constant to act as "time-averagers." The dc voltage at the peak detector outputs are proportional to the amount of energy present in the speech waveform within the passband of the respective bandpass filters. By periodically sampling the peak detectors, the CPU can identify ("recognize") words and phrases in any language by way of comparison methods. The A/D converts the detector voltages into digital form for the CPU via an analog multiplexer. The output port of the CPU determines which peak detector is sampled. The six unused bits can be used to control external devices in response to verbal commands.

Let us look at one last way in which our CPU module can be put to use. Suppose we desire to build a digital thermometer using an A/D and the CPU module. How do we convert temperature to a suitable voltage? There are a wide variety of temperature transducers available, the price of which seems to be proportional to the precision desired. But by taking advantage of the CPU's ability to manipulate data, we may employ a very inexpensive device as the transducer.

A very basic temperature transducer circuit is shown in Fig. 16A. The transducing element is an inexpensive thermistor that is by no means the most accurate or linear temperature transducer. But, by taking a sufficient number of calibration points (the number depending upon the linearity of the thermistor used), a high degree of accuracy can be obtained. Figure 16B illustrates the ideal output voltage/temperature transfer curve, which is a straight line. A real physical thermistor however will produce a curve that may be very irregular in shape, instead of a straight line. If calibration points are taken at regular intervals along the thermistor's curve, that is, if output voltages are measured for various known temperatures, a "calibration correction table" can be created for the thermistor. Stored in the CPU's memory, this table can be used to measure other temperatures accurately by methods of approximation. As shown in Fig. 16C, consider point x between two calibration points a and b. The unknown temperature Tx may be approximated by $Tx = Ta + \Delta T$ where $\Delta T \approx m\Delta V$, with m being the slope of the line intersecting points a and b. Then $Tx \approx Ta + m\Delta V = Ta + [(Tb - Ta)/(Vb - Va)]\Delta V$. Assume calibration points have been taken every 0.1 V along the horizontal axis. Then $Vb - Va = 0.1$ V. Thus, $Tx = Ta + [10(Tb - Ta)](Vx - Va)$, where the parameters Ta, Tb, and Va were determined during the calibration process. With the above formula and calibration parameters in the CPU's memory, Tx can be calculated for any Vx from the transducer. Note that the more calibration points taken, the more accurate is the approximation.

We have now covered the important aspects of interfacing and some applications. Part 3 of this series will introduce us to programming the CPU module in its machine language. Also included will be the details of building and using the Program Development board. ◇

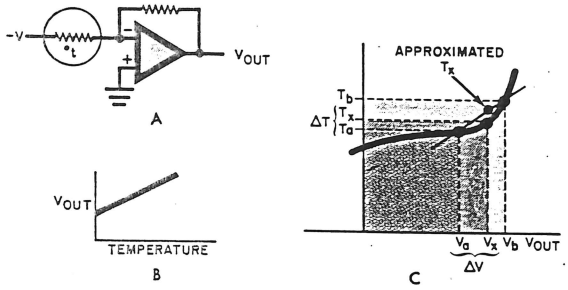
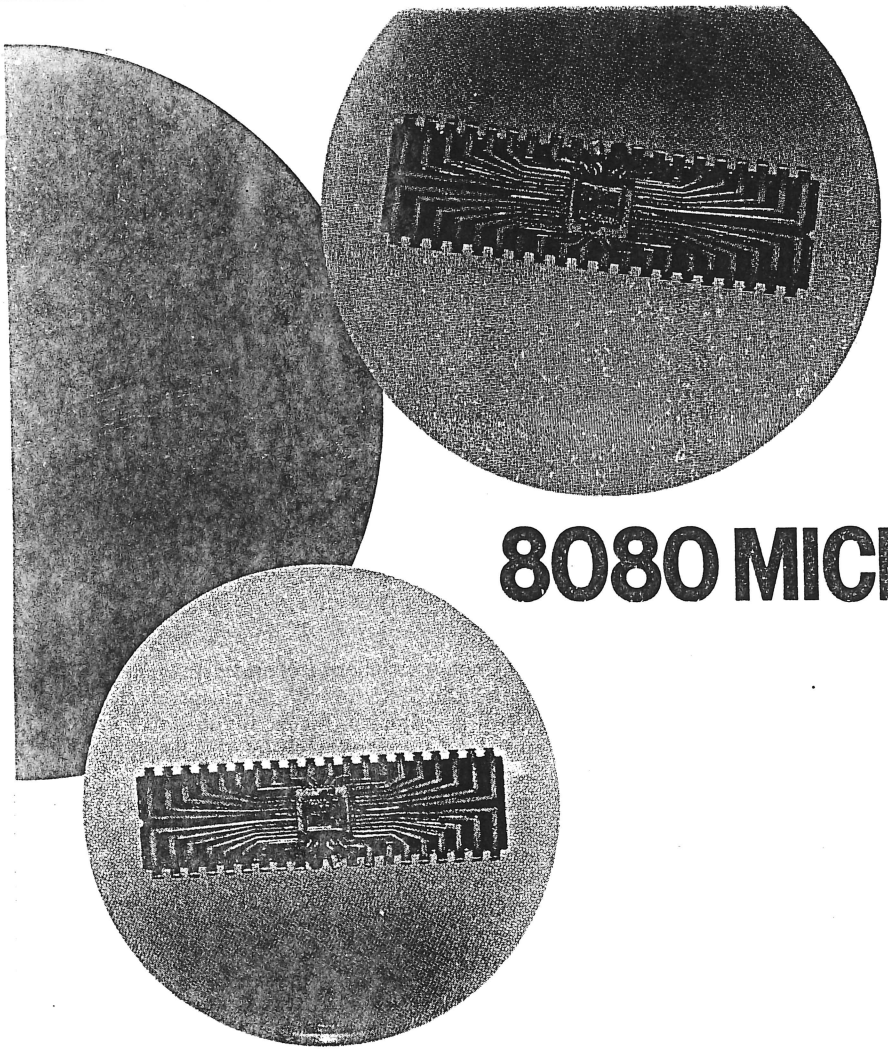


Fig. 16. A simple temperature transducer circuit (A); an ideal thermistor output characteristic (B); and how an actual curve is sampled to make a calibration curve to be stored in the CPU.



BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 3: Software

CPU instructions are defined with details on preparing a program

IN THE first two parts of this series we introduced 8080 I/O interfacing, general CPU architecture (though to a lesser extent), and a few applications. Until now we have limited the discussions to the microprocessor's *hardware*. But there is one other equally important realm of microprocessors that we must take into consideration: its *software*. A processor's software may be thought of as the collection of programs that have been developed for it.

CPU Instructions. Each instruction is stored in memory as a sequence of one, two, or three bytes. The first byte is called the *Instruction Code* or *Operation Code* (or simply *op code*), as it defines the general instruction or operation to be performed. For multiple-byte instructions the memory address of the op code is always the address of the instruction. The second byte of a multiple-byte instruction is called an *operand* and contains data or an address to modify or complete the instruction. The third byte of a three-byte instruction is a second operand, which again contains either data or an address to complete the instruction. The exact instruction format will depend upon the particular operation to be executed, as illustrated in Fig. 17. As an example of a three-byte instruction, consider one which operates on a byte of data contained in memory. The instruction must convey to the CPU *where* this data is located by specifying its memory address. Because memory addresses are 16 bits in length, two bytes will be required to specify the memory location. When a three-byte instruction references memory directly, the low-order bits of the memory address (which are sent out on A0-A7 of the address bus during execution of the instruction) are contained in byte 2, and the high-order bits (which are sent out on A8-A15) in byte 3.

The 8080 instruction set can manipu-

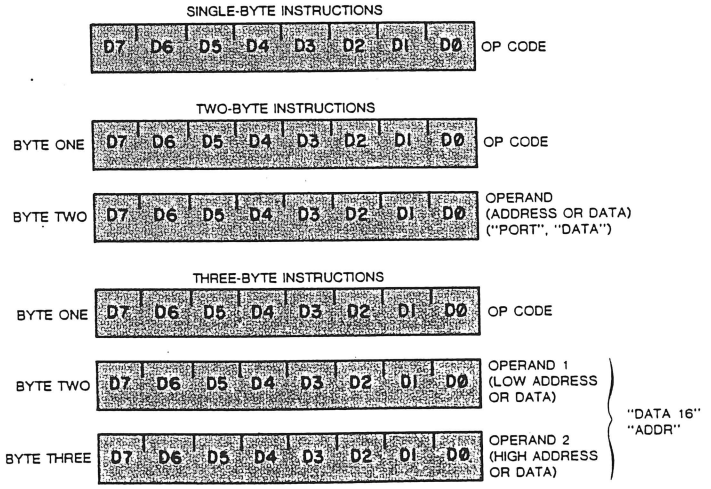


Fig. 17. 8080 instruction formats. Typical instructions for single-byte (top) are register-to-register, return, push, and pop; for two-byte: immediate mode and I/O; for three-byte: jump, call, and direct load. D0 is least significant bit; D7, most significant.

late seven 8-bit registers (internal to the CPU), a 16-bit Program Counter (PC), memory, I/O devices, and a 16-bit Stack Pointer (SP). The seven register names are A, B, C, D, E, H, and L, where register A is the all-important accumulator. Note that the SP and PC are 16 bits in length since they must be able to access any location in the 64K memory space. A number of 8080 operations use pairs of registers for 16-bit operands, and for these operations, register B is paired with register C, D with E, and H with L. Registers B, D, and H are the high-order bytes of these pairs, respectively. A register pair is designated by the name of the most significant byte of the pair. The SP is also considered to be a valid register pair. Figure 18 illustrates a way in which the 8080 registers may be visualized as paired together and stacked.

Conditionals. A number of 8080 instructions are "conditional," meaning that the full operation is performed *only if* a specified condition of the instruction is met. These conditionals are based on the state of one of the five processor flags. The flag bits are defined as:

Zero (Z): If the result of an instruction has the value zero, this flag is set; otherwise it is reset.

Sign (S): If the MSB of the result of

THE 8080 INPUTS AND OUTPUTS

IN OUR continuing discussion of the 8080 microprocessor, it is important to keep in mind the various input and output signals for the chip. They are as follows (and as shown in the diagram):

INT. By placing a low on this input line, an external device can interrupt the CPU operation. The interrupt system must be enabled for the CPU to honor the request (see INTE).

HOLD. An external device may suspend normal processor activity by putting a high on this input line. When acknowledged by the processor (HLDA), the address and data busses go into their high-impedance state so that the peripheral requesting the HOLD may conduct memory transfers without processor intervention (Direct Memory Access).

RDYIN. The Ready Input signal indicates to the CPU that valid data is available on the data bus. It is used to synchronize the CPU with slow memory or I/O devices. If, after sending out an address to an I/O device or memory, the CPU does not receive a high on this line, the CPU will idle as long as the line is low. If desired, the circuit can be arranged so that the CPU can be single-stepped (execute one instruction and halt) using this input.

RESIN. A low on the Reset Input line will clear the program counter, INTE and HLDA filp-flops. After reset, the CPU will begin program execution at memory location zero.

RESET. This output is used as a system reset for clearing external devices. It goes low when RESIN is active.

ϕ2. The phase-2 clock is a TTL output used for system timing. In a typical 8080 system, the clock frequency is approximately 2 MHz.

HLDA. This output goes high in response to a HOLD request and indicates that the data and address busses are going into their high-impedance state after the next rising edge of the phase-2 clock.

WAIT. This output goes high when the CPU enters a WAIT state (in response to RDYIN going low).

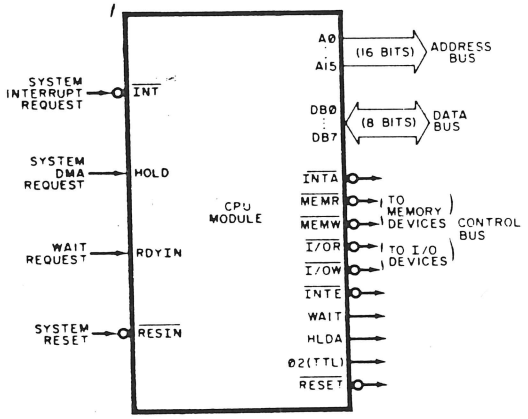
INTE. The Interrupt Enable output indicates the status of the CPU's internal interrupt enable flip-flop. This flip-flop can be set (INTE = low) using the 8080 "Enable Interrupts" instruction and reset (INTE = high) using the "Disable Interrupts" instruction. Interrupt requests on the INT line are ignored when the internal interrupt enable flip-flop is reset (when the interrupt system is disabled).

I/O W. The Input/Output Write line goes low when the CPU is executing an "Output" instruction, indicating that the address bus contains an output device address and the data bus contains the output data byte. Either the high- or low-order byte of the address bus (A15-A8 or A7-A0) may be decoded since both bytes contain the same information. I/O W may be used as a "strobe" signal to clock output latches, etc.

I/O R. The Input/Output Read line goes low when the CPU is executing an "Input" instruction, indicating that the address bus contains an input device address and that the input data may be placed on the data bus. Either the high- or low-order byte of the address bus (A15-A8 or A7-A0) may be decoded since both bytes contain the same information. I/O R may be used to gate input data onto the data bus.

MEMW. The Memory Write output goes low when the CPU is writing data to memory. The address bus contains the address of the memory location to be written into, while the data bus contains the data byte to be written. MEMW is usually connected to the read/write input of the RAM and strobes the data into the addressed location.

MEMR. The Memory Read output goes low when the CPU is reading data from memory (ROM or RAM). The address bus contains the address of the memory location to be read, while the data bus expects the memory data from the addressed memory location. MEMR is usually connected to a memory chip select or data buffer enable input to gate the read data onto the data bus.



INTA. The Interrupt Acknowledge output goes low in response to an INT request (assuming that the interrupt system is enabled). INTA is used to gate the "Restart" instruction from the interrupting device onto the data bus.

DB0-DB7. The data bus is formed from eight bi-directional lines (data may flow in either direction). This bus provides communication between memory, I/O devices, and the CPU for instructions and data transfers. The bus may go into its third (high-impedance) state in response to a HOLD request. DB0 is the least significant bit. The Control Bus provides the necessary timing signals to gate memory and I/O data on and off the data bus at the proper time.

A0-A15. The 16-line address bus provides the binary memory address during memory accesses (read or write) and also I/O device (port) numbers during input/output. Up to 65,536 bytes of memory (RAM, ROM, or any mixture thereof) and up to 256 input and 256 output devices can be directly addressed. The least significant bit is A0.

Through proper use of these CPU module signals, it is possible to design anything from a music synthesizer or darkroom controller to a full-blown computer system. It is not our intent here, however, to design a "computer" as such, but rather to demonstrate how this versatile device can be used to implement functions that would be impractical, or otherwise impossible, using traditional analog and digital circuits. ◇

the operation has the value 1, this flag is set; otherwise it is reset.

Parity (P): If there are an even number of binary 1's in the result of the operation (even parity), this flag is set; otherwise it is reset (odd parity).

Carry (CY): If the instruction resulted in a carry (from addition) or a borrow (from subtraction or a comparison) out of the high-order bit, this flag is set; otherwise it is reset. In this respect, the CY bit may be thought of as an extension, or ninth bit, of the register being operated on.

Auxiliary Carry (AC): If the instruction caused a carry out of bit 3 and into bit 4 of the resulting value, this flag is set; otherwise it is reset. This flag is affected by single precision (8-bit) additions, subtractions, increments, decrements, comparisons, and logical operations, but is principally used preceding a DAA (Decimal Adjust Accumulator) instruction.

We now know enough of the 8080 architecture to understand and use its machine language. Soon to follow is a summary of all 78 instructions of the 8080 instruction set. Once you are able to program the 8080, you will also be able to program the 8085 and are halfway to mastering the Z-80. Each machine instruction will be listed in its *assembly language* form, in which each instruction is assigned a unique mnemonic, making it easier to read and remember. A program written in assembly language is called a *source program.* A special program known as an *assembler* can then take this source program and generate the binary equivalent (machine code, or *object code*) of each instruction mnemonic. This binary object code is then stored in memory for the CPU hardware to execute. Assembler programs offer many other advantages; but since very few of us have access to a computer system with an 8080 assembler, source program assembly "by hand" will have to suffice. This is fairly easy with the aid of Fig. 19.

In the following summary of the 8080 instruction set, the term "addr" indicates that a 16-bit address must immediately follow the op code in program memory (refer to Fig. 17). Similarly, "data16" indicates a 16-bit data quantity must immediately follow the op code. In both cases, byte 2 of the instruction is the low-order address or data byte, and byte 3 is the high-order byte. The term "data" indicates that an 8-bit data quantity must immediately follow the op code in memory. The actions taken by each instruction on the five flag (condition) bits will be given with the description of each instruction. We will begin with the group of instructions called the "Data Transfer Group," which transfers data to and from the registers and memory. Condition flags are not affected by any instruction in this group.

Data Transfer Group.

MOV *r1, r2* (Move register to register)

The content of register *r2* is transferred (copied into) to register *r1*. Any of the seven register names are legal values for *r1* and *r2*. The content of register *r2* is not affected.

MOV *M, r* (Move register to memory)

The content of register *r* is transferred to the memory location whose address is

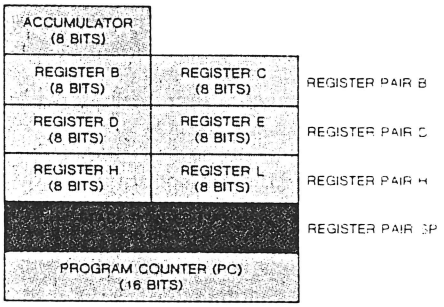


Fig. 18. The 8080 has seven 8-bit general-purpose registers and two 16-bit registers.

contained in register pair *H*.

MOV *r, M* (Move memory to register)

The content of the memory location, whose address is contained in register pair *H*, is transferred to register *r*.

MVI *M, data* (Move data immediate to memory)

The content of byte 2 of this instruction is transferred to the memory location whose address is contained in register pair *H*.

MVI *r, data* (Move data immediate to register)

The content of byte 2 of this instruction is transferred to register *r*.

LXI *rp, data16* (Load register pair with data immediate)

The 16-bit value of *data16* is stored in register pair *rp*.

STA *addr* (Store accumulator in memory)

The content of the accumulator is stored in the memory location specified by *addr*.

LDA *addr* (Load accumulator from memory)

The content of the memory location specified by *addr* is transferred to the accumulator.

SHLD *addr* (Store register pair *H* in memory)

The content of register *L* is stored in the memory location specified by *addr*. The content of register *H* is stored in the succeeding memory location *(addr+1)*.

LHLD *addr* (Load register pair *H* from memory)

The content of the memory location specified by *addr* is transferred to register *L*. The content of the succeeding memory location *(addr+1)* is transferred to register *H*.

STAX *rp* (Store accumulator indirect)

The content of the accumulator is stored in the memory location whose address is obtained from register pair *rp*. Only register pairs *rp=B* and *rp=D* can be specified.

LDAX *rp* (Load accumulator indirect)

The content of the memory location, whose address is obtained from register pair *rp*, is transferred to the accumulator. Only register pairs *rp=B* and *rp=D* can be specified.

XCHG (Exchange register pairs *H* and *D*)

The content of register *H* is exchanged with the content of register *D*. The content of register *L* is exchanged with the content of register *E*.

Arithmetic Group. This group performs arithmetic operations on data contained in registers and memory. Unless indicated otherwise, all instructions in this group affect the condition flags according to the standard rules. All subtraction operations are performed using two's-complement arithmetic and set the Carry flag to 1 to indicate a borrow, and clear it to indicate no borrow.

ADD *r* (Add register to accumulator)

The content of register *r* is added to the content of the accumulator, with the result placed in the accumulator.

ADD *M* (Add memory to accumulator)

The content of the memory location specified by register pair *H* is added to the content of the accumulator; the result is placed in the accumulator.

ADI *data* (Add data immediate to accumulator)

The content of the second byte of this instruction is added to the content of the accumulator; the result placed in the accumulator.

ADC *r* (Add register with carry to accumulator)

The content of register *r* and the content of the CY bit are added to the content of the accumulator. The result is placed in the accumulator.

ADC *M* (Add memory with carry to accumulator)

The content of the memory location specified by register pair *H* and the content of the CY bit are added to the content of the accumulator. The result is placed in the accumulator. This instruction is used primarily in "multiple precision" additions in which a number is actually contained in several, usually

adjacent memory locations. Such an addition begins at the low-order end (byte) of the two numbers being added using the ADD *M* instruction. Successive additions on more significant bytes of the numbers use the ADC *M* instruction, which corrects for overflow (carries) from preceding (less significant) additions.

ACI *data* (Add data immediate with carry to accumulator)

The content of the second byte of this instruction and the content of the CY bit are added to the content of the accumulator. The result is placed in the accumulator.

SUB *r* (Subtract register from accumulator)

The content of register *r* is subtracted from the content of the accumulator, with result placed in the accumulator.

SUB *M* (Subtract memory from accumulator)

The content of the memory location specified by register pair *H* is subtracted from the content of the accumulator. Result is placed in the accumulator.

SUI *data* (Subtract data immediate from accumulator)

The content of the second byte of this instruction is subtracted from the content of the accumulator. The result is placed in the accumulator.

SBB *r* (Subtract register with borrow from accumulator)

The content of register *r* and the content of the CY bit are subtracted from the content of the accumulator. The result is placed in the accumulator.

SBB *M* (Subtract memory with borrow from accumulator)

The content of the memory location specified by register pair *H* and the content of the CY bit are subtracted from the content of the accumulator. The result is placed in the accumulator. This is the "multiple precision" form of the SUB *M* instruction (see ADC *M*).

SBI *data* (Subtract data immediate with borrow from accumulator)

The content of the second byte of this instruction and the content of the CY bit are subtracted from the content of the accumulator. The result is placed in the accumulator.

INR *r* (Increment register)

The content of register *r* is incremented by one. The CY flag is not affected by this instruction.

INR *M* (Increment memory)

The content of the memory location specified by register pair *H* is incremented by one. The CY flag is not affected by this instruction.

DCR *r* (Decrement register)

The content of register *r* is decremented by one. The CY flag is not affected by this instruction.

DCR *M* (Decrement memory)

The content of the memory location specified by register pair *H* is decremented by one. The CY flag is not affected by this instruction.

INX *rp* (Increment register pair)

The content of register pair *rp* is incremented by one. No condition flags are affected.

DCX *rp* (Decrement register pair)

The content of register pair *rp* is decremented by one. No condition flags are affected.

JUMP

C3 JMP, C2 JNZ, CA JZ, D2 JNC, DA JC, E2 JPO, EA JPE, F2 JP, FA JM } Adr
E9 PCHL

CALL

CD CALL, C4 CNZ, CC CZ, D4 CNC, DC CC, E4 CPO, EC CPE, F4 CP, FC CM } Adr

RETURN

C9 RET, C0 RNZ, C8 RZ, D0 RNC, D8 RC, E0 RPO, E8 RPE, F0 RP, F8 RM

RESTART

C7 RST 0, CF RST 1, D7 RST 2, DF RST 3, E7 RST 4, EF RST 5, F7 RST 6, FF RST 7

ROTATE

07 RLC, 0F RRC, 17 RAL, 1F RAR

CONTROL

00 NOP, 76 HLT, F3 DI, FB EI

MOVE IMMEDIATE

06 MVI B, 0E MVI C, 16 MVI D, 1E MVI E, 26 MVI H, 2E MVI L, 36 MVI M, 3E MVI A } D8

Acc IMMEDIATE

C6 ADI, CE ACI, D6 SUI, DE SBI, E6 ANI, EE XRI, F6 ORI, FE CPI } D8

LOAD IMMEDIATE

01 LXI B, 11 LXI D, 21 LXI H, 31 LXI SP } D16

STACK OPS

C5 PUSH B, D5 PUSH D, E5 PUSH H, F5 PUSH PSW
C1 POP B, D1 POP D, E1 POP H, F1 POP PSW
E3 XTHL, F9 SPHL

DOUBLE ADD

09 DAD B, 19 DAD D, 29 DAD H, 39 DAD SP

SPECIALS

EB XCHG, 27 DAA, 2F CMA, 37 STC, 3F CMC

INCREMENT

04 INR B, 0C INR C, 14 INR D, 1C INR E, 24 INR H, 2C INR L, 34 INR M, 3C INR A

03 INX B, 13 INX D, 23 INX H, 33 INX SP

DECREMENT

05 DCR B, 0D DCR C, 15 DCR D, 1D DCR E, 25 DCR H, 2D DCR L, 35 DCR M, 3D DCR A

0B DCX B, 1B DCX D, 2B DCX H, 3B DCX SP

LOAD/STORE

0A LDAX B, 1A LDAX D, 2A LHLD Adr, 3A LDA Adr
02 STAX B, 12 STAX D, 22 SHLD Adr, 32 STA Adr

INPUT/OUTPUT

D3 OUT D8, DB IN D8

MOVE

40 MOV B,B, 41 MOV B,C, 42 MOV B,D, 43 MOV B,E, 44 MOV B,H, 45 MOV B,L, 46 MOV B,M, 47 MOV B,A
48 MOV C,B, 49 MOV C,C, 4A MOV C,D, 4B MOV C,E, 4C MOV C,H, 4D MOV C,L, 4E MOV C,M, 4F MOV C,A
50 MOV D,B, 51 MOV D,C, 52 MOV D,D, 53 MOV D,E, 54 MOV D,H, 55 MOV D,L, 56 MOV D,M, 57 MOV D,A

MOVE (cont)

58 MOV E,B, 59 MOV E,C, 5A MOV E,D, 5B MOV E,E, 5C MOV E,H, 5D MOV E,L, 5E MOV E,M, 5F MOV E,A
60 MOV H,B, 61 MOV H,C, 62 MOV H,D, 63 MOV H,E, 64 MOV H,H, 65 MOV H,L, 66 MOV H,M, 67 MOV H,A
68 MOV L,B, 69 MOV L,C, 6A MOV L,D, 6B MOV L,E, 6C MOV L,H, 6D MOV L,L, 6E MOV L,M, 6F MOV L,A
70 MOV M,B, 71 MOV M,C, 72 MOV M,D, 73 MOV M,E, 74 MOV M,H, 75 MOV M,L
77 MOV M,A
78 MOV A,B, 79 MOV A,C, 7A MOV A,D, 7B MOV A,E, 7C MOV A,H, 7D MOV A,L, 7E MOV A,M, 7F MOV A,A

ACCUMULATOR

80 ADD B, 81 ADD C, 82 ADD D, 83 ADD E, 84 ADD H, 85 ADD L, 86 ADD M, 87 ADD A
88 ADC B, 89 ADC C, 8A ADC D, 8B ADC E, 8C ADC H, 8D ADC L, 8E ADC M, 8F ADC A
90 SUB B, 91 SUB C, 92 SUB D, 93 SUB E, 94 SUB H, 95 SUB L, 96 SUB M, 97 SUB A
98 SBB B, 99 SBB C, 9A SBB D, 9B SBB E, 9C SBB H, 9D SBB L, 9E SBB M, 9F SBB A
A0 ANA B, A1 ANA C, A2 ANA D, A3 ANA E, A4 ANA H, A5 ANA L, A6 ANA M, A7 ANA A
A8 XRA B, A9 XRA C, AA XRA D, AB XRA E, AC XRA H, AD XRA L, AE XRA M, AF XRA A
B0 ORA B, B1 ORA C, B2 ORA D, B3 ORA E, B4 ORA H, B5 ORA L, B6 ORA M, B7 ORA A
B8 CMP B, B9 CMP C, BA CMP D, BB CMP E, BC CMP H, BD CMP L, BE CMP M, BF CMP A

D8 = constant, or logical arithmetic expression that evaluates to an 8 bit data quantity

D16 = constant, or logical/arithmetic expression that evaluates to a 16 bit data quantity.

Adr = 16 bit address

Fig. 19. Machine codes for the 8080 assembly language instructions. The hexadecimal machine code for each op code appears to the left of the instruction mnemonic.

DAD *rp* (Double precision add)

The content of register pair *rp* is added to the content of register pair *H*. The result is placed in register pair *H*. Only the CY flag is affected. It is set if there is a carry out of the double precision add; otherwise it is reset. Any register pair *(B, D, H, SP)* can be specified.

DAA (Decimal adjust accumulator)

The 8-bit number in the accumulator is adjusted to form two 4-bit binary-coded decimal digits by the following process:

1. If the value of the least significant four bits of the accumulator is greater than 9, or if the AC flag is set, 6 is added to the accumulator.
2. If the value of the most significant four bits of the accumulator is now greater than 9, or if the CY flag is set, 6 is added to the most significant four bits of the accumulator.

Logical Group. These instructions perform logical (Boolean) operations on data contained in registers, memory, and on condition flags. All instructions in this group affect the condition flags according to the standard rules, unless indicated otherwise.

ORA *r* (OR register with accumulator)

The content of register *r* is bit-wise logically inclusive-OR'd (Boolean addition) with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared. Each bit of the result is set to 1 if *either* of the corresponding accumulator/register bits is 1. For example, the value 10011101 OR'd with 01001011 will produce a result of 11011111 in the accumulator.

ORA *M* (OR memory with accumulator)

The content of the memory location specified by register pair *H* is logically inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

ORI *data* (OR data immediate with accumulator)

The content of byte 2 of this instruction is logically inclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

ANA *r* (AND register with accumulator)

The content of register *r* is bit-wise logically AND'ed (Boolean multiplication) with the content of the accumulator. The result is placed in the accumulator. The CY flag is cleared. Conceptually this operation is performed independently on each corresponding bit position of the accumulator and register *r*. The corresponding bit position in the result is set to 1 if, and only if, *both* of the corresponding accumulator/register bits are 1. The value 10011101 AND'ed with 01001011 will produce a result of 00001001 in the accumulator.

ANA *M* (AND memory with accumulator)

The content of the memory location specified by register pair *H* is logically AND'ed with the content of the accumulator. The result is placed in the accumulator. The CY flag is cleared.

ANI *data* (AND data immediate with accumulator)

The content of byte 2 of this instruction is logically AND'ed with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared. This instruction is often used to isolate or mask bits of the accumulator after an Input instruction for testing the status (ready/not ready) of external devices.

XRA *r* (Exclusive-OR register with accumulator)

The content of register *r* is bit-wise logically exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared. Each bit of the result is set to 1 if one, *and only one,* of the corresponding accumulator/register bits is 1. The value 10011101 exclusive-OR'd with 01001011 will produce a result of 11010110 in the accumulator. The instruction XRA *A* is often used to clear the accumulator and CY flag.

XRA *M* (Exclusive-OR memory with accumulator)

The content of the memory location specified by register pair *H* is logically exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

XRI *data* (Exclusive-OR data immediate with accumulator)

The content of byte 2 of this instruction is logically exclusive-OR'd with the content of the accumulator. The result is placed in the accumulator. The CY and AC flags are cleared.

CMP *r* (Compare accumulator with register)

The content of register *r* is subtracted from the content of the accumulator. The accumulator remains unaltered, and the condition flags are set as a result of the subtraction. The Z flag is set if the two values being compared are equal. The CY flag is set if the value in register *r* is greater than the value in the accumulator.

CMP *M* (Compare accumulator with memory)

The content of the memory location specified by register pair *H* is subtracted from the content of the accumulator. The accumulator remains unaltered. The condition flags are set as a result of the subtraction (see CMP *r*).

CPI *data* (Compare accumulator with data immediate)

The content of byte 2 of this instruction is subtracted from the content of the accumulator. The accumulator remains unaltered. The condition flags are set as a result of the subtraction (see CMP *r*).

There are four instructions in this group which are used to shift the contents of the accumulator. Each of these instructions shifts the accumulator bits one place left or right depending on the particular instruction. The only flag bit affected by these instructions is the CY flag. The directions "left" and "right" in the following descriptions assume that the more significant bits of the accumulator lie to the left.

RRC (Rotate right)

This is a circular right shift in which the CY bit receives the bit value shifted from the LSB of the accumulator. This same value shifted into the CY bit is also shifted into the MSB of the accumulator. For example, 00111001 becomes 10011100 after the shift and the CY bit is set to 1. Another shift of this value gives 01001110 and a CY value of 0.

RLC (Rotate left)

This shift is a left shift similar to RRC except the MSB is shifted into the CY bit and the LSB. All other accumulator bits are shifted left one position.

RAR (Rotate right through Carry)

This instruction shifts the accumulator contents one place right. The LSB is shifted into the CY bit as in the RRC instruction, but the old value of the CY bit is shifted into the MSB position of the accumulator. Shifting 00111001 with a value of 0 in the CY bit produces 00011100 and a CY value of 1. A second shift of this value produces 10001110 and a CY value of 0.

RAL (Rotate left through Carry)

The accumulator contents are shifted one place left with the MSB being sent to the CY bit and old value of the CY bit being shifted into the LSB position of the accumulator.

CMA (Complement accumulator)

The one's complement of the accumulator is placed in the accumulator. No condition flags are affected.

CMC (Complement Carry)

The CY flag is complemented, and no other flags are affected.

STC (Set Carry)

The CY flag is set to 1, and no other flags are affected. This instruction and the CMC instruction will affect all CY-related condition instructions as well as

the addition, subtraction, and shift instructions which use CY. These instructions are frequently used to return a status condition from a subroutine.

Branch Group. The 8080 is equipped with a full set of branch instructions which have the ability to alter the normal sequential flow of a program's execution. There are two types of branch instructions: conditional and unconditional. The execution sequence of a program is always altered by the unconditional type of transfer. The conditional type of transfer, on the other hand, examines the status of a condition flag in the instruction to see if the proposed branch is to be made. If the specified condition does not meet the requirement of the instruction, no branch is made and the program will resume execution at the next sequential instruction in memory. Condition flags are not affected by any instruction in this group. The conditions that can be specified are as follows:

Z - zero (Z=1)
NZ - not zero (Z=0)
C - carry (CY=1)
NC - no carry (CY=0)
PE - parity even (P=1)
PO - parity odd (P=0)
M - minus (S=1)
P - plus (S=0)

The AC flag cannot be used in a conditional branch instruction.

JMP *addr* (Unconditional jump)

Program control is unconditionally transferred to the memory address specified by *addr*. The next instruction executed will therefore be the one starting at this address (*i.e.*, the value of *addr* is moved into the PC).

J*condition addr* (Conditional jump)

A jump is made to the specified memory address if the specified condition is true (see JMP addr). If it is not true, program execution continues sequentially. There are actually eight unique instructions included here, since there are eight unique conditions that can be specified in the instruction op code (JZ, JNZ, JC, etc.).

PCHL (Jump H and L indirect)

This instruction performs the same operation as the JMP *addr* instruction except the transfer address is obtained from register pair *H*. This is most often used to branch to a routine in memory whose address has been computed or located in a table.

A transfer to a subroutine is made with one of the Call instructions to be described. When a call is made, two addresses become important. The "transfer address," the address of the subroutine being called, is contained in bytes 2 and 3 of the instruction (as in the Jump instructions). As the call is being made, however, a "return address" is stored (pushed) on the next available position (the top) of the Stack. The return address is obtained from the contents of the PC. The PC contains the address of the next sequential instruction. When the subroutine is finished, it can execute one of the Return instructions which will retrieve (pop) this address from the top of the Stack and perform a jump to this address. This return address represents the location of the instruction immediately following the Call instruction which gave control to the subroutine.

CALL *addr* (Unconditional call)

The most significant eight bits of the PC are stored in the memory location whose address is one less than the content of the SP (SP-1). The least significant eight bits of the PC are stored in the memory location whose address is two less than the content of the SP (SP-2). The content of the SP is decremented by two. A jump is then made to the memory location specified by *addr* (the branch address).

C*condition addr* (Conditional call)

The subroutine beginning at the specified memory address is called if the specified condition is true (see CALL *addr*). If it is not true, program execution continues sequentially. There are actually eight unique instructions included here, since eight unique conditions can be specified in the instruction op code (CZ, CNZ, CC, etc.).

RET (Unconditional return)

The content of the memory location specified by the SP is moved into the low-order byte of the PC. The content of the memory location whose address is one more than the content of the SP (SP+1) is moved into the high-order byte of the PC. The content of the SP is incremented by two. Care must be exercised when using this instruction to ensure that the Stack has been properly maintained in the subroutine, or a return may be made to an erroneous address.

R*condition* (Conditional return)

A return is made if the condition specified in the instruction is true (see RET). If it is not true, program execution continues sequentially. There are actually eight unique instructions included here, since eight unique conditions can be specified in the instruction op code (RZ, RNZ, RC, etc.).

RST *n* (Restart)

A Call is made to the memory location whose address is *eight times* the value of *n*, where *n* must be an integer value between 0 and 7. This is a single-byte instruction which is used primarily during program interrupt I/O, when a slow or sporadic peripheral has requested an interrupt from the CPU. It is similar to the subroutine Call instruction, except an external device will usually initiate this type of Call rather than the program itself. It is the responsibility of the peripheral that requested the interrupt to jam the Restart instruction's machine code onto the CPU's data bus for the CPU to execute during the Interrupt Acknowledge period ($\overline{\text{INTA}}$ low). If only one peripheral in the system is capable of requesting an interrupt, a RST 7 instruction may be automatically gated onto the data bus at the proper time by connecting a 1-kΩ resistor between the $\overline{\text{INTA}}$ output (P3-9) and +12 V. When

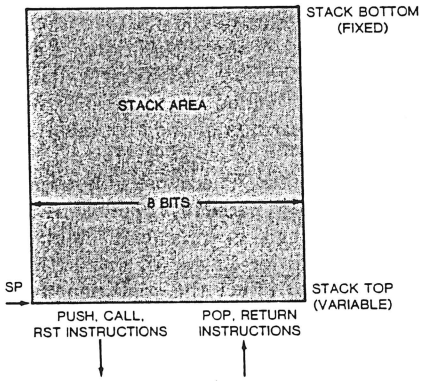


Fig. 20. The 8080 Stack is fixed at the bottom by the LXI SP, data 16 instruction and grows downward from there. The Stack Pointer keeps track of the top.

an interrupt is acknowledged, the interrupt system is immediately disabled ($\overline{\text{INTE}}$ high), keeping other interrupting peripherals from confusing things while the first interrupt is being handled. (Other methods exist which allow multiple interrupts to take place, that is, a second interrupt may interrupt the first interrupt, a third interrupt the second, etc. One such method is "interrupt vectoring," where each interrupting device is assigned a priority level and is serviced accordingly.) A routine to service the interrupting device must begin at the address specified by the Restart instruction. The possible Restart addresses are: 0000, 0008, 0010, 0018, 0020, 0028, 0030, and 0038 hexadecimal. When the interrupt-servicing routine is completed, control may be returned to the main program where the interrupt took place by executing one of the Return instructions (since RST *n* is a form of Call instruction).

Stack, I/O, and Machine Control Group. Unless indicated otherwise, the condition flags will not be affected by these instructions, which give the programmer direct control of the Stack and its pointer. The Stack can be a very ver-

satile data storage area for particular applications, but the programmer must be careful that the data stored in the Stack area is not confused with the return addresses stored there from subroutine Calls. Note that as data is stored (pushed) on the Stack, the Stack "grows downward" in memory. When data is retrieved (popped) from the Stack the reverse is true—the Stack "shrinks upward" in memory (Fig. 20).

PUSH *rp* (Push register pair)

The content of register pair *rp* is placed on the Stack in the following manner: The content of the high-order register of register pair *rp* is stored in the memory location whose address is one less than the content of the SP (SP-1). The content of the low-order register is stored in the memory location whose address is two less than the content of the SP (SP-2). The SP is decremented by two. Register pair *rp*=SP can not be specified.

POP *rp* (Pop register pair)

This instruction performs the inverse operation of the PUSH *rp* instruction. The content of the memory location specified by the SP is moved into the low-order register of register pair *rp*. The content of the succeeding memory location (SP+1) is moved into the high-order register of register pair *rp*. The content of the SP is incremented by two. Register pair *rp*=SP may not be specified.

PUSH PSW (Push processor status word)

The content of the accumulator is stored in the memory location whose address is one less than the content of the SP (SP-1). The processor flags are assembled into what is called the "processor status word," which is then stored in the memory location whose address is two less than the content of the SP (SP-2). The SP is decremented by two. The processor status word is assembled as follows:

D7	D6	D5	D4	D3	D2	D1	D0
S	Z	0	AC	0	P	1	CY

POP PSW (Pop processor status word)

This instruction performs the inverse operation of the PUSH PSW instruction. The content of the memory location specified by the SP is disassembled and moved into the processor flag bits. The content of the succeeding memory location (SP+1) is moved into the accumulator. The content of the SP is incremented by two.

SPHL (Move register pair *H* into SP)

The content of register pair *H* is moved into the Stack Pointer, destroying its previous contents. This provides a convenient way of changing the SP during a program, thereby allowing two or more Stacks to exist at once (one for subroutine control, one for data, etc.).

XTHL (Exchange top of Stack with register pair *H*)

The content of register *L* is exchanged with the content of the memory location whose address is specified by the SP. The content of register *H* is exchanged with the content of the succeeding memory location (SP+1).

IN *port* (Input)

This instruction reads the specified port and stores the data byte which it read (via the data bus) in the accumulator. During execution of this instruction, the specified port number is sent out on the high- and low-order bytes of the address bus for Port-Select decoding by the interface. The port number must be an integer value between 00 and FF_{16}. Attempting to read a nonexistent port with this instruction will place FF_{16} in the accumulator.

OUT *port* (Output)

This instruction writes the content of the accumulator to the specified port via the data bus. During execution of this instruction, the specified port number is sent out on the high and low-order bytes of the address bus for port-select decoding by the interface. The port number must be an integer value between 00 and FF_{16}. The content of the accumulator is unaltered. (See Part II for interfacing techniques).

EI (Enable interrupts)

The interrupt system is enabled ($\overline{\text{INTE}}$ *low*) following the execution of the *next* instruction. The CPU will then honor the next interrupt requested by an external device.

DI (Disable interrupts)

The interrupt system is disabled ($\overline{\text{INTE}}$ *high*) immediately following execution of this instruction. Devices attempting to interrupt the CPU with the interrupt system disabled will be ignored by the CPU and its related hardware. The interrupt system is automatically disabled when an interrupt is acknowledged or when the CPU is reset.

HLT (Halt)

The CPU is completely stopped by this instruction and can be reactivated in only two ways. One is to reset the CPU by forcing the $\overline{\text{RESIN}}$ input (P3-11) low, which will also reset the Program Counter. The other is to interrupt the CPU. Should the interrupt system be disabled at the time this instruction executes, the CPU must be reset to exit the Halt state.

NOP (No op)

No operation is performed. This instruction can be used in programs under development to reserve space in memory where changes are expected to be made. It may also be used to "delete" unwanted instructions in a program.

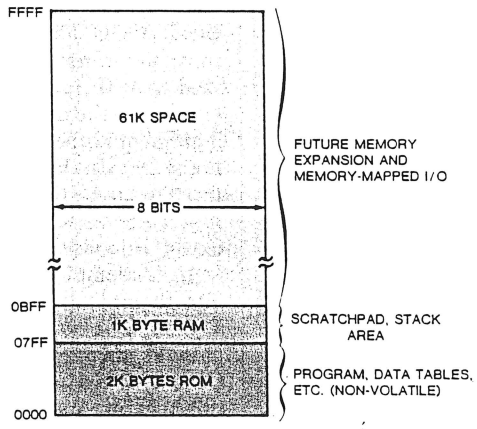


Fig. 21. Configuration of the CPU module memory. Permanent data are stored in ROM. Stack and scratchpad area is from 0800 to 0BFF. Top part is for memory expansion.

Writing Software for the CPU Module. It is easy to write programs for the CPU module. There are basically three steps to writing programs, and Fig. 21 will aid in the following summary of these steps.

1. Determine the maximum size (in bytes) that the Stack will be. This is most easily done by estimating the maximum number of nested subroutines and Push instructions that will be active in your program at any given time. This number should then be multiplied by two since each Call and Push instruction will use two bytes of Stack storage area. It is usually a good idea to increase this estimated value by a factor of 10% or 20% to ensure enough RAM will be reserved for the Stack area, just in case you underestimated the maximum Stack depth.

Now add the value just determined to the starting address of the RAM area

(800_{16}). This will be the address of the bottom of the Stack. Any RAM remaining above this address is free for any other use you may have (scratchpad, parameter storage, etc.)

2. Write your source program. A LXI SP, *data16* instruction initializing the SP to the value computed in step 1 should be 23_{16} and the six sensor outputs are connected to bits 0-5 of the port (bits 6 and 7 are grounded). Also assume that, when a forced-entry is detected, the triggered sensor's output goes high (+5 V). Try to write this subroutine yourself before reading any further. If you cannot, see the sample in the box:

SAMPLE PROGRAM

Mem. Address	Machine Code	Mnemonic	Explanation
0100	DB 23	IN 23H	Input status of sensors
0102	B7	ORA A	Update flags
0103	C8	RZ	Return with CY=0 if all sensors are 0 (Z flag=1 from last instruction)
0104	37	STC	Otherwise set CY=1
0105	C9	RET	and return to main program

included in the beginning of your program before any Stack-related instructions (Calls, Pushes, etc.) appear. For most programs, the SP may be initialized to BFF_{16}, the end of RAM, which eliminates the need of performing step 1. This should be done only if you can make certain the Stack will not be interfered with by other data the program may store in RAM. If any uncertainty exists, it is safer to perform step 1 first.

3. "Assemble" your source program into the object program with the aid of Fig. 19. The object program *must* originate at memory location 0 since the CPU automatically begins execution at this location at power-up. If your program must begin at a memory location other than 0 for one reason or another, a JMP *addr* instruction may be stored at location 0 which will transfer control to the beginning of your program (in this case *addr*=starting address of your program).

Sample Program. Let's try to write a short-machine-language program using the 8080 instruction set to demonstrate how easy it really can be.

You have just installed six sensors in various areas of your house for a new security system. Each sensor output is two-state (+5 or 0 volts) and is connected to an existing input port of your CPU module. In an attempt to devise the most complex security system in the neighborhood you are now confronted with the task of writing the software. Write a machine-language subroutine beginning at memory location 100_{16}, which will return to the calling program with the CY flag *set* if any one of the sensors has detected an intrusion. The subroutine should otherwise return with the CY flag *reset* if the house has been determined to be "secure." Assume that the input interface Port-Select circuitry has been wired to decode I/O port address

This is how a typical assembly listing would appear after an assembler program assembled the source program. This is also a good method to follow when writing your own programs without an assembler program.

The above subroutine is, of course, only *one* of a number of ways in which the algorithm can be written. You may have come up with something entirely different, but which, in fact, performs the same function. The fewer instructions used, however, the better (to save memory space and execution time).

The instruction following the one which called the above subroutine could be a conditional branch instruction which will branch-on-carry to another routine that determines which area of the house has the uninvited guest. This could be done by repeated use of the rotate instructions and testing the CY flag after each rotate. Another routine could take appropriate action—turn on the lights, sound an alarm, call the police—whatever you desire. And don't forget that heat, smoke, and moisture detectors can also be interfaced to the CPU.

If turning on lights is a response to an intrusion, the hardware already exists to automatically cycle the lights on and off systematically or randomly during vacation periods (which would give the house the appearance of being occupied). All that need be done is to write an appropriate program, which may even be stored in the same ROM with the security program. By interfacing a clock chip to the CPU (such as the National MM58167 or MM58174), it is a simple matter to write a short program which will turn on the lights and coffee pot before you get up in the morning.

To gain experience in writing longer programs, next month we will show how to write a program to receive Morse code off the air. ◇

Popular Electronics®

WORLD'S LARGEST SELLING ELECTRONICS MAGAZINE DECEMBER 1981/$1

THE ELECTRONIC WORLD

A User's Guide to Computer Languages

Detect Car-Battery Drain Before It's Too Late

Comparing New High-Tech Audio Cassettes

0 14024 14278 12

LOOK! ...ts IBM's New Personal Computer

BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 4: A Typical Program

PREVIOUS parts of the article have described how a typical microprocessor works and gave design details for a practical module. Last month, we covered the details of designing software for the system. Here is an example.

Morse Receiving Program. What would Samuel F.B. Morse have thought if he had heard his code being sent at 100 words per minute at the same time it was being printed out on a printer or television screen? Actually, the process involved in such a scheme is fairly simple and straight forward (if you happen to have a microprocessor). The procedure involves converting Mr. Morse's combinations of dots and dashes (which may come from a shortwave receiver) into a more modern and usable code called *ASCII* (American Standard Code for Information Interchange). ASCII is a widely-used code in which 96 displayable characters (letters, number, etc.) and 32 non-displayable control characters are each assigned a unique 7-bit code. For instance, the character "A" is represented in the ASCII code as 41_{16}; the character "3" is assigned the code 33_{16} (see Fig. 22). Many of today's computers communicate with each other using the ASCII code, and ASCII has even found its way into radio.

The 8080 readily handles the ASCII code since each register and memory location is capable of holding one ASCII character code. Occasionally the unused eighth bit (MSB) is utilized as a *parity bit*, which serves to minimize data errors when large amounts of information are being transferred. (This is the principal use of the Parity flag-related instructions in the 8080 instruction set.) The parity bit will not concern us in our discussions, however; we will assume it is always 0.

The Morse code is comprised of two types of marks—the dot and dash—and three types of spaces—the mark space, character space, and word space. Theoretically, the length of a dash is three times the length of a dot. Likewise, a mark space is equal in length to a dot, a character space equal to a dash, and a word space equal to seven dots.

Character	ASCII Code (hex)	Morse Code	Character	ASCII Code (hex)	Morse Code
A	41	• —	0	30	— — — — —
B	42	— • • •	1	31	• — — — —
C	43	— • — •	2	32	• • — — —
D	44	— • •	3	33	• • • — —
E	45	•	4	34	• • • • —
F	46	• • — •	5	35	• • • • •
G	47	— — •	6	36	— • • • •
H	48	• • • •	7	37	— — • • •
I	49	• •	8	38	— — — • •
J	4A	• — — —	9	39	— — — — •
K	4B	— • —			
L	4C	• — • •	.	2E	• — • — • —
M	4D	— —	,	2C	— — • • — —
N	4E	— •	?	3F	• • — — • •
O	4F	— — —	:	3B	— — — • • •
P	50	• — — •	:	3A	— — — • • •
Q	51	— — • —	(	28	— • — — • —
R	52	• — •	/	2F	— • • — •
S	53	• • •	-	2D	— • • • —
T	54	—	End of mes-	—	• — • — •
U	55	• • —	sage		
V	56	• • • —	Wait	—	• — • • •
W	57	• — —	End of work	—	• • • — • —
X	58	— • • —	Invitation to	—	— • — — •
Y	59	— • — —	transmit		
Z	5A	— — • •	Error	—	• • • • • • • •

Fig. 22. The ASCII code for the Morse symbols.

Ideally, a Morse recognition algorithm would test mark and space lengths at the points half-way between mark and space types, *e.g.*, 2/3 of a dash would be the "critical point" for deciding if a received mark was a dot or a dash, etc. However, after much experimentation with these critical points, a slightly modified algorithm was obtained. For instance, it was observed that many users of the code (hams in particular) tend to cut character spaces too short, which would confuse the ideal algorithm. For this reason, the character space critical point was changed from the ideal 2/3 dash to 1/2 dash (a 17% reduction), which enhanced character-recognition probability. A similar change was made for dot recognition for the same reason. Attempting to change or experiment with these critical points in a random-logic implementation of the code converter would be a large chore in itself.

The program has incorporated into it a subroutine for checking some types of noise often encountered while receiving Morse code on a radio receiver. This routine has proven to be quite effective in discriminating between some types of random noise and Morse code. In this respect, the routine can be thought of as a "software noise blanker." The "bandwidth" can be adjusted simply by changing the value of a parameter byte at memory location 010C of the program (the beauty of using a microprocessor). The bandwidth, and therefore the maximum code speed that may be accurately received, is inversely proportional to the numeric value stored in this memory location.

The Morse program assembly listing is shown here. The source program, which appears in the third column, was assembled beginning at memory location 0. The memory locations in the first column are followed by the object code contained, or assembled, into them. The second column lists the line numbers of the source program which are of no particular significance to the program itself.

Throughout the source program are "labels" to the immediate left of some of the instruction mnemonics. These labels represent relative addresses which are later assigned memory addresses during the assembly process. Labels allow the source program to reference other parts of the program (such as subroutines) without the need of knowing their memory addresses. For example, to Call the subroutine which checks for noise (labeled "VLDMK"), the corresponding source program instruction would be written as CALL VLDMK. VLDMK symbolizes the beginning address in program memory of the noise-checking subroutine. A list of all labels used in the Morse program, followed by their computed values and source program line numbers that use them, appears at the end of the assembly listing.

When assembling a source program by hand, the numeric values of any labels used must be determined before the corresponding machine language instructions can be completely assembled (labels will usually complete an instruction's operand, such as an address). In the case of the VLDMK label, its address was determined to be 0104_{16} after counting the number of bytes that preceded it in memory. So, in this case, "CALL VLDMK" and "CALL 0104H" are equivalent instructions and may be used interchangeably in the source program. It is more convenient, however, to use the symbolic label representation since the memory address may not even be known until the source program is completed. For this reason, and the fact that program changes are almost inevitable in the course of writing new programs (which will most likely change all the memory addresses of the instructions following the program changes), it is recommended that calculating memory addresses be one of the last steps performed when writing new programs. Using labels freely throughout the source program will help avoid some of the tedious work when program changes have to be made.

The next part of this article will cover the hardware implementation of the CPU module. ◇

MORSE-TO-ASCII CONVERSION PROGRAM

```
0000                    0100  ORG 0000
0000                    0105 *
0000                    0110 LEN EQU 64          TERMINAL WIDTH
0000                    0115 CW EQU 0FCH         MORSE INPUT PORT
0000                    0120 MASK EQU 01         DATA INPUT BIT 0 HAS MORSE
0000                    0125 PSTN EQU 0BFFH      LOCATE PSTN BYTE AT END OF RAM AREA
0000                    0130 *
0000                    0135 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
0000                    0140 *                                                     *
0000                    0145 * * * *   MORSE-TO-ASCII CONVERSION PROGRAM   * * * *
0000                    0150 *                                                     *
0000                    0155 * *                WRITTEN  BY                      * *
0000                    0160 *                                                     *
0000                    0165 * * * *          RANDY CARLSTROM, 9/78          * * * *
0000                    0170 *                                                     *
0000                    0175 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
0000                    0180 *
0000                    0185 * * * *      MORSE RECOGNITION ALGORITHM      * * * *
0000                    0190 *                                                     *
0000                    0195 *      IF THE NEW MARK LENGTH IS GREATER THAN OR      *
0000                    0200 * EQUAL TO TWICE THE LENGTH OF THE LAST MARK, THE     *
0000                    0205 * NEW MARK IS STORED AS A DASH.  IF THE NEW MARK       *
0000                    0210 * LENGTH IS LESS THAN ONE-HALF THE LENGTH OF THE      *
0000                    0215 * LAST MARK, THE NEW MARK IS STORED AS A DOT.  IF     *
0000                    0220 * THE NEW MARK DOES NOT FALL INTO EITHER ONE OF       *
0000                    0225 * THESE CATEGORIES, THE NEW MARK IS STORED AS THE     *
0000                    0230 * SAME TYPE AS THE LAST MARK.                         *
0000                    0235 *      IF A SPACE LENGTH REACHES ONE-HALF THE         *
0000                    0240 * LENGTH OF THE LAST DASH, A CHARACTER SPACE HAS       *
0000                    0245 * BEEN RECEIVED AND THE ELEMENTS ARE ASSEMBLED AND    *
0000                    0250 * CONVERTED TO THE CORRESPONDING ASCII CHARACTER.     *
0000                    0255 * THE CHARACTER IS THEN PRINTED AT THE OUTPUT          *
0000                    0260 * PERIPHERAL AND SPACE MEASUREMENT CONTINUES.          *
0000                    0265 * IF THE SPACE LENGTH REACHES THREE-HALVES THE         *
0000                    0270 * LENGTH OF THE LAST DASH, A WORD SPACE HAS BEEN      *
0000                    0275 * RECEIVED AND THE SPACE IS PRINTED.                  *
0000                    0280 *                                                     *
0000                    0285 * * * * * * *   REGISTER   ALLOCATION   * * * * * * *
0000                    0290 *                                                     *
0000                    0295 * B = DASH REG.  C = DOT REG.  D = LAST MARK LENGTH   *
0000                    0300 * E = LAST MARK TYPE: 00 = DOT, 01 = DASH             *
0000                    0305 * H = MARK/SPACE COUNTER  L = LAST DASH LENGTH        *
0000                    0310 *                                                     *
0000                    0315 * * * * * * * * *  INITIALIZATION  * * * * * * * * *
0000                    0320 *                                                     *
0000 F3                 0325        DI           DISABLE INTERRUPTS
0001 31 FD 0B           0330        LXI SP,0BFDH INITIALIZE STACK POINTER IN RAM
0004 CD AC 00           0335        CALL CRLF    START A NEW OUTPUT LINE
0007 21 40 00           0340        LXI H,0040H  LAST DASH LENGTH = 64; COUNTER = 0
000A 11 00 20           0345        LXI D,2000H  LAST MARK LENGTH = 32; LAST MARK = DOT
000D 01 00 00           0350        LXI B,0000   CLEAR DOT AND DASH REGISTERS
0010                    0355 *
0010                    0360 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
0010                    0365 *                                                     *
0010                    0370 * * * * * * * * *  MAIN  PROGRAM  * * * * * * * * * *
0010                    0375 *
0010 CD 04 01           0380 WAIT   CALL VLDMK   IS THERE A VALID MARK YET?
0013 DA 10 00           0385        JC WAIT      NO: WAIT
0016 CD 4B 00           0390 NEWMK  CALL MKTYP   YES: FIND MARK TYPE AND STORE IT
0019 26 00              0395        MVI H,00     CLEAR MARK/SPACE COUNTER
001B CD 04 01           0400 SPTYP  CALL VLDMK   IS THERE A VALID MARK YET?
001E D2 16 00           0405        JNC NEWMK    YES: PROCESS NEW MARK
0021 B7                 0410        ORA A        NO: CLEAR CY FLAG AND
0022 7C                 0415        MOV A,H      CHECK SPACE LENGTH
0023 17                 0420        RAL          REG. A = 2 * CURRENT SPACE LENGTH
0024 BD                 0425        CMP L        SPACE >= 1/2 LAST DASH? (CHAR. SPACE)
0025 D2 1B 00           0430        JC SPTYP     NO: CONTINUE MEASURING SPACE LENGTH
0028 CD 7A 00           0435        CALL CONVT   YES: CONVERT MORSE TO ASCII
```

```
002B CD BB 00          0440          CALL OUT        AND PRINT IT, THEN
002E CD DB 00          0445          CALL IPSTN      UPDATE PRINTER POSITION
0031 FE 3E             0450          CPI LEN-2       DO A CR/LF IF AT END OF LINE
0033 D4 AC 00          0455          CNC CRLF
0036 CD 04 01          0460 GENSP    CALL VLDMK      IS THERE A VALID MARK YET?
0039 D2 16 00          0465          JNC NEWMK       YES: PROCESS NEW MARK
003C B7                0470          ORA A           NO: CLEAR CY FLAG AND
003D 7D                0475          MOV A,L         CHECK SPACE LENGTH
003E 1F                0480          RAR
003F 85                0485          ADD L           REG. A = 3/2 LAST DASH
0040 3D                0490          DCR A
0041 BC                0495          CMP H           SPACE >= 3/2 LAST DASH? (WORD SPACE)
0042 D2 36 00          0500          JNC GENSP       NO: CONTINUE MEASURING SPACE LENGTH
0045 CD E3 00          0505          CALL SPOUT      YES: PRINT A SPACE OR CR/LF
0048 C3 10 00          0510          JMP WAIT        DONE; WAIT FOR A NEW CHAR. TO BEGIN
004B                   0515 *
004B                   0520 *
004B                   0525 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
004B                   0530 * * * *                SUBROUTINES                * * * *
004B                   0535 * * * * * * * * * * * * * * * * * * * * * * * * * * * *
004B                   0540 *
004B                   0545 * MKTYP COMPARES THE NEW MARK WITH THE PREVIOUS MARK TO
004B                   0550 * SEE IF THE NEW MARK IS A DOT OR A DASH. IF THE NEW
004B                   0555 * MARK IS < 1/2 THE PREVIOUS ONE, IT IS A DOT. IF THE
004B                   0560 * NEW MARK IS >= 2 * THE PREVIOUS ONE, IT IS A DASH.
004B                   0565 * OTHERWISE THE NEW MARK IS THE SAME AS THE PREVIOUS MARK.
004B                   0570 *
004B 7A                0575 MKTYP MOV A,D
004C 17                0580          RAL             REG. A = 2 * LAST MARK LENGTH
004D 3D                0585          DCR A
004E BC                0590          CMP H           MARK >= 2 * LAST MARK LENGTH? (DASH)
004F D2 5F 00          0595          JNC DOT         NO: GO SEE IF A DOT
0052 54                0600 DASH     MOV D,H         YES: UPDATE LAST MARK LENGTH REGISTER
0053 6C                0605          MOV L,H         AND LAST DASH LENGTH REGISTER
0054 1E 01             0610          MVI E,01        UPDATE LAST MARK TYPE REGISTER
0056 78                0615          MOV A,B         SHIFT A "1" INTO DASH REGISTER
0057 07                0620          RLC
0058 F6 01             0625          ORI 01
005A 47                0630          MOV B,A
005B 79                0635          MOV A,C         SHIFT A "0" INTO DOT REGISTER
005C 07                0640          RLC
005D 4F                0645          MOV C,A
005E C9                0650          RET             A DASH WAS RECEIVED AND STORED; RETURN
005F 7C                0655 DOT      MOV A,H
0060 17                0660          RAL             REG. A = 2 * CURRENT MARK LENGTH
0061 3D                0665          DCR A
0062 BA                0670          CMP D           IS MARK < 1/2 LAST MARK LENGTH? (DOT)
0063 D2 72 00          0675          JNC SAME        NO: WAS THE SAME AS LAST MARK
0066 54                0680 DOTT     MOV D,H         YES: UPDATE LAST MARK LENGTH REG.
0067 1E 00             0685          MVI E,00        AND LAST MARK TYPE REGISTER
0069 79                0690          MOV A,C         SHIFT A "1" INTO DOT REGISTER
006A 07                0695          RLC
006B F6 01             0700          ORI 01
006D 4F                0705          MOV C,A
006E 78                0710          MOV A,B         SHIFT A "0" INTO DASH REGISTER
006F 07                0715          RLC
0070 47                0720          MOV B,A
0071 C9                0725          RET             A DOT WAS RECEIVED AND STORED; RETURN
0072 7B                0730 SAME     MOV A,E         GET LAST MARK TYPE
0073 1F                0735          RAR             WAS IT A DASH?
0074 DA 52 00          0740          JC DASH         YES: GO TO DASH ROUTINE
0077 C3 66 00          0745          JMP DOTT        NO: GO TO DOT ROUTINE
007A                   0750 *
007A                   0755 *
007A                   0760 * CONVT ASSEMBLES THE DOT AND DASH REGISTERS IN SUCH
007A                   0765 * A WAY THAT IT FORMS A NEW UNIQUE 8-BIT CODE. THEN
007A                   0770 * A TABLE OF THESE UNIQUE CODES IS SEARCHED FOR A MATCH,
007A                   0775 * AND THE CORRESPONDING ASCII CHARACTER IS RETURNED IN
007A                   0780 * THE ACCUMULATOR. IF NO MATCH WAS FOUND, AN ERROR CHAR.
007A                   0785 * IS RETURNED.
007A                   0790 *
007A 79                0795 CONVT MOV A,C         GET DOT REGISTER
007B 07                0800          RLC             MULTIPLY IT BY 2
007C 80                0805          ADD B           AND ADD DASH REGISTER
007D 01 00 00          0810          LXI B,0000      READY DOT AND DASH REG. FOR NEXT CHAR.
0080 E5                0815          PUSH H          REG. A NOW HAS UNIQUE CODE
0081 21 19 01          0820          LXI H,TABLE     POINT TO TABLE STARTING ADDRESS
0084 BE                0825 NEXT     CMP M           FOUND CHARACTER?
0085 CA 94 00          0830          JZ ASCII        YES: CONVERT TO ASCII
0088 F5                0835          PUSH PSW        NO: SAVE UNIQUE CODE AND
0089 7E                0840          MOV A,M         CHECK FOR END OF TABLE
008A B7                0845          ORA A           END OF TABLE?
008B CA 98 00          0850          JZ ERROR        YES: NO MATCH WAS FOUND
008E F1                0855          POP PSW         NO: GET UNIQUE CODE BACK AND
008F 23                0860          INX H           BUMP REG. PAIR H TO NEXT
0090 23                0865          INX H           CHARACTER ADDRESS
0091 C3 84 00          0870          JMP NEXT        AND TRY AGAIN
0094 23                0875 ASCII    INX H           POINT TO ASCII CHAR. IN TABLE
0095 7E                0880          MOV A,M         AND PUT IT IN REG. A
0096 E1                0885          POP H
0097 C9                0890          RET             RETURN WITH IT
0098 F1                0895 ERROR    POP PSW         ELSE RETURN WITH A "*"
0099 E1                0900          POP H           IF NO MATCH WAS FOUND
009A 3E 2A             0905          MVI A,'*'
009C C9                0910          RET
009D                   0915 *
009D                   0920 *
009D                   0925 * DLY1 IS A TIME DELAY USED WHEN MEASURING THE LENGTH
009D                   0930 * OF A MARK OR SPACE. THE VALUE OF REG. B ACTS AS A
009D                   0935 * "COURSE-ADJUST," WHILE THE VALUE OF REG. C ACTS MORE
009D                   0940 * LIKE A "FINE-ADJUST." THE PRESET VALUES FOR REGISTERS
009D                   0945 * B AND C MAY BE CHANGED FOR DIFFERENT CODE SPEEDS OR
009D                   0950 * CPU'S, BUT THESE VALUES ARE OPTIMUM FOR ON-THE-AIR USE
009D                   0955 * USING AN 8080, 8085, OR Z-80 CPU RUNNING AT 2 MHZ.
009D                   0960 * REG. H SHOULD NOT BE ALLOWED TO COUNT ABOVE 7FH DURING
```



```
009D                   0965 * A MARK MEASUREMENT, OR THE MORSE ALGORITHM WILL FAIL.
009D                   0970 * INCREASING THE VALUE OF REGISTER PAIR B INCREASES THE
009D                   0975 * LENGTH OF THE TIME DELAY, THEREBY DECREASING THE FINAL
009D                   0980 * COUNT OF REG. H FOR A GIVEN MARK OR SPACE LENGTH.
009D                   0985 *
009D C5                0990 DLY1  PUSH B
009E 06 03             0995       MVI B,03        PRESET VALUE FOR REG. B
00A0 0E 40             1000       MVI C,40H       PRESET VALUE FOR REG. C
00A2 0D                1005 LOOP1 DCR C           KILL SOME TIME...
00A3 C2 A2 00          1010       JNZ LOOP1
00A6 05                1015       DCR B
00A7 C2 A2 00          1020       JNZ LOOP1
00AA C1                1025       POP B
00AB C9                1030       RET
00AC                   1035 *
00AC                   1040 *
00AC                   1045 * CRLF OUTPUTS A CARRIAGE-RETURN LINE-FEED WHEN CALLED.
00AC                   1050 *
00AC 3E 0D             1055 CRLF  MVI A,0DH       OUTPUT A CR
00AE CD BB 00          1060       CALL OUT
00B1 3E 0A             1065       MVI A,0AH       OUTPUT A LF
00B3 CD BB 00          1070       CALL OUT
00B6 AF                1075       XRA A           SET PRINT POSITION COUNTER TO 0
00B7 32 FF 0B          1080       STA PSTN
00BA C9                1085       RET
00BB                   1090 *
00BB                   1095 *
00BB                   1100 * OUT SENDS THE CHAR. TO BE PRINTED TO THE OUTPUT
00BB                   1105 * DEVICE (CRT, PRINTER, ETC.). THERE IS RESERVED
00BB                   1110 * STORAGE SPACE HERE FOR THE USER'S OWN OUTPUT ROUTINE.
00BB                   1115 *
00BB C5                1120 OUT   PUSH B          MY OUTPUT DRIVER NEEDS THE CHAR.
00BC 47                1125       MOV B,A         IN REG. B
00BD CD 19 C0          1130       CALL 0C019H     CALL MY MONITOR'S OUTPUT DRIVER
00C0 C1                1135       POP B           RESTORE REG. B
00C1 C9                1140       RET             AND WE'RE DONE!
00C2                   1145       DS 4            HERE'S MORE STORAGE AREA FOR THOSE
00C6                   1150       DS 4            LONGER OUTPUT ROUTINES...
00CA                   1155       DS 4            THE CHAR. TO BE PRINTED IS IN REG. A
00CE                   1160       DS 4            AND THE ROUTINE MUST END WITH A "RET."
00D2                   1165       DS 4            ALL REGISTERS MUST BE UNALTERED UPON
00D6                   1170       DS 5            RETURN, WITH THE EXCEPTION OF REG. A.
00DB                   1175 *
00DB                   1180 *
00DB                   1185 * IPSTN INCREMENTS THE PRINT POSITION COUNTER ONCE
00DB                   1190 * EACH TIME IT IS CALLED.
00DB                   1195 *
00DB 3A FF 0B          1200 IPSTN LDA PSTN        BUMP PRINT POSITION ONE COUNT
00DE 3C                1205       INR A
00DF 32 FF 0B          1210       STA PSTN
00E2 C9                1215       RET
00E3                   1220 *
00E3                   1225 *
00E3                   1230 * SPOUT OUTPUTS EITHER A SPACE OR A CR/LF, DEPENDING
00E3                   1235 * ON IF WE ARE GETTING CLOSE TO THE END OF A LINE.
00E3                   1240 * IN THIS WAY WORDS ARE NOT BROKEN BETWEEN LINES
00E3                   1245 * OF OUTPUT.
00E3                   1250 *
00E3 3A FF 0B          1255 SPOUT LDA PSTN        GET PRINT POSITION
00E6 FE 36             1260       CPI LEN-10      ARE WE NEAR END OF LINE?
00E8 DA EF 00          1265       JC PRSP         NO: CONTINUE ON SAME LINE
00EB CD AC 00          1270       CALL CRLF       YES: START A NEW LINE
00EE C9                1275       RET
00EF 3E 20             1280 PRSP  MVI A,' '       PRINT A SPACE
00F1 CD BB 00          1285       CALL OUT
00F4 CD DB 00          1290       CALL IPSTN      UPDATE PRINT POSITION
00F7 C9                1295       RET
00F8                   1300 *
00F8                   1305 *
00F8                   1310 * MKLEN MEASURES THE LENGTH OF A MARK. THE LENGTH
00F8                   1315 * IS RETURNED IN REG. H, AND SHOULD NEVER BE
00F8                   1320 * ALLOWED TO BE GREATER THAN 7FH (SEE DLY1).
00F8                   1325 * REG. H IS INCREMENTED BY ONE IF A MARK IS NOT
00F8                   1330 * PRESENT (SPACE) WHEN THIS ROUTINE IS CALLED.
00F8                   1335 *
00F8 24                1340 MKLEN INR H           BUMP MARK LENGTH COUNT
00F9 CD 9D 00          1345       CALL DLY1
00FC DB FC             1350       IN CW
00FE E6 01             1355       ANI MASK        END OF MARK?
0100 C2 F8 00          1360       JNZ MKLEN       NO: BUMP COUNT AGAIN
0103 C9                1365       RET             YES: RETURN WITH MARK LENGTH
0104                   1370 *
0104                   1375 *
0104                   1380 * VLDMK CHECKS THE VALIDITY OF A MARK. ANY MARK LENGTH
0104                   1385 * LESS THAN 6 IS CONSIDERED NOISE, THE NOISE LENGTH
0104                   1390 * BEING ADDED TO THE PRESENT SPACE LENGTH BEFORE RETURN.
0104                   1395 * OTHERWISE THE VALID MARK LENGTH IS RETURNED IN REG. H.
0104                   1400 * THE CY FLAG IS CLEARED UPON RETURN IF A VALID MARK
0104                   1405 * WAS RECEIVED; OTHERWISE IT IS SET.
0104                   1410 *
0104 E5                1415 VLDMK PUSH H          SAVE SPACE LENGTH TEMPORARILY
0105 26 00             1420       MVI H,00        SET UP TEMPORARY MARK COUNTER
0107 CD F8 00          1425       CALL MKLEN      MEASURE MARK LENGTH
010A 7C                1430       MOV A,H
010B FE 06             1435       CPI 06          VALID MARK? (BANDWIDTH IS SET HERE)
010D DA 13 01          1440       JC SPACE        NO: ADD MARK LENGTH TO SPACE LENGTH
0110 33                1445       INX SP          YES: NEW MARK; DON'T WANT OLD
0111 33                1450       INX SP          SPACE LENGTH ON STACK
0112 C9                1455       RET             RETURN WITH NEW MARK LENGTH AND CY = 0
0113 7C                1460 SPACE MOV A,H         ADD INVALID MARK LENGTH TO
0114 E1                1465       POP H           SPACE LENGTH
0115 84                1470       ADD H
0116 67                1475       MOV H,A         REG. H HAS ADJUSTED SPACE LENGTH
```

```
0117 37              1480        STC          SET INVALID MARK FLAG (CY = 1)
0118 C9              1485        RET          AND RETURN WITH NEW SPACE LENGTH
0119                 1490 *
0119                 1495 *
0119 05              1500 TABLE  DB 05H
011A 41              1505        DB 'A'
011B 16              1510        DB 16H
011C 42              1515        DB 'B'
011D 14              1520        DB 14H
011E 43              1525        DB 'C'
011F 0A              1530        DB 0AH
0120 44              1535        DB 'D'
0121 02              1540        DB 02H
0122 45              1545        DB 'E'
0123 1C              1550        DB 1CH
0124 46              1555        DB 'F'
0125 08              1560        DB 08H
0126 47              1565        DB 'G'
0127 1E              1570        DB 1EH
0128 48              1575        DB 'H'
0129 06              1580        DB 06H
012A 49              1585        DB 'I'
012B 17              1590        DB 17H
012C 4A              1595        DB 'J'
012D 09              1600        DB 09H
012E 4B              1605        DB 'K'
012F 1A              1610        DB 1AH
0130 4C              1615        DB 'L'
0131 03              1620        DB 03H
0132 4D              1625        DB 'M'
0133 04              1630        DB 04H
0134 4E              1635        DB 'N'
0135 07              1640        DB 07H
0136 4F              1645        DB 'O'
0137 18              1650        DB 18H
0138 50              1655        DB 'P'
0139 11              1660        DB 11H
013A 51              1665        DB 'Q'
013B 0C              1670        DB 0CH
013C 52              1675        DB 'R'
013D 0E              1680        DB 0EH
013E 53              1685        DB 'S'
013F 01              1690        DB 01H
0140 54              1695        DB 'T'
0141 0D              1700        DB 0DH
0142 55              1705        DB 'U'
0143 1D              1710        DB 1DH
0144 56              1715        DB 'V'
0145 0B              1720        DB 0BH
0146 57              1725        DB 'W'
0147 15              1730        DB 15H
0148 58              1735        DB 'X'
0149 13              1740        DB 13H
014A 59              1745        DB 'Y'
014B 12              1750        DB 12H
014C 5A              1755        DB 'Z'
014D 1F              1760        DB 1FH
014E 30              1765        DB '0'
014F 2F              1770        DB 2FH
0150 31              1775        DB '1'
0151 37              1780        DB 37H
0152 32              1785        DB '2'
0153 3B              1790        DB 3BH
0154 33              1795        DB '3'
0155 3D              1800        DB 3DH
0156 34              1805        DB '4'
0157 3E              1810        DB 3EH
0158 35              1815        DB '5'
0159 2E              1820        DB 2EH
015A 36              1825        DB '6'
015B 26              1830        DB 26H
015C 37              1835        DB '7'
015D 22              1840        DB 22H
015E 38              1845        DB '8'
015F 20              1850        DB 20H
0160 39              1855        DB '9'
0161 51              1860        DB 51H
0162 28              1865        DB '('
0163 72              1870        DB 72H
0164 3F              1875        DB '?'
0165 2C              1880        DB 2CH
0166 2F              1885        DB '/'
0167 69              1890        DB 69H
0168 2E              1895        DB '.'
0169 4B              1900        DB 4BH
016A 2C              1905        DB ','
016B 46              1910        DB 46H
016C 3A              1915        DB ':'
016D 54              1920        DB 54H
016E 3B              1925        DB ';'
016F 2D              1930        DB 2DH
0170 2D              1935        DB '-'
0171 FF              1940        DB 0FFH      ERROR
0172 5C              1945        DB '\'
0173 34              1950        DB 34H       END OF MESSAGE
0174 3C              1955        DB '<'
0175 28              1960        DB 28H       EXCLUSIVE INVITATION TO TRANSMIT
0176 23              1965        DB '#'
0177 36              1970        DB 36H       WAIT
0178 25              1975        DB '%'
0179 79              1980        DB 79H       END OF WORK
017A 3E              1985        DB '>'
017B 00              1990        DB 00        END-OF-TABLE FLAG

ASCII   0094    0830
CONVT   007A    0435
CRLF    00AC    0335 0455 1270
CW      00FC    1350
DASH    0052    0740
DLY1    009D    1345
DOT     005F    0595
DOTT    0066    0745
ERROR   0098    0850
GENSP   0036    0500
IPSTN   00DB    0445 1290
LEN     0040    0450 1260
LOOP1   00A2    1010 1020
MASK    0001    1355
MKLEN   00F8    1360 1425
MKTYP   004B    0390
NEWMK   0016    0405 0465
NEXT    0084    0870
OUT     00BB    0440 1060 1070 1285
PRSP    00EF    1265
PSTN    0BFF    1080 1200 1210 1255
SAME    0072    0675
SPACE   0113    1440
SPOUT   00E3    0505
SPTYP   001B    0430
TABLE   0119    0820
VLDMK   0104    0380 0400 0460
WAIT    0010    0385 0510
```


BY FRED BLECHMAN
AND DAVID McDONALD

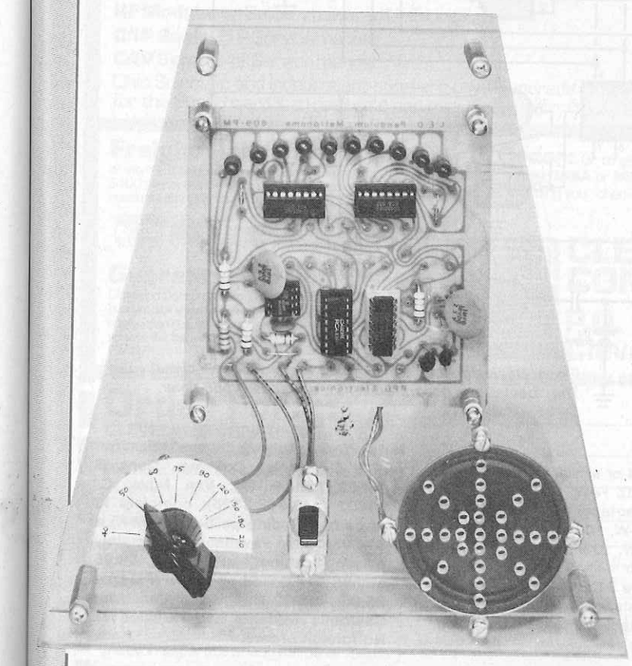
A LED PENDULUM METRONOME

An electronic replacement for the old mechanical music timer

The mechanical metronome, reputedly invented by Maezel in the 19th century, has been a familiar sight around musicians and music students for many years. It uses a wind-up clock mechanism to swing a weighted arm, generating a series of clicks as the escapement gears make contact. The clicking rate is conventionally adjustable from 40 to 210 beats per minute by positioning the weight on the calibrated oscillating arm to change the moment of inertia and the rate of the swing.

Redoubtable though it may be, Maezel's brainchild suffers from defects common to all mechanical devices: wear, drift of calibration, and the need for fairly frequent maintenance. In addition, it must be wound often. A battery-operated, solid-state electronic design, such as the LED Pendulum Metronome described here, circumvents or alleviates the problems of the mechanical metronome. It is stable in calibration and reliable.

Partly for nostalgic reasons, the pendulum movement of the mechanical metronome is simulated in the project as a flashing sequence of LEDs arranged in an arc. (A click from a loudspeaker occurs as the LED at either end of the string fires.) However, the LEDs offer the user the option of "reading" the metronome signal visually in circumstances where a click might be inaudible or objectionable to the user.

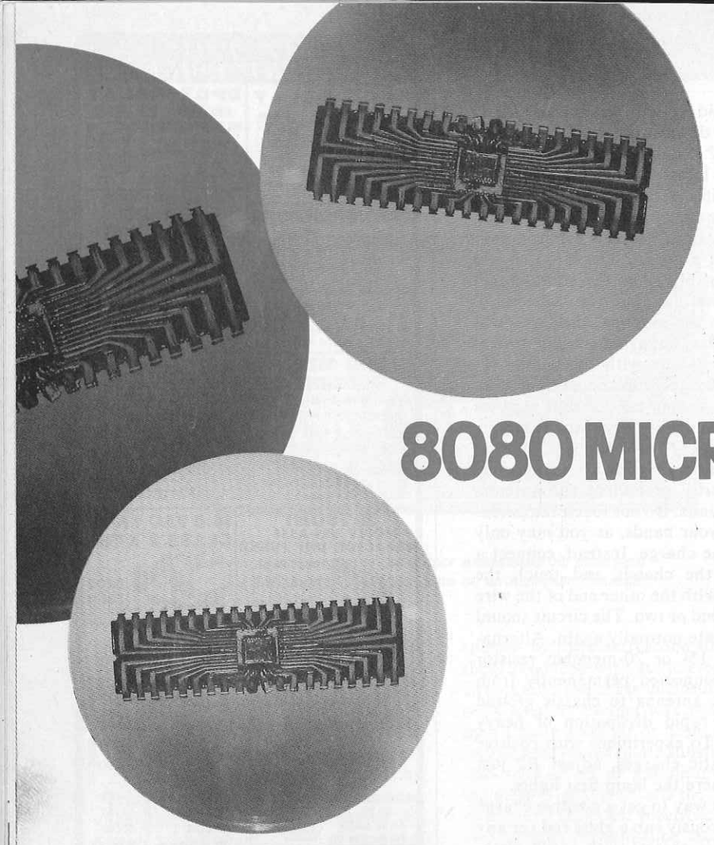


Circuit Operation. The "beats" are generated by *IC1*, which is used as an oscillator (Fig. 1). Resistors *R1*, *R3*, and capacitor *C1* limit the frequency of operation that can be set by means of potentiometer *R2*. Capacitor *C2* decouples the *IC1* modulation input. Each cycle of operation of *IC1* results in a positive-going pulse at pin 3, which is fed to the clock input of up-down counter *IC4*. This counter can be set to count from 0 to 9 (10 counts) or 0 to 15 (16 counts), depending on the status of pin 9. With pin 9 positive (as shown), *IC4* counts from 0 to 15. Counting up or down is controlled by pin 10; positive for up-counting, ground for down-counting. The A, B, C, and D outputs of *IC4* (pins 6, 11, 14, and 2) go positive in a 4-bit binary sequence with the D output (pin 2) low during counts 0 to 7 and high during counts 8 to 15.

Both *IC2* and *IC3* are identical 1-of-8 switches. Depending on the 3-bit binary input, one of eight outputs is connected to pin 3 through a low resistance (typically 120 ohms). This is called the "on" condition for this pin. The A, B, and C inputs to *IC2* and *IC3* (pins 11, 10, and 9) are addressed by the output pins (6, 11 and 14, respectively) of *IC4*. However, *IC2* or *IC3* must be enabled by a low on pin 6. For counts 0 to 7, pin 2 of *IC4* is low, enabling *IC2*. Notice, however, that pin 6 of *IC3* is high because of *IC5A*, one section of a quad 2-input NAND gate, wired as an inverter. This disables *IC3* while *IC2* is enabled.

As *IC4* counts from 0 to 7, the outputs of *IC2* are turned "on" in sequence. In this case, on is *not* ground, but is an internal low resistance to pin 3, which is grounded. This low resistance to ground allows the LED connected to an on pin to glow.

Only five LEDs are connected to the eight outputs, with *LED1* connected to three outputs, *LED2* to two outputs, and *LED3*, *LED4*, and *LED5* to one output each. This is done to simulate the swinging motion of a pendu-



BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 5: Morse Code Hardware Interface

THE interface required for the Morse program described in Part 4 of this series consists of one parallel input port and one parallel output port, as shown in Fig. 23. The Morse program was originally written for a system which incorporated a printer or CRT as the output medium. If a printer or CRT is not available, the output display shown in Fig. 24 may be used in conjunction with the necessary program changes given in Table I.

In Fig. 23, *IC2* and *IC3* constitute the Port-Select logic, which decodes I/O port FC. Pin 1 of *IC3B* responds to an IN FCH instruction by going high; pin 4 of *IC3A* goes high in response to an OUT FCH instruction. *IC1* latches the output data during an OUT FCH instruction, the output of which may be connected to a printer, CRT, or the single-character display shown in Fig. 24. *IC4* performs the function of buffering and gating the input data byte onto the CPU Data Bus during an IN FCH instruction. Only bit 0 (pin 2 of *IC4*) is used by the Morse program; the remaining input bits may be used in other applications if desired. *IC5* functions as a form of A/D converter. It has one analog input which accepts audio voltages (such as from a radio receiver's speaker) and one TTL-compatible output (pin 5). The output goes low whenever audio of sufficient amplitude (from a received dot or dash) is present at the input, and is high in the absence of audio (spaces).

Turning our attention now to Fig. 24, we find that *IC7* converts the ASCII code latched in *IC1* (which was output by the Morse program) into a multiplexed 5-bit code necessary for driving the alphanumeric display *DIS1*. *IC6*, *IC10*, *IC11*, and buffers *IC8* and *IC9* complete the interface to *DIS1*. Bit 7 of the output latch (the unused parity bit) is used to turn the display on or off; setting this bit to 1 enables the display.

Installation and adjustment of the interface is straightforward. Connect *J1*, *J2*, and *J3* of the interface to *P1*, *P2*, and *P3* of the CPU module using three 16-conductor ribbon cables, and the audio input of the interface to your receiver's speaker. Install the ROM containing the Morse program machine code in the *IC5* socket of the CPU module.

Apply power and adjust the receiver volume to a comfortable listening level. Then tune the receiver to a spot where no signals are present and adjust sensitivity control *R1* until *LED1* lights. Now back off *R1* just past the point where *LED1* extinguishes. This is the point of maximum sensitivity of the detector,

TABLE I—SUBSTITUTE SUBROUTINE

```
0000                    0100 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
0000                    0105 *
0000                    0110 * THIS IS A SUBROUTINE WHICH MAY BE USED IN PLACE OF
0000                    0115 * THE ORIGINAL "OUT" SUBROUTINE IN THE MORSE PROGRAM.
0000                    0120 * IT WAS WRITTEN PRIMARILY FOR USE WITH THE SINGLE-
0000                    0125 * CHARACTER DISPLAY DESCRIBED ELSEWHERE IN THIS ARTICLE.
0000                    0130 * THE FIRST LETTER OF EACH RECEIVED WORD IS OUTPUT IN
0000                    0135 * UPPER-CASE, WHEREAS THE REMAINING LETTERS OF THE WORD
0000                    0140 * ARE OUTPUT IN LOWER-CASE. THIS MAKES IT EASIER TO
0000                    0145 * IDENTIFY WORD SPACES WHEN USING THE SINGLE-CHARACTER
0000                    0150 * DISPLAY.
0000                    0155 *
0000                    0160 *
0000                    0165        ORG OUT     BEGIN ASSEMBLY AT ORIGINAL "OUT" ADDR
00BB                    0170 *
00BB C5                 0175 OUT    PUSH B      WANT TO RETURN WITH REG. B UNALTERED
00BC FE 41              0180        CPI 'A'     MODIFY ONLY ALPHABETIC CHARACTERS
00BE DA D1 00           0185        JC DSPLY
00C1 FE 5B              0190        CPI 'Z'+1
00C3 D2 D1 00           0195        JNC DSPLY
00C6 47                 0200        MOV B,A     SAVE ACCUMULATOR TEMPORARILY
00C7 3A FF 0B           0205        LDA PSTN    PUT WORD FLAG INTO ACCUMULATOR
00CA 1F                 0210        RAR         ROTATE FLAG BIT INTO CARRY BIT FOR
00CB 78                 0215        MOV A,B     TESTING AND RESTORE ACCUMULATOR
00CC DA D1 00           0220        JC DSPLY    SEND LETTER AS UPPER-CASE IF NEW WORD
00CF F6 20              0225        ORI 20H     ELSE CONVERT TO LOWER-CASE
00D1 F6 80              0230 DSPLY  ORI 80H     SET DISPLAY ENABLE BIT
00D3 D3 FC              0235        OUT CW      SEND CHARACTER TO LATCH
00D5 AF                 0240        XRA A
00D6 32 FF 0B           0245        STA PSTN    CLEAR WORD FLAG
00D9 C1                 0250        POP B       RESTORE REG. B TO THE WAY IT WAS PRIOR
00DA C9                 0255        RET         TO ENTERING THIS SUBROUTINE AND RETURN
00DB                    0260 *
00DB                    0265 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
00DB                    0270 *
00DB                    0275 * IF THE ABOVE ROUTINE IS USED IN PLACE OF THE ORIGINAL
00DB                    0280 * "OUT" ROUTINE IN THE MORSE PROGRAM, THE FOLLOWING
00DB                    0285 * CHANGES MUST ALSO BE MADE:
00DB                    0290 *
00DB                    0295 *
00DB                    0300        ORG CRLF
00AC                    0305 *
00AC C9                 0310 CRLF   RET         SUPRESS ALL CAR. RET.'S AND LINE FEEDS
00AD                    0315 *
00AD                    0320 *
00AD                    0325        ORG IPSTN
00DB                    0330 *
00DB C9                 0335 IPSTN RET          DON'T UPDATE PRINTER POSITION COUNTER
00DC                    0340 *
00DC                    0345 *
00DC                    0350        ORG SPOUT
00E3                    0355 *
00E3 3E 01              0360 SPOUT MVI A,01     SET WORD FLAG RATHER THAN SENDING
00E5 32 FF 0B           0365       STA PSTN     A SPACE, THEN RETURN
00E8 C9                 0370       RET
00E9                    0375 *
00E9                    0380 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

CRLF    00AC    0300
DSPLY   00D1    0185 0195 0220
IPSTN   00DB    0325
OUT     00BB    0165
SPOUT   00E3    0350
```

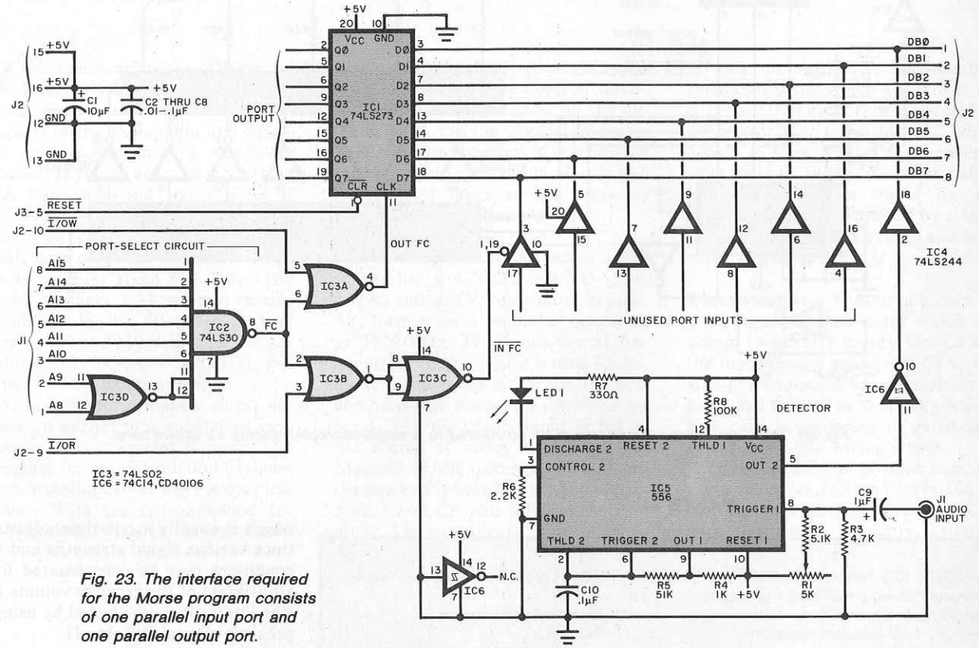



Fig. 23. The interface required for the Morse program consists of one parallel input port and one parallel output port.

TABLE II—MORSE INTERFACE TEST PROGRAM

0000	31 FF 0B	LXI SP, OBFFH	(Initialize Stack Pointer to end of RAM area)
0003	CD 00 01	LOOP CALL TEST	(Call the test subroutine)
0006	C3 03 00	JMP LOOP	(Do it again)
0100	3E 41	TEST MVI A,41H	(Load accumulator with the ASCII (character code for "A")
0102	F6 80	ORI 80H	(Set display-enable bit)
0104	D3 FC	OUT FCH	(Sent data byte to the port)
0106	DB FC	IN FCH	(Read the port too)
0108	C9	RET	(and return to main program)

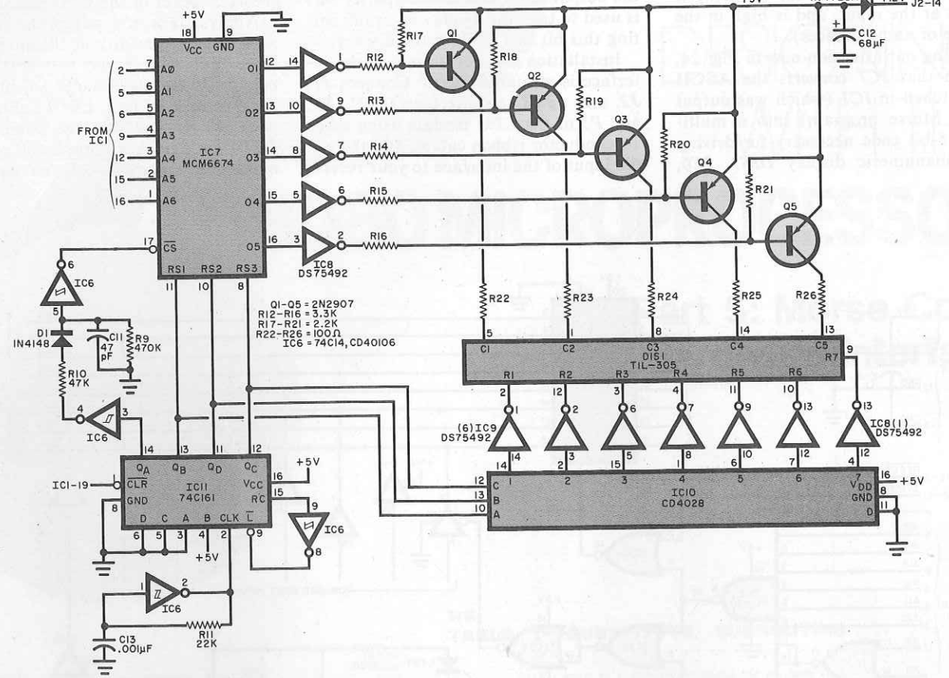


Fig. 24. The output of the interface can be connected to a single-character display as shown here.

PARTS LIST

C1—10-µF, 10-V tantalum capacitor
C2 through C8—0.01-µF or 0.1-µF capacitor distributed near ICs
C9—1-µF, 10-V tantalum capacitor
C10—0.1-µF disc ceramic capacitor
C11—47-pF disc ceramic capacitor
C12—68-µF, 15-V tantalum capacitor
C13—0.001-µF disc ceramic capacitor
D1—1N4148 switching diode
D2—1N4728A 3.3-V Zener diode
DIS1—TIL-305 5×7 alphanumeric LED display
IC1—74LS273 octal D-flip-flop
IC2—74LS30 8-input NAND gate
IC3—74LS02 quad 2-input NOR gates
IC4—74LS244 octal noninverting tristate buffers/receivers
IC5—LM556C dual timer
IC6—74C14 or CD40106 hex Schmitt-trigger inverters
IC7—MCM6674 5×7 character generator (Motorola)
IC8, IC9—DS75492 MOS-to-LED hex digit drivers
IC10—CD4028 BCD-to-Decimal decoder
IC11—74C161 or CD40161 binary counter
J1, J2, J3—16-pin DIP socket
LED1—Red LED
Q1 through Q5—2N2907 or PN2907 transistor
Unless otherwise specified, the following are 1/4-watt, 10%-tolerance, fixed carbon-composition resistors:
R1—5-kΩ, PC-mount potentiometer
R2—5.1 kΩ
R3—4.7 kΩ
R4—1 kΩ
R5—51 kΩ
R6, R17 through R21—2.2 kΩ
R7—330 Ω
R8—100 kΩ
R9—470 kΩ
R10—47 kΩ
R11—22 kΩ
R12 through R16—3.3 kΩ
R22 through R26—100 Ω, 1/2-W
Misc.—IC sockets, Vector board or printed-circuit board, wire-wrap wire or solder, etc.

which is usually a one-time adjustment since various signal strengths and noise conditions may be compensated for by adjustment of the receiver volume level. The interface can be tested by using the program shown in Table II.

The Morse program is now operational. In crowded band conditions, it is especially important that the receiver have adequate selectivity, or the Morse program will not know which signal to lock on to. Code speed variations are automatically tracked and compensated for by the program.

The Morse program may also be used in conjunction with a code-practice oscillator for code practice or troubleshooting of the interface. It has also proven to be a very effective aid in learning the Morse code since each Morse character may be seen immediately after it is heard, making it easier to associate the Morse "sounds" with the characters they represent.

Next month we will discuss programming the CPU ROM. ◇

A SIMPLE **SHORTWAVE CONVERTER** FOR ANY AM RADIO

Inexpensive device enables AM radios to receive shortwave broadcasts

BY JEFF HIRSCHL

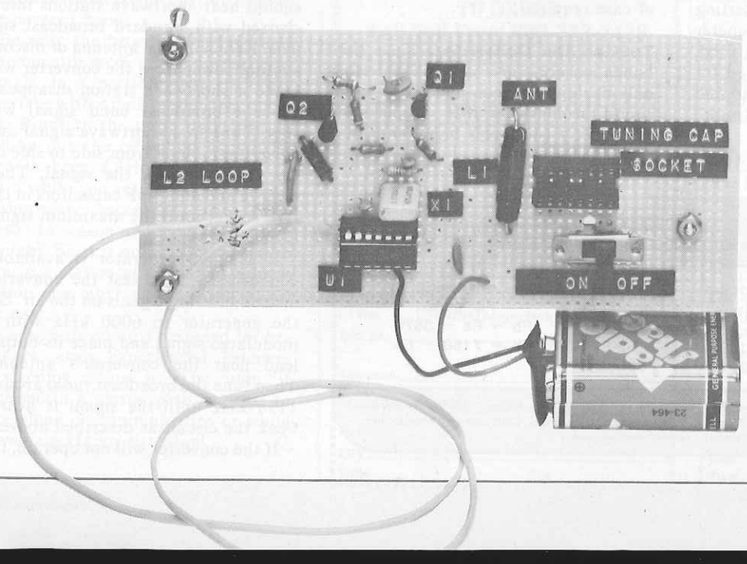
YOU can hear dozens of powerful English-language broadcasts offering news, music, and drama from all parts of the globe night and day—but only if you have a shortwave receiver. If you've never been involved with shortwave and want to see if you'd like to pursue this hobby seriously, without a significant investment, here's a little converter which can be built for about $13. It lets you use an ordinary AM radio to receive broadcasts in the 60-meter tropical band (4750 to 5060 kHz) and the 49-meter band (5950 to 6200 kHz), two of the 11 SW bands available.

Although performance does not stand up to that of a good shortwave receiver, this converter is more than adequate for an introduction to shortwave listening and at a great deal less money. With the recommended 10-foot antenna, signals from *Radio Nederland*, the *BBC*, *Radio Canada International*, and the *Voice of America* can be easily received on the 49-meter band. On the tropical 60-meter band, so called because of the location of the stations that use it, signals can be received from as far away as Colombia and Venezuela.

About the Circuit. As shown in the schematic, a CMOS NAND gate, *IC1A*, and a TV color-burst crystal, *X1*, form a local oscillator operating at 3579 kHz. The fundamental frequency of this oscillator is used for 60-meter band reception, while the second harmonic is used for the 49-meter band. The oscillator signal is fed to the source of mixer transistor, *Q1*. Meanwhile, the incoming signal from the antenna is tuned by plug-in capacitors, *C1* to *C5*, and is fed to the gate of *Q1*. The two signals "mix" in *Q1* to provide an output in the standard broadcast band, which appears at the drain of *Q1*. This output is coupled by *C6* to amplifier transistor, *Q2*, which boosts it to a level and impedance suitable to drive the broadcast radio's loop antenna. The signal for the broadcast radio is provided by a loop, *L2*, wound around the radio and driven by the emitter of *Q2*.

Construction. The circuit may be built on any circuit board which can accept 14-pin DIP sockets for *IC1* and the input tuning capacitors *C1* to *C5*. Use point-to-point wiring and try to keep lead lengths as short as possible. Use care in soldering to avoid cold solder joints and wiring errors.

To reduce the risk of static damage, use a socket at *IC1* and leave the IC out during assembly. Be careful to wire this socket correctly, avoiding



Popular Electronics®

WORLD'S LARGEST SELLING ELECTRONICS MAGAZINE FEBRUARY 1982/$1

New Single-IC Video Modulator

FOR CRISPER TV COLOR FROM COMPUTERS & GAMES

Evaluating the Xerox 820 Personal Computer

Elapsed-Time Device Logs TV Use Automatically

Buyer's Guide to Telephone Controllers

● STORED MEMORY ● AUTOMATIC REDIALING ● CALL TIMING

Tested in this Issue:

13" Color TV

ument Modules

0 02 14024 14278

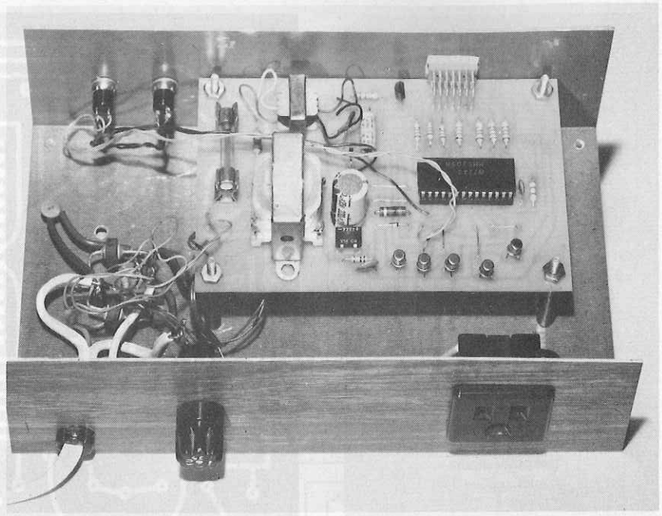


Photo of the internal arrangement of the author's prototype.

on and verify that the recorder starts timing. To find the cumulative time-on for a period longer than a day, note the recorder's display daily and then reset the recorder. At the end of the time period, add the results for each day. (The maximum display is 24 hours before the clock resets itself to zero.)

Please note that the recorder's maximum load is 720 watts (i.e., a load requiring no more than 6 amperes).

Fig. 5. An RC shunt on the triac improves operation with inductive loads.

Some loads, such as a dehumidifier may be rated at 5 A but when first turned on will draw in excess of 6 A. This will cause fuse *F1* to blow. In this case, the time-on recorder cannot be used. Also, if the ac line has noise spikes on it (from appliances such as a dehumidifier) the recorder may be accidentally reset and lose its count. To remedy this add capacitors *C5* and *C6* as shown in the schematic. These components may also be needed if the load is a fan. These capacitors short the noise spikes on the ac line to ground. Note also that there is a minimum load requirement to start the recorder timing. A load as small as 10 watts will activate the recorder, while a load of 7½ watts will not.

Although the recorder can be used to tell you how much television the family is watching or the like, it can also tell you how much the television is costing you to use. To find the cost, take the total time in hours, multiply it by the power rating of the load in kilowatts and then multiply by the cost of a kilowatt-hour in your area.

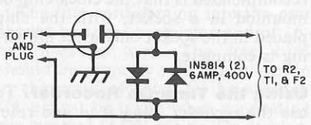
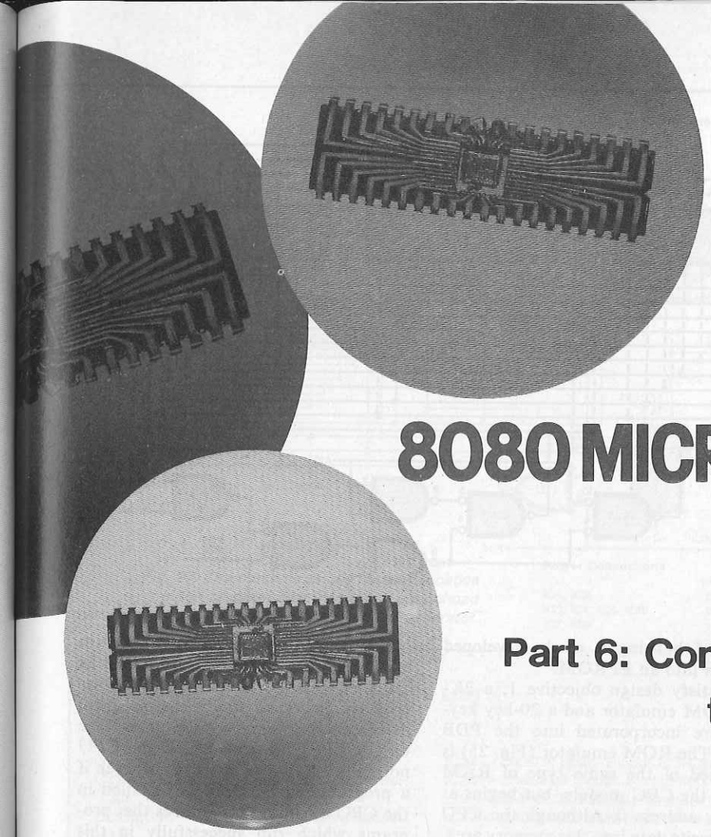


Fig. 6. Two diodes can be used to replace the triac if desired.

Going Further. The circuit shown in Fig. 1 uses a triac to produce the signal that enables the clock chip. The triac was selected because of its easy availability. It does, however, restrict the types of loads that can be timed. That is, the triac will prevent large inductive loads such as fans and other motors from starting.

If you want to avoid this, the circuit can be changed. Shunting the triac with a series RC circuit as in Fig. 5 would eliminate the problem mentioned with inductive loads. This solution, however, requires two additional components and creates the problem of finding the values of the two components. Therefore, the circuit shown in Fig. 6 is preferable. If the diodes are available, use the circuit to replace the part of the original circuit which uses the triac. No other change in the circuit is necessary.

If one of the diodes in Fig. 6 fails, fuse *F2* will blow to protect *T1*. In this case, the bad diode and *F2* will have to be replaced before the circuit will operate. This type of circuit failure should be kept in mind if *F2* blows but there are no wiring errors present. ◇



BY RANDY CARLSTROM

DESIGNING WITH THE 8080 MICROPROCESSOR

Part 6: Conclusion—Programming the CPU Module's ROM

HAVING designed and built the interface for receiving Morse code, into the CPU, it is now necessary to program the CPU program memory.

There are several types of read-only memory (ROM), each of which has its unique way of being programmed. One type is the *mask programmable* ROM in which the desired binary state of each memory cell (bit) is programmed by selectively including or excluding a small conducting jumper in the cell during manufacture. This type of ROM programming is permanent and the bit pattern cannot be altered once programmed. A change of even one bit requires that a new custom mask be made, which is a relatively expensive process (about $1,000). This type of ROM is generally used only in high-volume production applications where the desired bit pattern has already been proven and the probability of pattern changes or updates is very unlikely.

A second type of ROM is the *Programmable Read-Only Memory* (PROM). This device is similar to the mask programmable ROM, but has the advantage of being *field programmable*—that is, the customer can program it himself. This is done by selectively "blowing" fusible links (made of polycrystalline silicon or nichrome) in each memory cell with a relatively high-current pulse to obtain the desired bit pattern. Needless to say, the PROM must be discarded and a new one programmed if any bit changes are to be made which would otherwise require the repair of a blown fuse link. The PROM shares the same disadvantage of the mask programmable ROM—it is not reprogrammable.

The last type of ROM we will examine is the *Erasable and Programmable Read-Only Memory* (EPROM). As its name suggests, the EPROM can be programmed by the user. However, rather than "blowing" fusible links as in the PROM, a small electric charge is selectively deposited in each memory cell. The EPROM has the property that when its chip surface it exposed to ultraviolet light, any charges deposited in the memory cells are removed, which completely "erases" the memory. This unique property of the EPROM gives it the added feature of being reprogrammable. The EPROM chip is covered by a transparent quartz lid (rather than the conventional opaque metal or plastic cover) which allows it to be exposed to ultraviolet light. The erasing process (exposure to UV light) normally takes from 20 to 30 minutes, and will set all of the EPROM's memory cells to a logic 1. Programming a cell to a logic 0 is done electrically by depositing a small electric charge in that cell. However, the only way a programmed cell can be changed from the 0 state to the 1 state is by erasing the *entire* device.

The EPROM is generally used in prototype systems, where bit pattern changes are likely to be made. Because of the distinct advantages the EPROM offers, it is this type which is used for the CPU module's program memory.

Program Development Board Design. In a 2K EPROM, there are over 16,000 memory bits to be programmed (depending on the length of the user's program), and there are strict timing requirements that must be adhered to when programming. To solve these problems the Program Development Board (PDB) was designed. Its objectives are:

(1) To provide a "program-development memory area" where 8080 programs can be conveniently stored and edited. This memory area should exist in the same address space as the CPU's EPROM, and the CPU must be able to execute any program stored in this area. In this way a new program can be loaded, tested, and debugged before it is copied ("burned") into an EPROM.

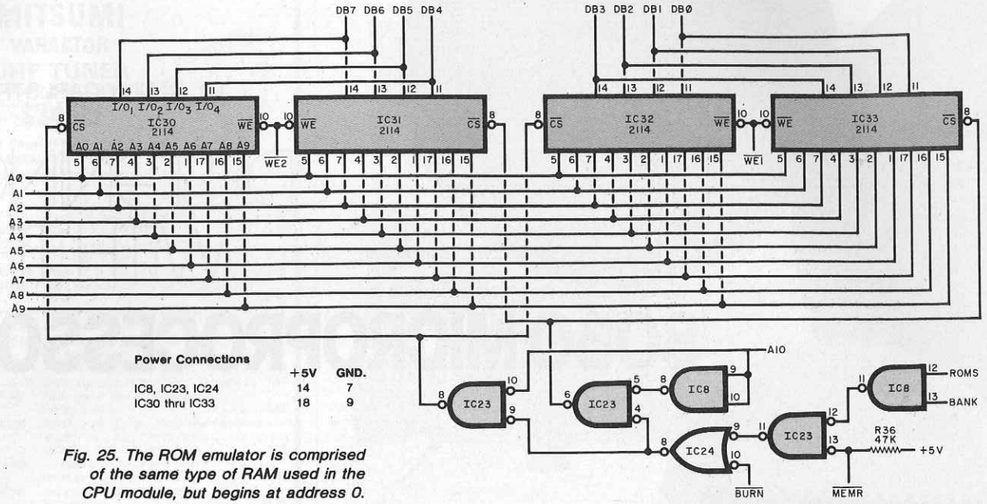


Fig. 25. The ROM emulator is comprised of the same type of RAM used in the CPU module, but begins at address 0.

(2) To monitor the status of the Address, Data, and Control busses at all times.

(3) To single-step the CPU one instruction at a time, thereby enabling the programmer to observe the results of each program instruction as it is executed. This provides an instructional tool as well as an aid in debugging new programs.

(4) To single-step and monitor the execution of a program already contained in the CPU's EPROM.

(5) To provide a fast and efficient means of burning a newly developed program into an EPROM.

To satisfy design objective 1, a 2K-byte ROM emulator and a 20-key keypad were incorporated into the PDB design. The ROM emulator (Fig. 25) is comprised of the same type of RAM used in the CPU module, but begins at memory address 0. Although the CPU cannot write data into this memory area, it can read program instructions and data previously stored there. This provides built-in protection to prevent a bad program from completely wiping itself out. It is only possible to begin program execution at memory location 0, just as the CPU module does when first powered-up. In these ways, the program development memory simulates the CPU's EPROM from which the CPU normally obtains its instructions, even if a programmed EPROM is installed in the CPU module. This ensures that programs which run successfully in this memory area will also run properly when they are transferred to the CPU's EPROM. Program instructions and data are loaded into the program development memory via the keypad (Fig. 26). It is also possible to examine the contents of any memory location in the program development or scratchpad memory via the keypad (Fig. 27).

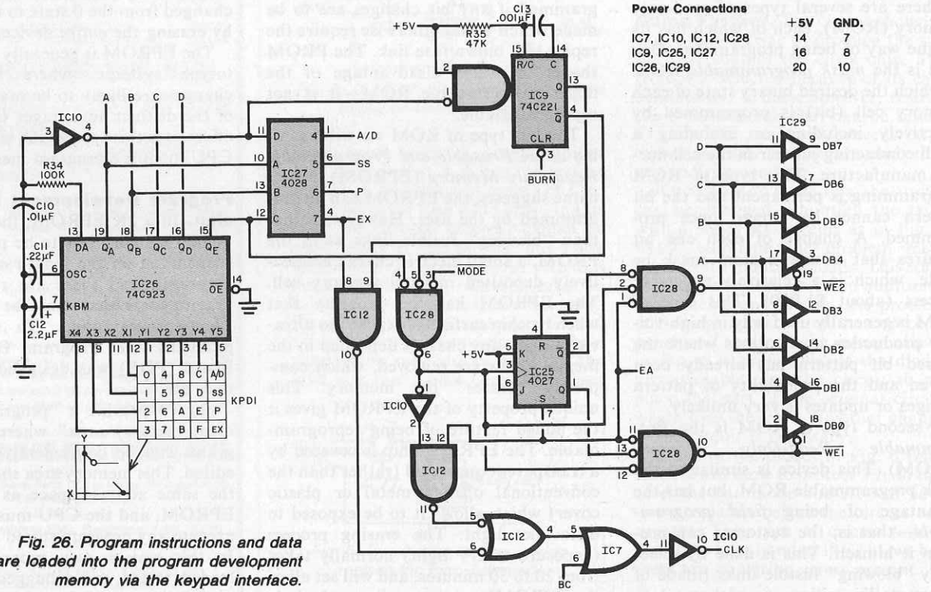


Fig. 26. Program instructions and data are loaded into the program development memory via the keypad interface.

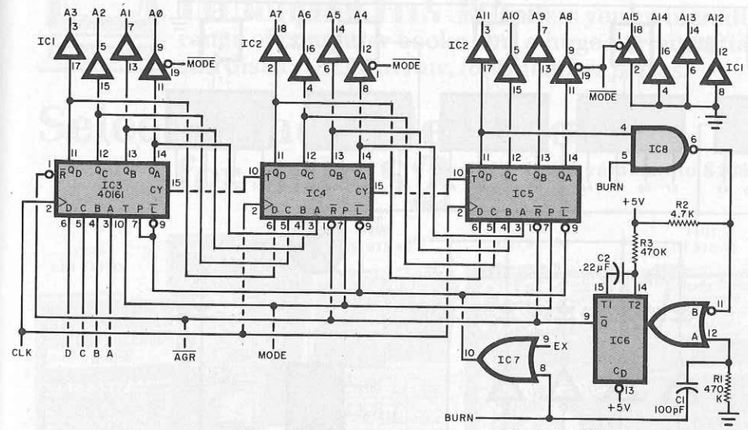


Fig. 27. The contents of any memory location in the program development or scratchpad memory can be examined using the address generator.

Power Connections

	+5V	GND.
IC1, IC2	20	10
IC3, IC4, IC5, IC6	16	8
IC7, IC8	14	7

PARTS LIST

C1—100-pF disc ceramic capacitor
C2,C9,C11—0.22-µF disc ceramic capacitor
C3,C8,C10—0.01-µF disc ceramic capacitor
C4—0.47-µF Mylar capacitor
C5,C13—0.001-µF disc ceramic capacitor
C6—47-pF disc ceramic capacitor
C7—0.33-µF, 50-V capacitor
C12—2.2-µF, 10-V tantalum capacitor
DIS1 through DIS6—Common-cathode, 7-segment LED display (H.P. 5082-7740, T.I. TIL313, or equivalent)
LED1 through LED7—Red light-emitting diode
IC1,IC2,IC29—MM74C244 octal noninverting tri-state buffer
IC3,IC4,IC5—CD40161BC or MM74C161 binary counter
IC6—CD4528BC dual monostable multivibrator
IC7—CD4071BC quad 2-input OR gate
IC8—74LS00 quad 2-input NAND gate
IC9—MM74C221 dual monostable multivibrator
IC10—CD40106BC or MM74C14 hex Schmitt trigger inverter
IC11—CD4013BC dual D flip-flop
IC12—CD4011BC quad 2-input NAND gate
IC13—CD4081BC quad 2-input AND gate
IC14—LM340LA-12 positive 12-volt regulator
IC15,IC16,IC17,IC18,IC19,IC20—MC14495 BCD-to-seven-segment decoder/driver
IC21—74LS74A dual D flip-flop
IC22—7417 or 7407 hex buffer with open-collecter outputs
IC23—74LS32 quad 2-input OR gate
IC24—74LS08 quad 2-input NOR gate
IC25—CD4027BC dual J-K flip-flop
IC26—MM74C923 20-key encoder
IC27—CD4028BC BCD-to-decimal decoder
IC28—CD4075BC triple 3-input OR gate
IC30 through IC33—2114L 1024x4 RAM
J1,J2,J3—16-pin DIP plug
P1,P2,P3—16-pin DIP plug
Q1,Q2,Q7—2N3904 or equivalent transistor
Q3—2N2907, PN2907, or equivalent transistor
Q4,Q5,Q6—2N2222, PN2222, or equivalent transistor
The following, unless otherwise specified, are ¼-watt, 10% fixed carbon-composition resistors:
R1,R3,R29—470 kΩ
R2,R8,R28,R33—4.7 kΩ
R4—330 kΩ
R5—50-kΩ pc-mount potentiometer
R6—82 kΩ
R7,R34—100 kΩ
R9—10 kΩ
R10,R13,R20,R27,R31,R32,R35,R36—47 kΩ
R11—22 kΩ
R12—1 kΩ
R14—1.2 kΩ
R15—560 Ω
R16—1-kΩ pc-mount potentiometer
R17,R18,R19,R21 through R26—330 Ω
R30—1 MΩ
S1 through S4—DIP switch
Misc.—0.01-µF or 0.1-µF disc ceramic bypass capacitors distributed near ICs; 4x5 X-Y matrix keypad or 20 spst NO momentary-contact pushbutton switches; IC sockets; perf or printed-circuit board; wire or solder, etc.

opment memory via the keypad (Fig. 26). It is also possible to examine the contents of any memory location in the program development or scratchpad memory via the keypad (Fig. 27).

A four-digit hexadecimal display monitoring the CPU's Address Bus, a two-digit hexadecimal display monitoring the Data Bus, and four LED's monitoring four of the five Control Bus lines satisfy design objective 2 (Fig. 28).

Through the use of a special key of the keypad it is possible to single-step the CPU in one of two selectable modes (Fig. 29). The first of these modes causes the CPU to execute one instruction with each depression of this key. The second mode causes one 8080 machine cycle to be executed with each depression of the key. The states of the CPU busses are always displayed in these modes to aid in monitoring an executing program's progress.

To satisfy design objective 4, the PDB allows the programmer to select the "memory bank" the CPU will use to obtain its program instructions from; the program development memory or an installed CPU EPROM. The CPU scratchpad RAM area beginning at memory location 800_{16} is always accessible to a running program, regardless of the memory bank selected.

When the programmer is satisfied with the operation of his new program, the depression of a single key of the keypad will burn the entire contents of the program development memory into an erased EPROM installed in the CPU module in less than two minutes (Fig. 30). After a successful burn, the programmer can separate the CPU module and interface(s) from the PDB (Fig. 31) and connect the CPU module directly to the interface(s). The programming of the CPU module in this particular application is now complete, and so are the design objectives of the PDB.

Functions. There are two primary modes of operation of the PDB—edit and execute. The edit mode allows program instructions and data to be loaded, examined, and altered in the program development memory. The PDB enters the execute mode whenever the CPU is single-stepped or run. Here are the PDB functions:

0 through F (numeric): Used to enter hexadecimal memory addresses and load program instructions and data into the program development memory area. The destinations of the numbers entered with these keys are governed by the A/D key.

A/D (Enter Address/Load Data): Places the PDB in the edit mode and also determines where subsequent nu-

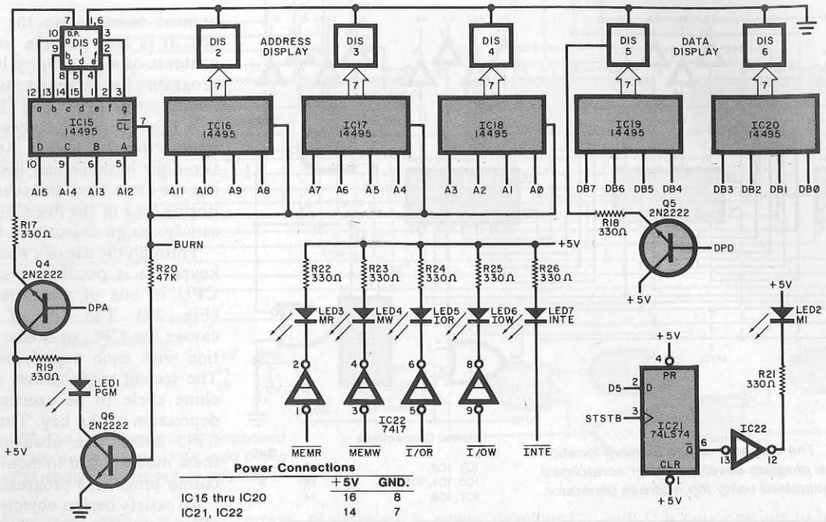


Fig. 28. The display logic drives a four-digit address display, two-digit data display, and four of the five LEDs on the control lines.

meric entries will be sent. Each key depression alternately lights a decimal point in the display corresponding to the destination of subsequent numeric entries (address or data). Actuating this key while a program is running will momentarily reset the CPU, restarting the running program.

SS (Single-Step): Places the PDB in the execute mode and allows the CPU to execute one instruction or one machine cycle (as selected by the instruction cycle switch) each time the key is actuated. A program can be restarted in the single-step mode at memory location 0 by depressing the A/D key followed by the SS key.

P (Program): Depressing this key will burn the contents of the program development memory into an erased EPROM installed in the CPU module. The keypad is disabled until the burn cycle has ended.

EX (Examine): Operates only when the PDB is in the edit mode. It allows the contents of the next sequential memory location (as displayed on the Address display) to be viewed on the Data display.

BANK: Determines which memory bank the CPU will use to obtain its instructions and data when single-stepped or run. Closing the switch will select the EPROM installed in the CPU module; otherwise the PDB's program development memory is selected.

INSTRUCTION CYCLE: Operates in conjunction with the SS key. Closing the switch causes the SS key to operate in the instruction-cycle mode; otherwise it operates in the machine-cycle mode.

SLOW: Also operates in conjunction with the SS key. Closing this switch

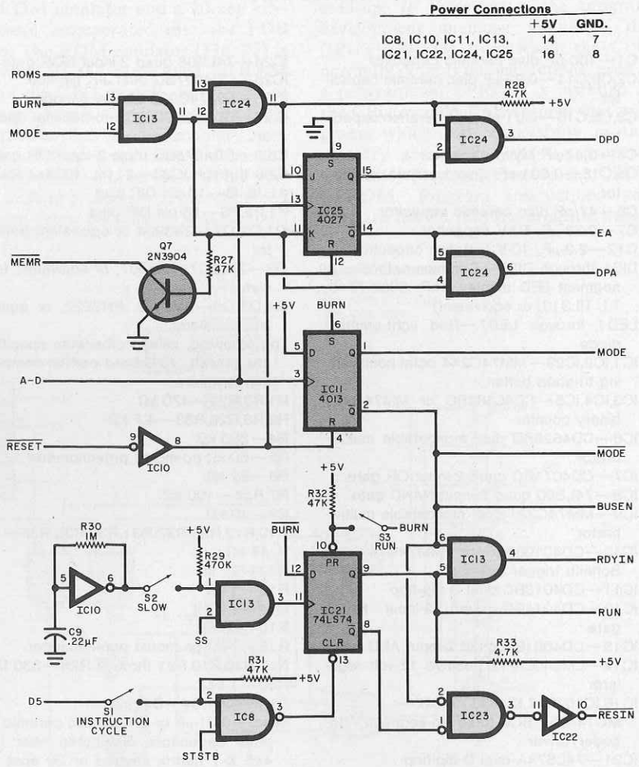


Fig. 29. The CPU can be single-stepped in one of the two selectable modes.

allows execution of instruction or machine cycles at a rate of approximately two cycles per second for the duration of the SS key's depression.

RUN: Places the PDB in the execute mode and allows the CPU to exit the Wait state, so that the program contained in the selected memory bank is executed at full speed (approximately 500,000 machine cycles per second). All keypad functions except A/D are disabled.

ADDRESS DISPLAY: A dual-function display of either the contents of the CPU Address Bus (execute mode) or the address of the addressed location of the program development and scratchpad memory areas (edit mode).

DATA DISPLAY: A dual-function display of either the contents of the CPU Data Bus (execute mode) or that of the addressed location of the program development and scratchpad memory areas (edit mode).

M1: Indicates the CPU is in machine-cycle one of an instruction cycle.

PGM: Indicates the PDB is in the process of burning an EPROM.

INTE: Indicates the CPU signal $\overline{\text{INTE}}$ is active.

The following LED's display the status of the control bus signals:

MR: Indicates the CPU signal $\overline{\text{MEMR}}$ is active.

MW: Indicates the CPU signal $\overline{\text{MEMW}}$ is active.

IOR: Indicates the CPU signal $\overline{\text{I/O R}}$ is active.

IOW: Indicates the CPU signal $\overline{\text{I/O W}}$ is active.

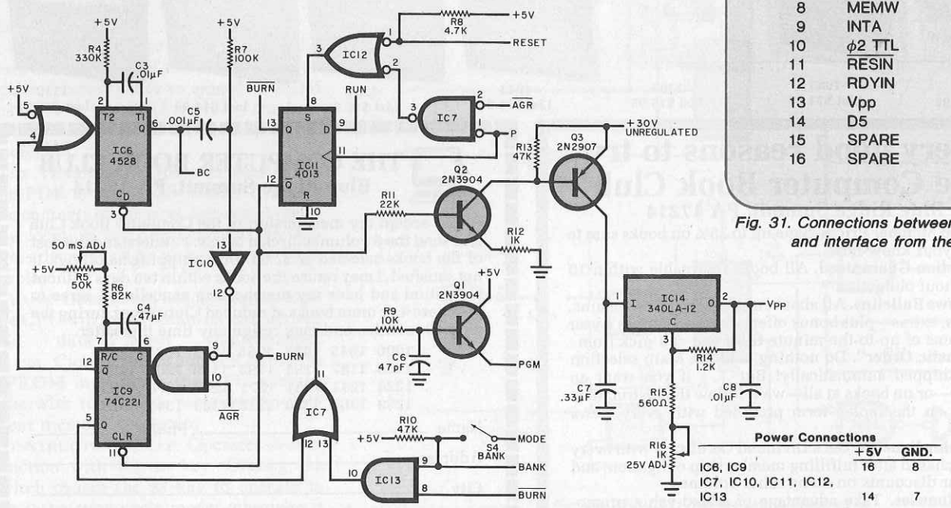
Power Supplies and Adjustments. The PDB can be powered from the CPU module's supplies (*J2*), but an unregulated +30-volt supply is also required for programming an EPROM. The latter supply must be between +28 and +40 volts for the on-board 25-volt regulator to operate properly. Three (preferrably four) fresh 9-volt transistor radio batteries connected in series will give satisfactory operation if you don't want to build a separate ac-powered supply.

Two adjustments must be made on the PDB to ensure proper programming of EPROMs. First remove any EPROM that may be installed in the CPU module and attach the CPU module to the PDB. Apply power to the CPU module and depress the Program key on the PDB. (The unregulated 30-volt supply must be connected to the PDB for this adjustment.) Adjust *R16* on the PDB for +25 volts at pin 13 of *J3* (V_{pp}). (This voltage must lie in the range of +24 to +26 volts or damage to the EPROM could result.)

The second adjustment can be performed with the 30-volt supply disconnected. Depress the Program key while monitoring pin 11 of *J2* (PGM) with an oscilloscope. A low-frequency rectangular waveform should appear. Adjust *R5* so that the positive portions of this waveform are 50 ms ±5 ms in duration. These are the programming pulses to the EPROM which, in conjunction with the 25-volt programming supply, burns each memory location of the EPROM. There are 2048 program pulses during a complete burn cycle (corresponding to

From CPU Module Pin. No.	Function	To Interface Pin No.
J1		**P1**
1	A8	1
2	A9	2
3	A10	3
4	A11	4
5	A12	5
6	A13	6
7	A14	7
8	A15	8
9	A7	9
10	A6	10
11	A5	11
12	A4	12
13	A3	13
14	A2	14
15	A1	15
16	A0	16
J2		**P2**
1	DB0	1
2	DB1	2
3	DB2	3
4	DB3	4
5	DB4	5
6	DB5	6
7	DB6	7
8	DB7	8
9	$\overline{\text{I/O R}}$	9
10	$\overline{\text{I/O W}}$	10
11	PGM	11
12	GND	12
13	GND	13
14	+12	14
15	+5	15
16	+5	16
J3		**P3**
1	$\overline{\text{INT}}$	1
2	$\overline{\text{ROMS}}$	2
3	$\overline{\text{INTE}}$	3
4	$\overline{\text{STSTB}}$	4
5	$\overline{\text{RESET}}$	5
6	$\overline{\text{BUSEN}}$	6
7	$\overline{\text{MEMR}}$	7
8	$\overline{\text{MEMW}}$	8
9	INTA	9
10	ϕ2 TTL	10
11	$\overline{\text{RESIN}}$	11
12	RDYIN	12
13	Vpp	13
14	D5	14
15	SPARE	15
16	SPARE	16

Fig. 31. Connections between CPU module and interface from the PDB.



Power Connections

	+5V	GND
IC6, IC9	16	8
IC7, IC10, IC11, IC12, IC13	14	7

Fig. 30. Schematic of the EPROM programmer.

the 2048 memory locations of the EPROM). The adjustment of *R5* can be made with an EPROM installed if the unregulated 30-volt supply is left disconnected. The EPROM will not be programmed without this supply. (If an oscilloscope is not available, *R5* may be adjusted by setting it to the point such that the time interval between the initial actuation of the Program key and the PGM LED extinguishes is 108 seconds).

Using the PDB. Operation of the PDB is fairly simple once its functions are understood. As an illustration of its use we will load and run the program shown in Table II of last month's installment of this series. This program tests the Morse interface.

Before the program is loaded, the CPU module and Morse interface should be connected to the PDB (*P1, P2, P3* of the CPU module connect to *J1, J2, J3* of the PDB; *P1, P2, P3* of the PDB connect to *J1, J2, J3* of the interface). Set the PDB Run, Instruction Cycle, and Bank switches to off and apply power to the CPU module. The RUN LED of the CPU module should be off, indicating that the CPU is in a Wait state. The PDB should display address 0000 and the contents of this location, which is indeterminant now. The M1 and MR LEDs should be lit, indicating that the CPU is in an "instruction fetch" cycle.

To begin loading the program at the first memory address, depress the A/D key twice (the decimal point in the Data display should then light), which places the PDB in the Load Data mode. Subsequent numeric entries will now be deposited in the addressed memory location as an instruction (or data) and displayed in the Data display. Load the first part of the program by entering each instruction machine code with the numeric keys of the keypad. After each byte is deposited in its memory location, depress the Examine key to increment the address to the next sequential memory location (i.e., 3-1-EX, F-F-EX, and so on). After the byte at memory location 0008 has been loaded, it will be necessary to alter the normal sequential loading sequence by entering the subroutine's starting address (A/D, 0-1-0-0) and then continue by loading the subroutine instructions (A/D, 3-E-EX, 4-1-EX, and so on). After the subroutine's RET instruction has been loaded, the program will be ready to be executed.

Begin single-stepping the program by pressing the SS key. The first time this key is depressed the CPU will be reset and the PDB will go into the Execute mode (neither decimal point lit). Further actuations of the SS key will allow the CPU to execute one machine cycle each time the key is depressed. (*Machine cycles* are small units of processing activity which comprise each instruction cycle. Every instruction cycle consists of one to five machine cycles, depending on the type of instruction involved. A machine cycle is required each time an instruction requires the CPU to access memory or an I/O port). Table III follows the program's execution one machine cycle at a time using the PDB displays to monitor the CPU's activities (refer to the program listing and the individual instruction descriptions as necessary). Remember that the Address and Data displays are monitoring the CPU's Address and Data busses, respectively.

TABLE III
TESTING THE MORSE INTERFACE AND MACHINE CYCLE ILLUSTRATIONS

SS Key Depression Number	Address Display	Data Display	(LEDs ON)	Activity or Action
1	0000	31	M1,MR	The first instruction's op code (31) is being fetched (read) from memory. This is the first machine cycle of the instruction.
2	0001	FF	MR	The first instruction's first operand is being read from memory.
3	0002	0B	MR	The second operand of the first instruction is being read.
4	0003	CD	M1,MR	The three bytes of the first instruction were "put together" inside the CPU and executed. The first machine cycle of the Call instruction is entered (fetches the op code).
5	0004	00	MR	First operand is read from memory.
6	0005	01	MR	Second operand is read from memory.
7	0BFE	00	MW	The CPU is writing the high-order byte of the PC (00) into memory location SP-1 (0BFE).
8	0BFD	06	MW	The CPU is writing the low-order byte of the PC (06) in memory location SP-2 (0BFD). The stack now holds the return address.
9	0100	3E	M1,MR	The second-half of the previous instruction was executed (jump to memory address 0100). The CPU is now in machine cycle one of the first instruction of the subroutine.
10	0101	41	MR	The MVI A, 41H instruction's operand is read from memory.
11	0102	F6	M1,MR	The previous instruction was executed; fetching op code of ORI 80H instruction of subroutine.
12	0103	80	MR	Read operand from memory.
13	0104	D3	M1,MR	ORI 80H instruction was executed (accumulator contains $C1_{16}$); enter machine cycle one of output instruction is entered.
14	0105	FC	MR	Read operand (port address) from memory.
15	FCFC	C1	IOW	The CPU is writing the contents of the accumulator to port FC (Morse interface). A dc voltmeter would read approximately 4 volts at pin 4 of *IC3* of the interface at this time, which latches the contents of the Data Bus in *IC1* (Fig. 23). The letter "A" should appear at the output device if connected.
16	0106	DB	M1,MR	Fetch op code of Input instruction.
17	0107	FC	MR	Read port address from memory.
18	FCFC	FF/FE	IOR	The CPU is reading port FC. A dc voltmeter would read near 0 volt at pin 10 of *IC3* of the interface, which enables the tri-state buffers (*IC4*). Adjust *R1* of the interface, observing the LSB of the Data Bus. It should follow the detector, *LED1*.
19	0108	C9	M1,MR	The CPU stored the data placed on the Data Bus from the input interface in the accumulator. The last instruction of the subroutine is now being fetched.
20	0BFD	06	MR	The CPU is reading the data stored at the memory location specified by the SP (0BFD). This byte will soon be moved into the low-order byte of the PC.
21	0BFE	00	MR	The CPU is reading the data stored at memory location SP+1. This byte will go into the high-order byte of the PC. The CPU now holds the return address.
22	0006	C3	M1,MR	The second-half of the previous instruction was executed (jump to memory address 0006). The CPU is fetching the Jump instruction's op code.
23	0007	03	MR	Read low-order bits of Jump address
24	0008	00	MR	Read hi-order bits of Jump address
25	0003	CD	M1,MR	And we're back to start again!

After single-stepping through the program once, examine memory locations 0BFD and 0BFE, noting their contents. This is where the CALL instruction stored its return address. One may load different codes at memory location 0101 to view the different characters that can be displayed on the Morse interface display (Fig. 24). Note also how the PDB single-step mode can be used to troubleshoot an interface.

To burn this program into an EPROM for permanent storage, an erased EPROM must be installed at *IC5* in the CPU module and the unregulated 30-volt supply connected to the PDB. Pressing the Program key will then completely program the EPROM in a little less than two minutes.

After programming an EPROM it is wise to verify its contents by selecting it for a "test run" by the CPU (Bank and Run switches closed). The content of the program development memory is left intact after a burn cycle, making it possible to repeat the burn if necessary. If an EPROM won't verify properly after it is programmed (the program contained in it will not run properly), it is possible that it was not fully erased to start with. Re-erase and try again, or try another EPROM.

Once it has been determined that an EPROM contains valid information, a small piece of electrical tape or other opaque material should be placed over its transparent window to keep out ambient light. Sunlight and some types of artificial light can cause the EPROM's data to slowly decay and eventually become erased.

This series of articles has laid a foundation for one to design an interface and write his own machine-language programs for the popular 8080 family of microprocessors. What is built on this foundation is solely up to one's imagination and ingenuity. ◇