AN 8031 IN-CIRCUIT EMULATOR

BY GEORGE DINWIDDIE

Build this emulator and avoid debugging headaches

DEBUGGING A microprocessor-based program in assembly language can be difficult for a home engineer. The equipment to aid such a task—logic analyzers and emulators—is usually incredibly expensive. Even a large company might not be able to afford the thousands of dollars needed for an emulator because of the processor-specific nature of the tool. Such was the case when I was developing firmware for an embedded processor controlling a commercial product. The processor in use was an Intel 8031, and at that time, only Intel had an emulator for it. Obviously, Intel wanted a lot of money for the emulator. Worse, it needed a new microprocessor development system to host it.

My alternative was to edit, assemble, burn EPROMs, trace the execution using a logic analyzer, and deduce where the code was going wrong. Once I found a bug, I usually had to start over at the edit step before I could hunt for the next one. This was a slow process.

A BETTER WAY

It didn't take me long to become frustrated with this method. There *had* to be a better way. At the suggestion of a microprocessor guru, I decided to build a monitor. Because the product hardware was mostly developed, I designed the monitor board so that it could function as an emulator—plugging into the processor socket of the application hardware.

The primary benefit of my monitor is that it gives you the ability to patch errors without repeating the edit, assemble, burn EPROM cycle. Sometimes a patch alters only a byte or two, but more often than not it involves a jump to an unused section of RAM where a hand-assembled routine is inserted.

Another handy debugging tool is the monitor's breakpoint capability. While the breakpoint in my monitor is extremely limited and simple, it does provide a way to verify which path of a conditional jump was taken without having to resort to the logic analyzer. Furthermore, you can check the actual values in variables and internal registers instead of deducing them from external behavior.

Another technique I found helpful was to insert temporary test patches that displayed the value of some variable each time through a loop. This was accomplished by calling monitor subroutines via a convenient jump table provided at the start of the monitor. This technique was at the expense of real-time operation, but it provided insights into the dynamics of the software that were unobtainable by any other means.

I also included some basic but necessary functions that let you initialize memory, copy a block of memory, compare two blocks of memory, and perform a hexadecimal dump on a block of memory. Of course, I had to provide commands to edit the contents of memory and to execute instructions starting at any arbitrary location. Finally, I added some convenience commands to display the command syntax and the locations in the jump table and to perform hexadecimal arithmetic.

THE PROCESSOR

The Intel 8031 single-chip microcomputer is the ROM-less version of the

(continued)

George Dinwiddie (13808 Wayside Dr., Clarksville, MD 21029) is a software engineer for Mitron Systems Corp. He is currently working on his M.S. degree in computer science.

Listing 1: *The code for performing* ASCII-*hexadecimal to binary conversion.*

```
06EB 9430                  SUBB   A,#'0'
06ED 4017                  JC     BAD                  ; ACC < '0'
06EF 940A                  SUBB   A,#10
06F1 5005                  JNC    $+7                  ; ACC > '9'
06F3 240A                  ADD    A,#10     ;RESTORE
06F5 020704                LJMP   GOOD                 ; '0' <= ACC <= '9'

06F8 2403                  ADD    A,#('0'+10-'A'+10)         ;CORRECT FOR (-'0'-10)
                                                          & MAP 'A' INTO 10
06FA 20E709                JB     ACC.7,BAD            ; '9' < ACC < 'A'
06FD C3                    CLR    C
06FE 9410                  SUBB   A,#16
0700 5004                  JNC    BAD                  ; ACC > 'F'
0702 2410                  ADD    A,#16
0704 C3           GOOD:    CLR    C
0705 22                    RET
0706 D3           BAD:     SETB   C
0707 22                    RET
```

8051, the high end of Intel's 8-bit microcontrollers. It has five interrupts—two external inputs, two hardware timers, and one hardware asynchronous serial port (UART). It sports a 16-bit external address bus and 16 bidirectional I/O lines. The processor has three separate memory spaces—a 64K-byte program memory space, a 64K-byte external data memory space, and 128 bytes of internal RAM. The 8031 even features a hardware multiply and divide.

While this processor has a lot of speed and power, it has its share of problems, too. The architecture is accumulator-based. This means a lot of processing time and bytes of code are used to load and store the accumulator. The instruction set is highly irregular. For example, the only compare instruction is a compare-and-jump-if-not-equal. To do a compare-and-jump-if-equal requires an exclusive-or, a subtract, or a jump-around-a-jump. A jump-less-than or a jump-greater-than also involves extra work. You can see some of the problems this causes in the ASCII-hexadecimal to binary conversion in listing 1. The instruction set also has few instructions that are capable of accessing the two large external memory spaces.

The Emulator Board Hardware

The hardware design of the emulator board is straightforward. Figures 1a and 1b show the schematic diagram. It features 4K bytes of monitor EPROM, 8K bytes of application EPROM space, and 8K bytes of emulation RAM. With the current prices for larger memories it would make more sense to use 8K by 8 monitor and application EPROM

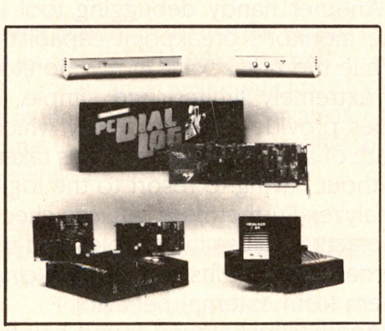
(continued)

You already own a computer that can talk.
Now let it.
Meet the Votrax Family of Voice Products.

★ **P.C. Dial/Log**
Televoice management system
Digitizes your voice
Saves and sends voice messages
Auto answer/Auto dial
Call store and forward
Message distribution
P.C. to P.C. communication
For IBM-PC, XT, AT and compatibles

★ **Votalker IB & AP**
Board level speech synthesizers
Unlimited vocabulary
Text-to-speech software
Voice mode selection switch
Speech filter selection switch
For IBM-PC, XT, AT and compatibles
For Apple II, IIE, II Plus and compatibles

★ **SC-01 · SC-02**
Phonetic speech synthesizer chips
Unlimited vocabulary



Now you can upgrade almost any personal computer and make it more powerful than ever, by giving it the power of speech.

★ **P.S.S. & T.N.T.**
Stand-alone speech synthesizers
Unlimited vocabulary
Text-to-speech
Parallel/RS-232 compatible
Exception word table
On-board music chip
Adaptable to most computers

★ **Votalker C-64**
Plug in speech synthesizer
Unlimited vocabulary
Text-to-speech
Rate, volume and pitch control
Three speaking modes
Free trivia game
For Commodore C-64/128

★ **Software Developers Welcome**

★ **Dealers and Distributors Welcome**

votrax
1394 Rankin, Troy, Michigan 48083
1-800-521-1350
In Michigan 313-588-0341
For a voice demonstration call
313-588-2926

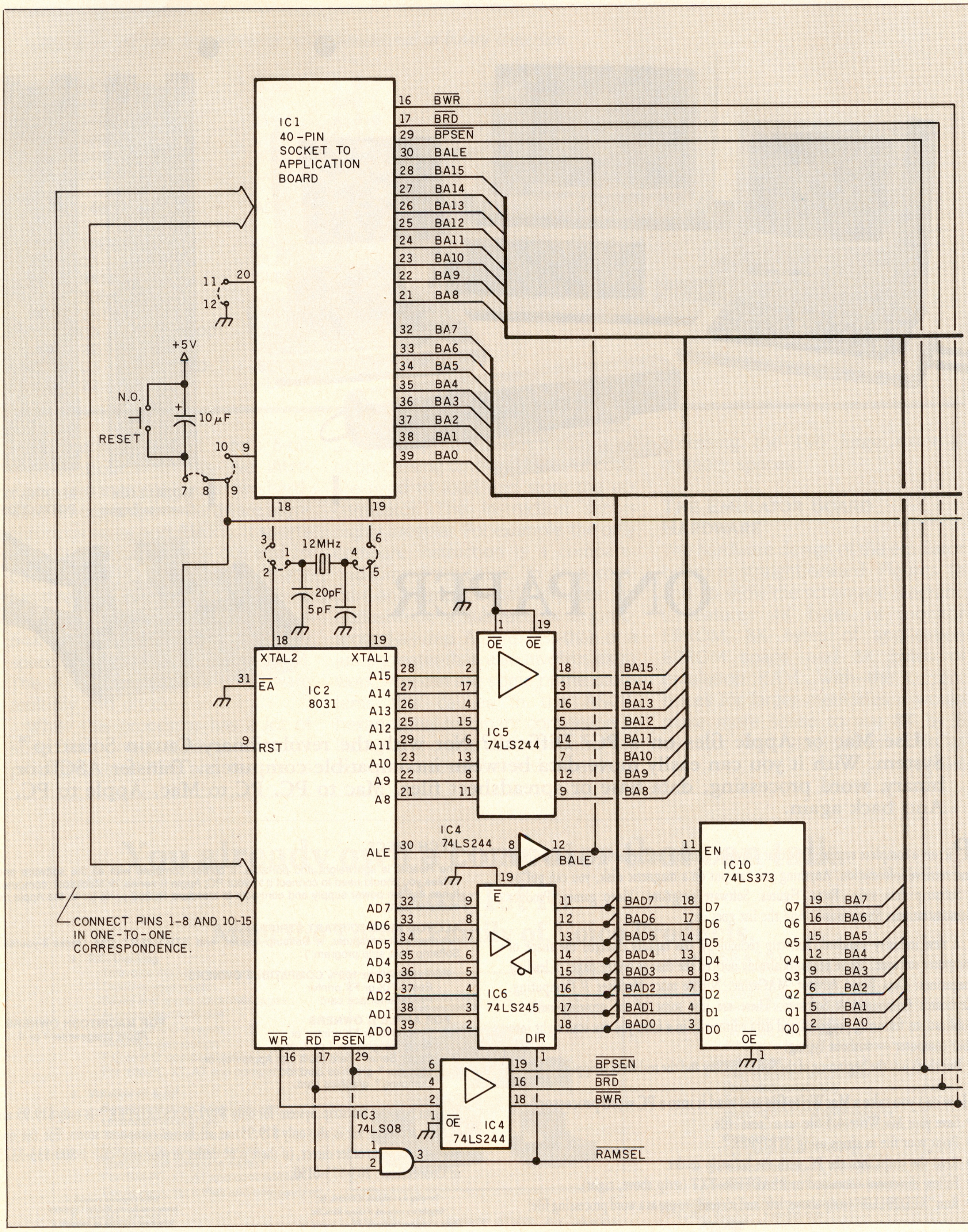
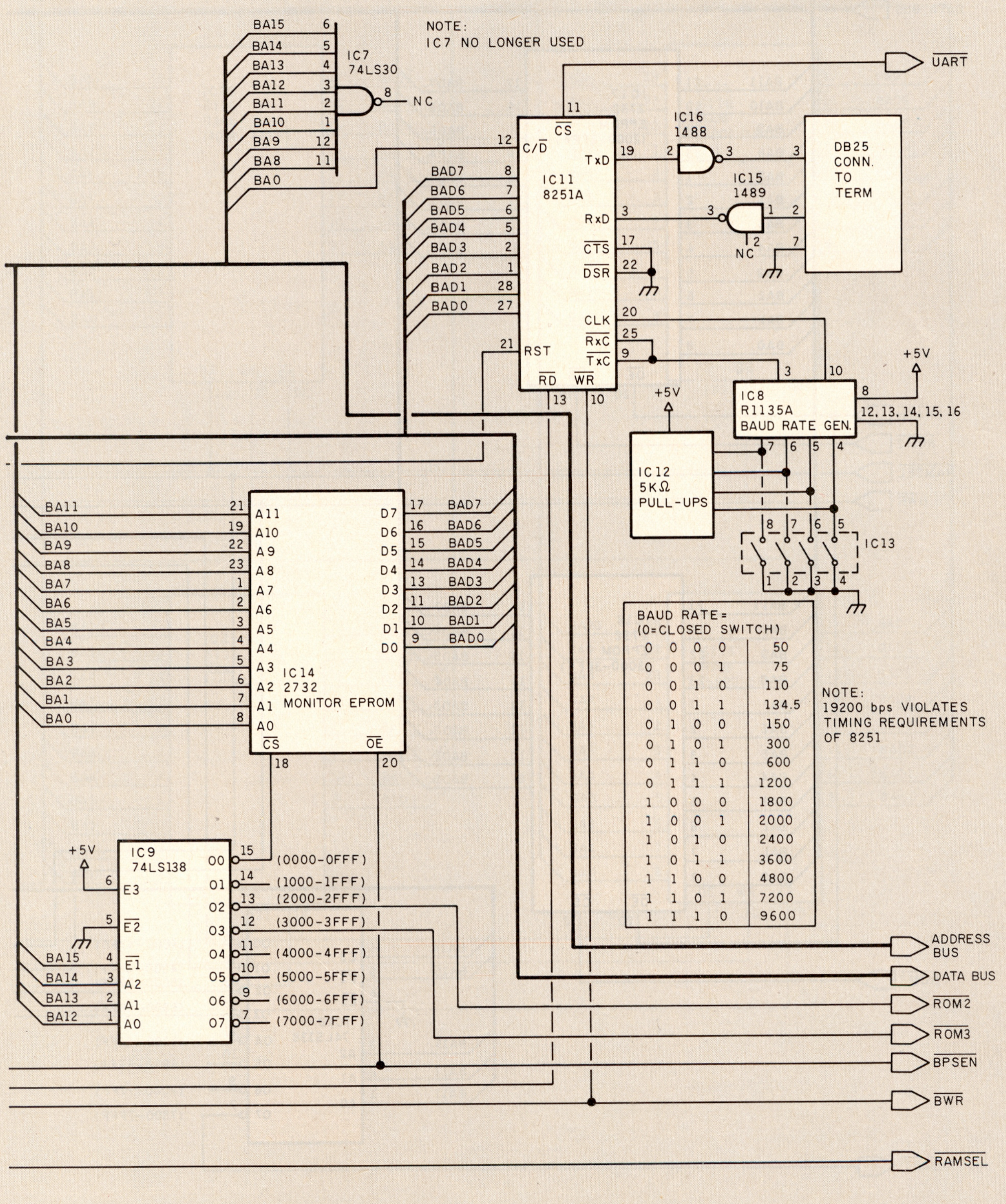


Figure 1a: *Schematic of the 8031 emulator board, showing the central processor, UART, and address-decoding circuitry.*



NOTE:
IC7 NO LONGER USED

BAUD RATE=
(0=CLOSED SWITCH)

0	0	0	0	50
0	0	0	1	75
0	0	1	0	110
0	0	1	1	134.5
0	1	0	0	150
0	1	0	1	300
0	1	1	0	600
0	1	1	1	1200
1	0	0	0	1800
1	0	0	1	2000
1	0	1	0	2400
1	0	1	1	3600
1	1	0	0	4800
1	1	0	1	7200
1	1	1	0	9600

NOTE:
19200 bps VIOLATES TIMING REQUIREMENTS OF 8251

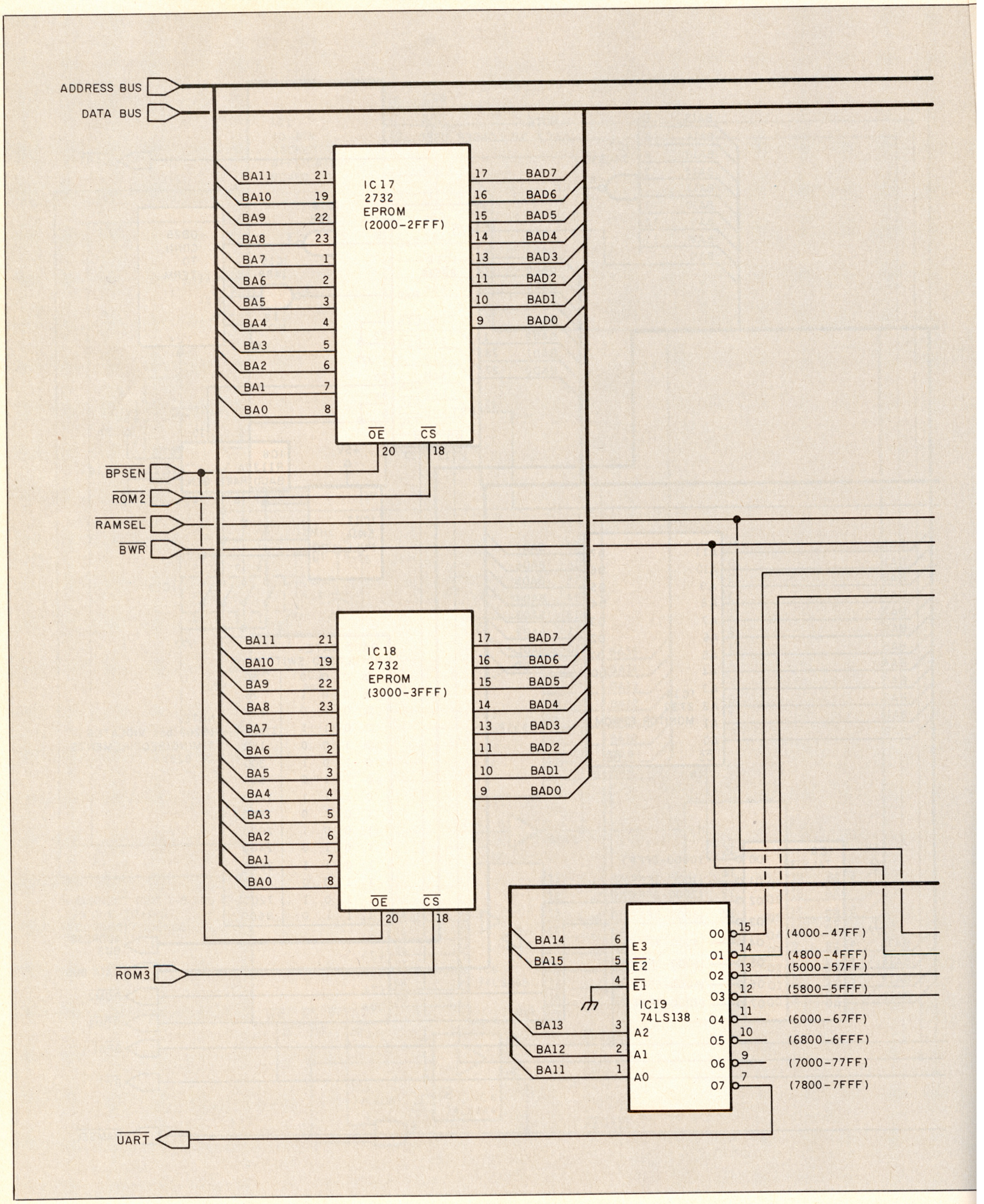
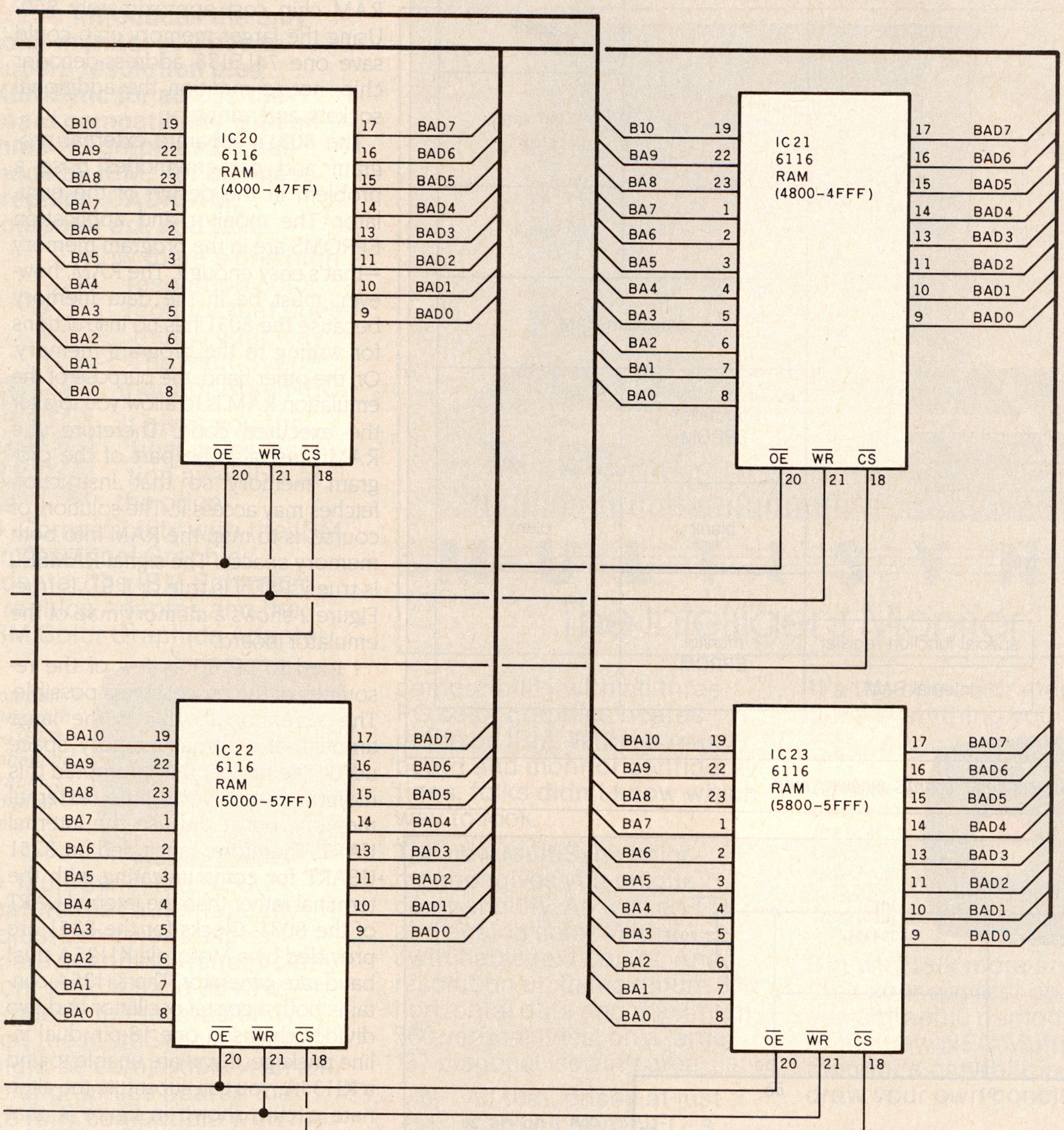


Figure 1b: *Schematic of the 8031's ROM and RAM circuitry.*



IC20 6116 RAM (4000–47FF)
IC21 6116 RAM (4800–4FFF)
IC22 6116 RAM (5000–57FF)
IC23 6116 RAM (5800–5FFF)
B10 19
BA10 19
BA9 22
BA8 23
BA7 1
BA6 2
BA5 3
BA4 4
BA3 5
BA2 6
BA1 7
BA0 8
17 BAD7
16 BAD6
15 BAD5
14 BAD4
13 BAD3
11 BAD2
10 BAD1
9 BAD0
OE 20
WR 21
CS 18

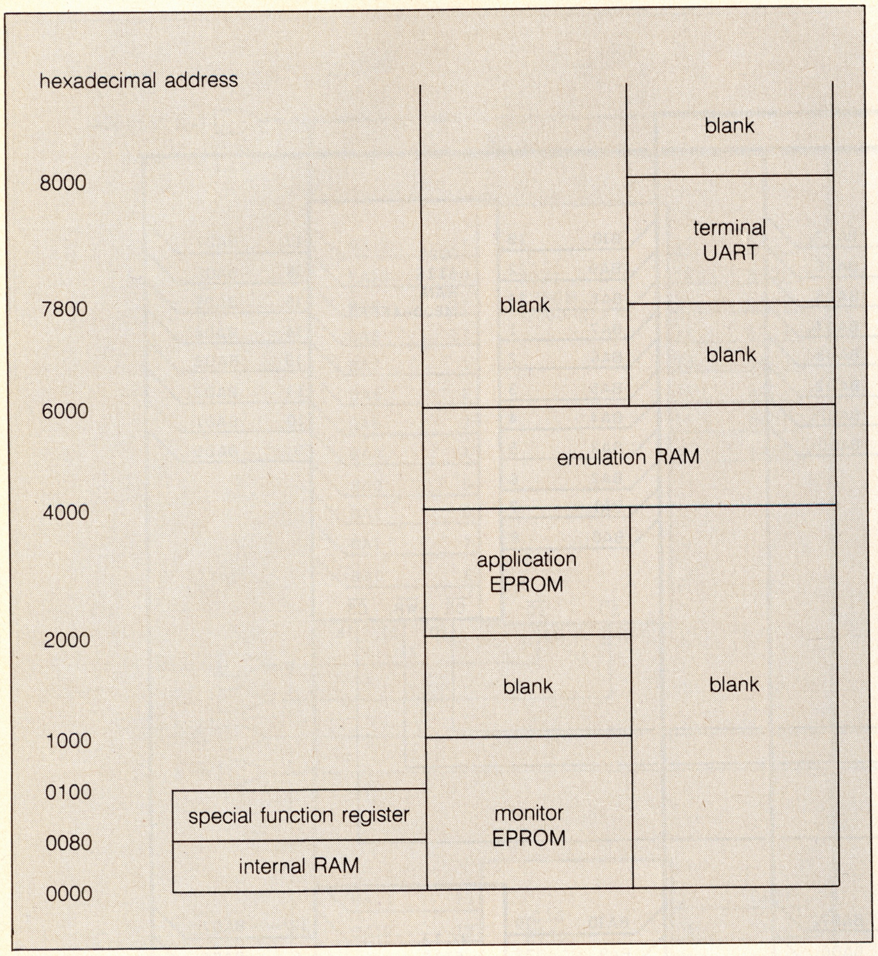


Figure 2: *The emulator board's memory map.*

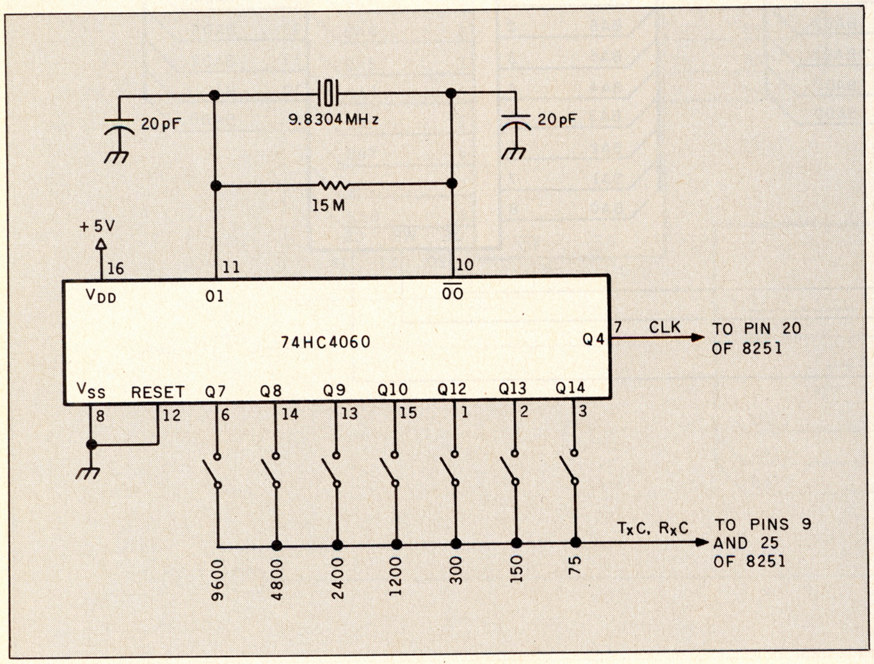


Figure 3: *Alternate circuit for generating baud-rate clock signal.*

chips and an 8K by 8 emulation RAM chip. When this circuit was first designed, however, an 8K by 8 static RAM chip cost approximately $50. Using the larger memory chip could save one 74LS138 address-decoder chip, not to mention the additional sockets and wiring.

The 8031's separate external program and data memories pose a problem in the design of the emulator. The monitor and application EPROMS are in the program memory —that's easy enough. The RAM, however, must be in the data memory because the 8031 has no instructions for writing to the program memory. On the other hand, the purpose of the emulation RAM is to allow you to alter the executed code. Therefore, the RAM must also be part of the program memory so that instruction fetches may access it. The solution, of course, is to map the RAM into both memory spaces. The signal $\overline{\text{RAMSEL}}$ is true if $\overline{\text{PSEN}}$ is true or if $\overline{\text{RD}}$ is true. Figure 2 shows a memory map of the emulator board.

I tried to tie up as few of the resources of the processor as possible. The exception to this is the large amount of external memory space used. The project that prompted this monitor used very little external memory, but it did use the internal UART. Therefore, I included an 8251 USART for communicating with the terminal rather than the internal UART of the 8031. Clocks for the 8251 are provided by a Motorola K1135A dual baud-rate generator. The K1135A contains both a crystal oscillator and two divider chains in one 18-pin dual inline package. If you are unable to find a K1135A, you can substitute the alternate circuit, shown in figure 3, that uses a 74HC4060 oscillator/divider chip.

The address, data, and control signals are buffered to allow for the extra loads placed on them by circuitry on the monitor board. If these signals are also buffered on the target board, it may cause excessive propagation delays. If necessary, replace one set of buffers with jumpers. Jumpers are also provided to choose either the onboard crystal or an external clock

(continued)

from whatever application board you are using the emulator with. Similarly, there is a jumper selection enabling the on-board reset button, the application board reset, or both. Another jumper allows connecting or isolating the grounds between the target and emulator boards.

Operation of the Software

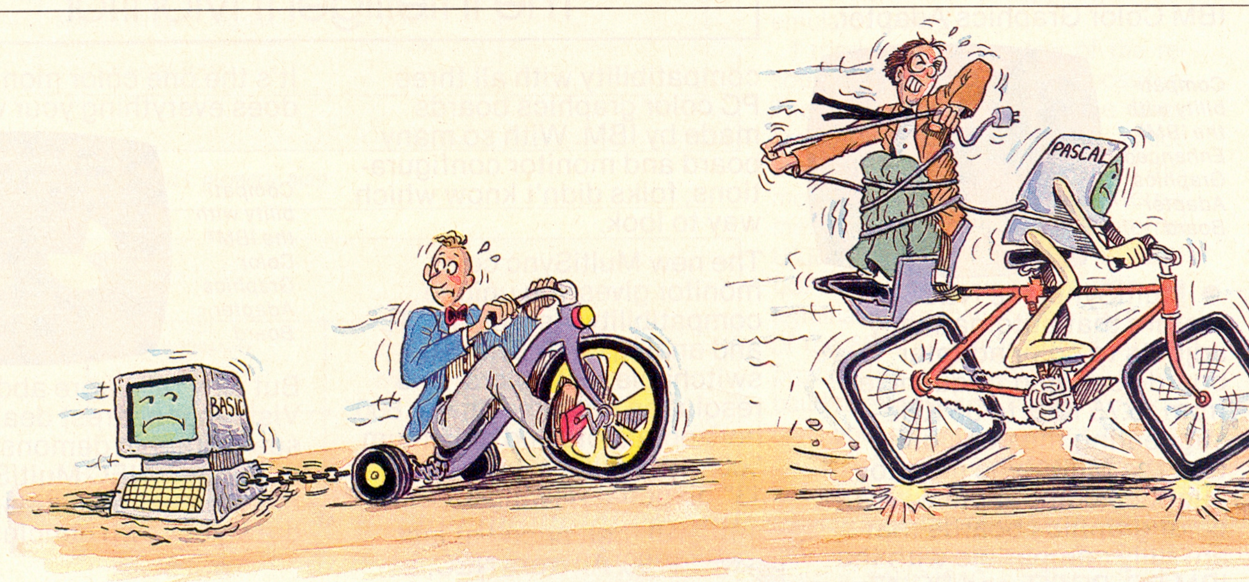
First, edit and assemble your source code. Program the object code in EPROMs and plug them into the application code space. Note that the code will be executed out of the emulator board's RAM. Therefore, the origin statement in the source code should specify 4000 hexadecimal, the start of RAM, rather than 2000 hexadecimal, the start of the application EPROM. The interrupt vectors in the monitor EPROM all jump to the equivalent offset in the RAM space, allowing use of all the interrupts with only a slight additional latency. The reset vector, however, jumps to the monitor initialization code. This scheme allows the application EPROMs, burned for development purposes, to be used in the final project in some cases. In the products I developed with the aid of this monitor, the memory decoding was incomplete. The same EPROM was addressed at 4000 hexadecimal and 000 hexadecimal. [*Editor's note: The source code for the author's monitor program,* UGHBUG.ASM, *is available in a variety of formats. See pages* 459–461 *for details.*]

The Commands

The commands are invoked by the first character of the command name. There are a few exceptions to this rule. The internal varieties of the DUMP and ALTER commands (which access the 8031's internal memory) have an "I" appended to distinguish them from their external equivalents. The HEXMATH command is invoked with a number sign (#). All numeric values are expressed in hexadecimal. If you mistype an address or a data value, just keep typing. The monitor accepts the last four digits entered for address values and the last two for data values. You need not type leading zeros unless you're covering up a mistake. To cancel a command that you've started to type, just type any illegal character. To abort a command that outputs to the screen, merely type any character. These rules are consistent for all the commands.

The first command needed in any debugging session is the COPY command. Copy the application code from the EPROM to the RAM. Be sure not to overwrite the last 9 bytes of RAM; these are used by the breakpoint routine.

Next, use the VERIFY command to make sure the transfer was successful. VERIFY indicates agreement between two blocks of memory by doing nothing. Any differences are displayed. (I never got around to adding a memory test. With only four RAM chips it



A Personal Language

When it comes to problem solving, the APL★PLUS System is the undisputed leader.

That's because the APL★PLUS System works with you. It goes far beyond what application software like Lotus® or dBASE® could possibly ever offer. And, it won't tie you down with the details of standard programming languages.

The APL★PLUS System is a personal language, with productivity features that help you concentrate on getting answers, rather than struggle with intricate calculations and modeling.

With it you can manipulate tables of numbers as easily as single numbers and get quick results from your computer using short, simple statements.

When you've reached the limits of other packages, move up to the APL★PLUS System. It's a powerful and flexible tool that grows with you as your needs become more sophisticated. With over 200 built-in applications—like graphics, report formatting and communications—you have all the tools at your fingertips to quickly and easily solve those seemingly impossible problems.

Best of all, the APL★PLUS System interfaces well with software packages you're already using—like databases, spreadsheets, and graphics packages. The APL★PLUS System also makes it easy to link those packages that aren't didn't seem worth the effort. Usually, corrupted memory was the result of stack overflow or some other errant code.) Always VERIFY the damages after your code goes into the weeds.

Run your code using the GO command. If you do not specify an address, execution will restart at the breakpoint. (I'll talk more about breakpoints later.) Although the most likely starting place is 4000 hexadecimal—the reset vector of the application code—you can GO to any address in the program memory.

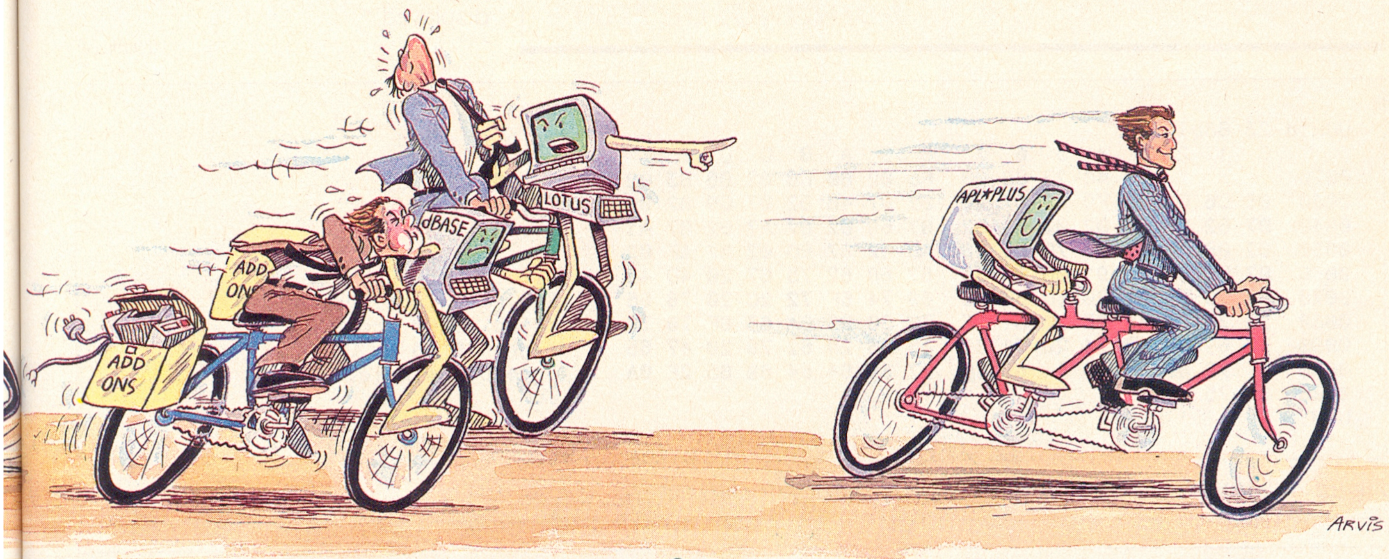
I had to resort to some tricks in my monitor's code. The 8031 does have an indexed jump instruction, JMP @A + DPTR. Unfortunately, the data pointer is only easily loaded by a constant—a variable must be loaded a byte at a time—and the accumulator is only 8 bits. Besides that, I wanted to be able to restore the data pointer and the accumulator when resuming after a breakpoint. The simple solution, shown in listing 2, is to push the address onto the stack and execute a return-from-subroutine instruction.

Before running your code you may want to set a breakpoint. The BREAK routine requires a little explanation. Many processors have a single-byte software-interrupt instruction that can be used for breaking back to the monitor. This single byte may be substituted for the first byte of any instruction. When the software interrupt is executed, it transfers control to a breakpoint routine. The 8031 lacks such an instruction; you have to use a long jump instruction, which is 3 bytes long. The break address must be aligned with the first byte of an instruction so that it will be executed and not treated as data.

First you want to save the original instructions. Because an 8031 instructon may be 1, 2, or 3 bytes long, inserting the 3-byte jump instruction at a given point in the code may clobber a sequence of 3, 4, or 5 bytes of code. The number of bytes affected is the optional final parameter of the break command; 3 is the most convenient value and is the default. This parameter is stored at the location BYTENUM, and the bytes of code are stored in the following 5 bytes (padded with NOP instructions if necessary). Following this, the next 3 bytes of RAM are filled with a jump instruction to the location following the code that was copied, that is, BYTENUM bytes after the break address. After all this is done, a jump-to-the-breakpoint-routine instruction is written at the break address. When execution reaches the break address, control passes to the breakpoint routine. The breakpoint routine saves the registers that are treated as volatile memory and displays the values that they contained.

A GO command without a starting address resumes from the break address. First it restores the saved registers and executes the saved code,

(continued)



The APL★PLUS® System and You.

currently talking with each other.

With all this problem-solving power, it's no wonder STSC's APL★PLUS System is the personal choice of so many business professionals—financial planners, business analysts, actuaries, scientists, mathematicians, engineers, statisticians, and consultants. Especially since the APL★PLUS System is available on a full range of computers from desktops to mainframes.

Put the power, speed, and flexibility of the APL★PLUS System to work for you. See your local dealer today to get your APL★PLUS System. If they don't have it, refer them to STSC or call STSC toll-free, **(800) 592-0050.**
In Maryland or Canada, **(301) 984-5123.**

Available nationally through Softsel, Micro Central, and distributors worldwide. Dealer inquiries welcome.

Problem-Solving at the Speed of Thought

STSC STSC, Inc.
2115 East Jefferson St.
Rockville, MD 20852

A PLUS★WARE™ PRODUCT

Inquiry 337

and then it jumps back to the application code following the break address. If you execute the breakpoint routine, you should reset the board before attempting to initiate a GO command to a starting address. This will recover the 5 bytes of stack space used to store the volatile registers.

The code saved at location BYTENUM+1 is restored to its original location when a new break address is entered or if the BREAK command is invoked without a new address. You should do this before a reset because the initialization code clears BYTENUM and the following locations.

The DUMP command comes in two flavors, internal and external. The external version does a memory dump of the program memory in hexadecimal, showing the ASCII translation to the right (see figure 4). If no ending address is specified, 0FFFFH is assumed. A DUMP may be interrupted, as can any command that writes to the screen, by typing any key.

The internal DUMP command is similar but dumps the internal RAM and the special function registers. No ASCII translation is shown since it is unlikely to find ASCII strings in internal memory. The special function registers are indicated symbolically in addition to their addresses. This is shown in figure 5.

The ALTER command also comes in two flavors. The external version displays the current byte from program memory followed by a dash. If you enter a hexadecimal value, that value will be inserted at that location in external data memory. A space or a carriage return leaves the location unchanged and displays the next byte (carriage return puts you on a new line, space leaves you on the current line). A period or a backspace backs up the displayed byte by one location. Any other nonhexadecimal character cancels the command. Remember that the RAM on the monitor board is mapped to both the program memory and the data memory. The ALTER command writes to data memory because there is no way to write to program memory. It is frequently convenient to use the ALTER command to check code, hitting carriage return after each instruction for readability.

The internal form of the ALTER command accesses the 8031's internal RAM. As in the DUMP command, when you reach the special function registers (locations above 7F hexadecimal) the name of the register is displayed.

(continued)

Listing 2: *The author's solution to the* 8031*'s lack of an indexed jump instruction.*

```
060F            GXXXX:
060F C049                   PUSH     LOBYTE
0611 C048                   PUSH     HIBYTE
0613 22                     RET
```

```
UGH:d 23 00b1
      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
0020           02 40 23 02 00  F5 02 00 F8 02 06 B3 02       .@#.. ........
0030  06 C6 02 06 F0 02 07 20  02 07 48 02 07 C9 02 06    .......  ..H.....
0040  D7 02 06 CD 02 07 6C 02  07 69 02 07 A7 02 07 A1    ......l. .i......
0050  02 08 16 02 07 70 02 06  AD 02 07 BD 02 07 AF 02    .....p.. ........
0060  07 EA 02 05 91 0D 0A 55  67 68 62 75 67 20 4D 43    .......U ghbug MC
0070  53 2D 35 31 20 6D 6F 6E  69 74 6F 72 2C 20 76 65    S-51 mon itor, ve
0080  72 73 69 6F 6E 20 31 2E  30 30 0D 0A 63 6F 70 79    rsion 1. 00..copy
0090  72 69 67 68 74 20 31 39  38 36 20 62 79 20 47 65    right 19 86 by Ge
00A0  6F 72 67 65 20 44 69 6E  77 69 64 64 69 65 2E 0A    orge Din widdie..
00B0  04 75                                               .u
UGH:
```

Figure 4: *Sample output of the external version of the* DUMP *command.*

```
UGH:di 69

      0  1  2  3  4  5  6  7   8  9  A  B  C  D  E  F
60                                9E AC C9 1C CE CC E4
70    88 90 80 30 02 20 88 A2  DF 9B EC E4 BB 46 E6 E4

80=P0  :55  81=SP  :52  82=DPL :69  83=DPH :08  87=PCON:7F  88=TCON:00
89=TMOD:00  8A=TL0 :00  8B=TL1 :00  8C=TH0 :00  8D=TH1 :00  90=P1  :FF
98=SCON:00  99=SBUF:00  A0=P2  :08  A8=IE  :60  B0=P3  :FF  B8=IP  :E0
D0=PSW :29  E0=ACC :01  F0=B   :0D
UGH:
```

Figure 5: *Sample output of the internal version of the* DUMP *command.*

Both the DUMP and ALTER commands use the SFR table (special function registers table—it can be found near the end of the monitor's source code listing) to access these registers. The special function registers cannot be accessed indirectly, and the memory space they occupy is sparsely populated. The SFR table provides a solution to both of these problems. Each table entry is 11 bytes long. The first 5 bytes give the symbolic name of the register. At an offset of five is a subroutine to read the register. Within this subroutine, at an offset of six, is the hexadecimal address of the register. Eight bytes from the start of each entry is a subroutine to alter the contents of the register. Some registers, such as the stack pointer, would crash the system if they were altered, so these entries return an error flag instead. Some registers, such as the accumulator, are treated as volatile by the monitor. You can alter them, but the monitor makes no attempt to preserve the contents of these registers.

The MODIFY command allows you to enter information into external data memory as characters instead of hexadecimal values. There are only two special characters to remember within this command. A backspace backs up the address pointer to allow the correction of mistakes. An EOT (Control-D) terminates the command. There is no internal version of the MODIFY command since it seems unlikely that the limited internal RAM would be used for storing strings.

The INSERT command fills a selected block of external RAM with a single hexadecimal byte. I used INSERT rather than FILL because I intended to add a FIND command.

The final three commands, HELP, JUMPTABLE, and HEXMATH, are conveniences. HELP displays the syntax of the commands and JUMPTABLE displays the entry points to the utility subroutines. HEXMATH, invoked by #, performs both addition and subtraction in hexadecimal. This is convenient for calculating relative jumps.

Some people might consider these last commands to be insignificant, but I disagree. It takes a short time to forget the command syntax. The HELP command also makes this tool easily available to others. The JUMPTABLE command allowed me to create test patches easily. Before I added this command I was constantly searching for the monitor program listing to look up the jump-table addresses. The HEXMATH is a cheap convenience—convenient to include when compared with the trouble caused by missing a jump target by 1 byte. Even simple software such as this should attempt to be as helpful as possible—there are better ways to spend time.

CONCLUSION

In this article, I have focused on one particular implementation of a debug monitor. If you are using a different processor it is well worth the effort to create your own. In addition to the debugging help, writing a monitor is a good exercise. It helps you to become familiar with a new instruction set.

Build your monitor one function at a time. First start with displaying a sign-on message. Then accept input and echo it back to the screen. Once you have the terminal interface working, add the more basic commands such as DUMP and ALTER. At this point you will have tools to aid you in developing the rest of the code.

It took me two weeks to develop the hardware and enough of the software to start using my new tool to develop application code. A year later I was still adding features and refining functions. Every minute I spent working on the monitor was quickly repaid in time saved. In the first week I used it, I accomplished six weeks of debugging measured by my previous standards.

What features would I like to add to this or any other monitor? A LOAD command to download a hexfile directly to RAM would be first choice. A FIND or SEARCH command to pick out variable-length byte sequences would be nice. A disassembler and single-line assembler would be a great help. These two are big jobs, though. I got to be pretty good at patching code with hand assembly, but I was doing it every day. The list of features could go on and on. A monitor is never finished until you quit using it. ■

Special thanks to Jim Gaudreault.