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.
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 and access the data from RAM.
Figure 1a: Schematic of the 8031 emulator board, showing the central processor, UART, and address-decoding circuitry.
Figure 1b: Schematic of the 8031’s ROM and RAM circuitry.
AN 8031 EMULATOR

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 RAMSEL is true if PSEN is true or if 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 in-line 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 on-board crystal or an external clock.
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 breakpoint 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 instruction may be 1, 2, or 3 bytes long, inserting the 3-byte jump instruction at a given point in the code may clutter 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-routing 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)
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 BYTE-NUM+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 BYTE-NUM 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, OFFFFH 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.

---

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

<table>
<thead>
<tr>
<th>060F</th>
<th>C049</th>
<th>0611</th>
<th>C048</th>
<th>0613</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>GXXX:</td>
<td>PUSH</td>
<td>LOBYTE</td>
<td>PUSH</td>
<td>HIBYTE</td>
<td>RET</td>
</tr>
</tbody>
</table>

---

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

---

**Figure 5: Sample output of the internal version of the DUMP command.**
27 million
Americans can't read.
And guess who pays the price.

Every year, functional
illiteracy costs American
business billions.
But your company can
fight back...by joining
your local community's
fight against illiteracy.
Call the Coalition for
Literacy at toll-free
1-800-228-8813 and find
out how.
You may find it's the
greatest cost-saving
measure your company
has ever taken.

A literate
America is a
good investment.