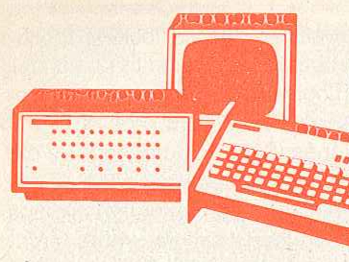

Computer Bits

By Hal Chamberlin

DEBUGGING AIDS

WHAT DO you use your computer's front panel for? Loading programs into memory, monitoring their execution, altering their execution with the sense switches, single-stepping the program to find errors ("bugs"), changing memory locations, and troubleshooting hardware are some examples. Last month, the role of monitor programs such as Motorola's Mikbug in performing routine computer operation tasks was described here. It should have been apparent that the use of a monitor for such functions is far more convenient than using a traditional front panel. But what about non-routine, troubleshooting tasks such as tracing the execution of a new, untried program or locating a hardware malfunction on a new board? Many hobbyists feel that a control panel is indispensable in such situations. Let us examine how monitor software can be used to perform even these "debugging" functions more effectively than a front panel.

Breakpoints. Assume that you have just finished writing a relatively complex program of 200 instructions in machine language and are ready to test it. First you load it into memory, hopefully using a monitor and keyboard. After loading by hand, you save the program on tape just in case it wipes itself out in memory as errant programs often do. Then using the monitor's "G" command you execute the program. Chances are it does not execute properly. In fact, it's probably not even close. At this point debugging begins, which can take considerable time without good debugging tools.

If you have an Altair, Imsai or other front-panel oriented computer you will probably single-step your way through the program in an effort to find out where it goes awry. This means simply that the computer is placed in a single-step operating mode where every operation of the "single-step" switch causes exactly one machine cycle to be executed. Generally, as each cycle is executed, you can see in the console lights the memory or input/output address referenced by the cycle and the data transferred. Additional status lights identify the type of cycle out of about a half-dozen possibilities. However there are many things that the panel lights fail to show. For example, when executing an "add register B to register A" instruction in single-step mode, you will only see the memory address of the instruction and the operation code, 200 in octal. You will not see the contents of register A, register B, or the result of the addition. Even the condition flags such as overflow are hidden from view. Obviously the panel is of limited value if you think the root of a particular bug lies in incorrect register contents or if there is an uncertainty as to what a specific instruction does. If this information is critical (as it often is), you must temporarily modify your program to store the necessary data into memory and then halt so that it may be examined.

Another often encountered difficulty is that single-stepping through a loop a dozen times to catch an error that occurs on the thirteenth iteration can take a long time. If the problem occurs on the 387th iteration it would not even be practical to single-step. Again the program must be temporarily modified to make it halt close to the error condition. Such temporary patches are called "breakpoints."

Many monitors have commands or functions that make inserting and keeping track of breakpoints much easier. A command like "3:213B" might insert a breakpoint at location 213 in page 3 (octal notation). What would actually happen is that the monitor would first look at the indicated address and save whatever was there. Next it would store a CALL instruction in the same location which would transfer control to a "breakpoint subroutine" in the monitor. This monitor routine prints the contents of all registers and the condition flags.

Now that the breakpoint is set up, the program would be entered with the normal "G" command. When it got to the "CALL BREAKPOINT" instruction that was inserted, the registers and flags would be printed. After the printout, the monitor would be waiting for another command. When the breakpoint is no longer needed, an "E" command might erase the breakpoint and restore the instruction it saved.

A more sophisticated monitor could allow multiple breakpoints. It would automatically keep track of the instruction displaced by each breakpoint and identify the breakpoint when the registers were printed. A really good breakpoint routine might even execute the saved instruction after printing and then automatically return to the program being debugged. In this case, a breakpoint could not be placed on top of a JUMP or CALL instruction.

A breakpoint routine can also take care of the "error on the 387th loop" problem mentioned earlier. Each time the breakpoint routine is entered, a software counter is decremented. If the counter is not zero, the user program is re-entered without printing the registers. Only when the counter finally does become zero do the registers get printed—thus saving a lot of time and paper. There would, of course, be a command to set the initial value of the counter.

Some microprocessors make the task of implementing a breakpoint facility in a monitor much easier. In the 8080 a normal CALL instruction is three bytes long. When placed in a breakpoint location, the three bytes that were there have to be saved. These three bytes might represent as many as three separate instructions making the "print registers and continue" function very difficult to implement. The 6800 or 6502, on the other hand, has a one-byte BREAK instruction that seems to be custom designed for just this function.

Software Single-Step. Although breakpoint capability in a monitor greatly simplifies the debugging of machine language programs, it is not real single-stepping. A different class of monitor routines called "trace routines" allows the software equivalent of single-stepping. It is interesting to note that some microprocessors are "dynamic" and cannot be stopped to allow the usual hardware single-step function. With these, a trace routine is the only way to get a single-step operation.

Tracing is really equivalent to putting a breakpoint at every instruction in a program. Then when the program is executed, a printout of the location, instruction, and all registers would be given for each instruction executed. This would be exactly equivalent to manual single-stepping with the bonus of a written record of every aspect of the program execution. In a machine with a lot of registers, time and paper may be saved by only printing the registers that have *changed* since the last printout. A useful trace feature in a monitor would allow the setting of *trace limits* so that only the program section of interest would be traced. A fancy trace feature might even allow multiple sets of trace limits with possibly a counter to delay the printing until a specified number of traced instructions has been executed.

How are trace routines actually implemented? One simple method is to use the interrupt feature of the microprocessor itself. With this method, a simple circuit added to the computer is activated to issue an interrupt whenever an instruction is executed. This interrupt would prohibit further interrupts and cause execution of the monitor trace print subroutine. The print routine would not re-enable interrupts until just before it returns to the interrupted program, the one being traced. This prevents the trace print routine from being traced itself. The trace limit feature is implemented by having the trace print routine check if the instruction about to be printed is within the trace limits. If it is not, printing is suppressed and a return to the program is executed. An example is shown in Figs. 1 and 2.

Another method involves *interpreting* the instructions of the traced program rather than executing them. An interpreter is a program that acts just like the microprocessor itself. It literally looks at the operation codes, addresses, and other components of the instruction and, through software, accomplishes the same result as the real microprocessor would have. The purpose of this is that the interpreter program may also store or print detailed information about the instructions it "executes", something the real microprocessor, of course, would not do. This technique requires much more complex software than breakpoints or trace using interrupts does but it has an important advantage. Since the interpreter routine *simulates* the machine, it can also simulate hardware features that the real machine may not have, such as memory protect. While debugging, the simulator would trap any instruction that attempts to jump outside of the protected area as well as any instruction that tries to write into protected memory.

Unfortunately the standard, readily available monitors in read-only memory generally do not have these debugging functions. If anything, a simple breakpoint facility is all that is offered. Specialized monitors, designed primarily for debugging rather than routine operations, on the other hand can probably do everything that has been discussed as well as other handy functions. These monitors are often found in the microprocessor manufacturer's development systems, such as Intel's MDS or Motorola's Exorciser (meaning to "exorcise" bugs) and are quite expensive. However the functions described are not difficult to implement and are certainly worth the effort needed to write them. Club meetings provide opportunities to exchange such software with fellow hobbyists. ◇

```
                          *      TEST OF DOUBLE PRECISION SUBTRACT

001:000 006 123                  MVI  B,123Q       SET BC TO 123:156 = 21358 DECIMAL
001:002 016 156                  MVI  C,156Q
001:004 026 356                  MVI  D,356Q       SET DE TO 356:312 = -4405 DECIMAL
001:006 036 312                  MVI  E,312Q
001:010 315 000 003              CALL DSUB         DO THE SUBTRACTION
001:013 311                      RET               RETURN TO THE MONITOR

                          *      DOUBLE PRECISION SUBTRACT REGISTERS B AND C FROM
                          *      D AND E WITH RESULT PLACED IN D AND E

003:000 173               DSUB   MOV  A,E          SUBTRACT LOWER BYTES
003:001 221                      SUB  C
003:002 137                      MOV  E,A          MOVE RESULT INTO E
003:003 172                      MOV  A,D          SUBTRACT UPPER BYTES
003:004 230                      SBB  B
003:005 127                      MOV  D,A          MOVE RESULT INTO D
003:006 311                      RET               RETURN
```

Fig. 1. Program to be traced.

```
002:232 123  3:0,3:6T        (Command to trace between 003:000 and 003:006
002:232 123  1:0G            (Command to start execution at 001:000

003:000 173  A=312 B=123 C=156 D=356 E=312 H=113 L=002 SP=017:374 FLG=
003:001 221  A=130
003:002 137  E=130
003:003 172  A=356
003:004 230  A=203 FLG=SA
003:005 127  D=203

002:232 123  (next command)
```

Fig. 2. Monitor printout with trace. Underlined portion typed by user.