LAST ORDERS

There are three commands for our debugging program that are yet to be designed. Before we look at these, however, we will consider the interrupt mechanism used to transfer control between the debugging program and the program being debugged at the breakpoints. We will also design the initialisation procedure.

The interrupt mechanism is used at breakpoints in the original program, where we have replaced an instruction with an SWI (SoftWare Interrupt) op-code. The SWI, like the other interrupts on the 6809, is vectored through a specific memory location — namely, $FFFA. This means that when an SWI is executed the registers are saved on the stack and the processor loads the 16-bit address at $FFFA and $FFFB into the program counter (PC). Execution then continues from that address. Our task is to change this vector so that it points to the entry point of our debugger program. One problem here is that interrupt vectors are almost always held in ROM. The fact that these addresses are fixed, therefore, means that the operating system must have some other means of vectoring interrupts.

The normal system is to have a jump table (see page 639) held in an area of 'scratchpad' RAM, which is memory that is not normally available to programs but is reserved for use by the operating system. The address pointed at by the vector contains a JMP instruction followed by an address, which normally will point back into the operating system. However, we can change this address to the one we want so the first instruction executed after the software interrupt will be a JMP to the entry address of the debugger. We must be careful to replace the original contents of the jump table before our program finishes executing, because it is always possible that the operating system will execute an SWI subsequently. It is worth remembering that the 6809 has three software interrupts, and there is no reason why either SWI2 (op-code 10 3F and vector at $FFF4) or SWI3 (op-code 11 3F and vector at $FFF2) should not be used — although the fact that these use two-byte op-codes makes some changes necessary in the debugger program.

A further problem is that our program can only occupy whatever memory is left free by the program we are debugging. The debugger must therefore be relocatable. You will have noticed that all references to memory locations in the program have been (or should have been) made using program counter relative addressing. The problem is that at some point we must know the absolute address of the program entry point so that we can place it in the interrupt jump table. This address must be calculated at run-time, since the assembler cannot deal with it.

Our first task then is calculating this address and inserting it into the jump table. Note that the entry point address for SWI will be different from the start address of the debugger program, because the routine at the program start address must handle this initialisation procedure, which will not be needed when we re-enter the program via SWI. Accordingly, we will handle all the initialisation within a subroutine; the entry point will then be the address containing the instruction after the BSR call to this subroutine. Very conveniently, this address is precisely the one saved on the stack by the BSR call so we can read it from the stack in order to place it at the appropriate point in the jump table.

The other job of this initialisation procedure is to obtain the start address of the program to be debugged. Here is the completed design:

INITIALISATION PROCEDURE

Data:

Vector-Address is the address to be found at $FFFA in X
JMP-Opcode is the op-code for the JMP instruction in A
Entry-Address is the address of the entry point in Y
Start-Address of the program to be debugged in D

Process:

Get Vector-Address
Store JMP-Opcode at Vector-Address
Get Entry-Address
Store it at (Vector-Address + 1)
Get Start-Address from keyboard
Save it

B	insert Breakpoint
U	Un-insert (remove breakpoint)
D	Display current breakpoints
S	Start running program
G	Go (resume from where the program left off)
R	display contents of Registers
M	inspect and change Memory location
Q	Quit

We can now return and complete the coding of the three remaining commands. A further point to consider involves one of these commands — namely command R, which displays the contents of the registers. We do not, of course, want to display the current contents of the registers while the debugger is running; instead, we want to look at the contents of the registers as they were when the breakpoint occurred. This means that we want to look at the values that were placed on the stack by the SWI instruction. However, there will be other values placed on top of these on the stack by the time we want to get at them. We could probably calculate the number of unwanted bytes on the stack and obtain the register values by discarding this amount. But a simpler solution is to