BYTE THE DUST

Our 6809 Assembly language course comes to its conclusion with this instalment. We tie up all the loose ends of our debugging program, provide an overall view of the flow of command within it, and finally code the main module itself.

The first task of the main module is to set up the interrupt mechanism, which allows us to set breakpoints in the program being debugged. These transfer control to the debugger and allow us to inspect the contents of the registers and memory locations. We must then obtain the starting address of the program being debugged so that control can be passed to it using the S command. The rest of the main routine involves getting commands from the keyboard and executing them; control is transferred to the program being debugged by the S and G commands and returned to the debugger by the SWI instructions inserted at the breakpoints.

Two stages of initialisation for this module were coded in the last instalment (see page 817). The entry point for interrupts comes immediately after the call to this subroutine. The first instruction here is to save the stack pointer, S so that it can be used to reference the values from the registers saved on the stack by the SWI. The next stage is command interpretation. We have already developed subroutines to perform all the commands, so the problem here is to select the subroutine appropriate to the command entered.

It is possible to code this as a set of nested IF statements, but we will use the fact that the Get-Command routine returns an offset into a table of command characters to perform these calls using a jump table. This is not perhaps the most efficient method in this instance, but it is a useful technique that is worth looking at. It involves setting up a table of addresses for each of the subroutines that actually carry out a command.

The JMP instruction, unlike the branch instructions, can use any of the normal addressing modes, including indexed and indirect. If we load X with the base address of the table and use the offset in B (doubled because this will be a table of 16-bit addresses, unlike the table of eight-bit command letters), then the command:

```
JMP [B,X]
```

will transfer control to the appropriate subroutine. The BSR call is made to the address of this jump instruction. As we need to set up this table in advance, it is necessary to have another stage of initialisation to carry out this operation.

PROCESS SET-UP-JUMP-TABLE

Data:
Jump-Table is a table of eight 16-bit addresses
CMDB, CMDU, etc. are the start addresses for the subroutines
Process:
For each subroutine
Get start address
Save start address in Jump-Table
Endfor

We must now consider what is to happen at the end of the run, when the quit command (Q) is issued, although there is, in fact, very little that needs doing. It makes sense to leave both the debugger and the program intact so that they can be re-entered if necessary.

The stack should be in the same situation when we exit as it was when we started. One solution would be to use a separate stack for our program by setting S to a new value and then restoring the old value. This is often a useful technique, but in our situation it may be difficult finding unused space in memory, with the debugger sitting on top of another program. Another solution is simply to increment S by the appropriate amount to lose anything that we have left there, but this is also difficult because we do not know whether or not an interrupt has occurred and the amounts on the stack will be different. The simplest solution is to save the initial value of S and restore it as the last operation of the program.

The interrupt mechanism, as set up in the initialisation procedure, stores three bytes at the address given in the SWI vector at $FFFA; we must restore this or strange results may occur if the operating system uses SWI for its own purposes. What we clearly need is a further stage of initialisation where we save these values to be restored in our quit routine.

PROCESS SAVE-VALUES

Data:
Saved is the five bytes to store the saved values
Stack-Pointer is the current value of S, plus two
SWI-Vector is found at $FFFA
Process:
Save Stack-Pointer in Saved
Get SWI-Vector
Save three bytes at SWI-Vector in Saved

The quit routine (command Q) must simply reverse this process and transfer control back to the operating system. This can be done in a number of ways: the SWI instruction itself can be used, after its vector has been reset, or a jump can be made to a known entry point in the operating