```
    and Breakpoint-Number > 0
    Decrement Breakpoint-Number
  If found then
    Uninsert-Breakpoint
```

Breakpoint-Address can be kept in Y, leaving X available to use as a pointer into the table. Breakpoint-Number can be kept in B.

Command D, to display the breakpoints, is covered by the routine labelled DISPBP (Display-Breakpoints). This is simply accessed by a subroutine branch:

```
CMDD    BRA DISPBP
```

Command S, to start running the program, is rather more complicated, since this is where breakpoints have to be inserted. The op-code for the SWI instruction must be inserted at each address in Breakpoint-Table, and the op-code that is already there is put into Removed-Values. When this has been done, control must be transferred to the start address of the program. We must also note that the next breakpoint is number 1. The full process for the start of program is:

COMMAND S

Data:

Number-Of-Breakpoints is an eight-bit value
Breakpoint-Table
Removed-Values
Breakpoint-Number is an eight-bit counter
Next-Breakpoint is an eight-bit value
SWI-Opcode is an eight-bit value
Start-Address is a 16-bit starting address for the program that we are debugging

Process:

```
Set Breakpoint-Number to Number-Of-Breakpoints
While Breakpoint-Number > 0
  Set-Up-Breakpoint (Breakpoint-Number)
  Decrement Breakpoint-Number
Endwhile
Set Next-Breakpoint to 1
Jump to Start-Address
```

For this, we use the routine Set-Up-Breakpoint — this is already coded — which requires the Breakpoint-Number (minus one, so that it can be used as an offset into the tables) in A. For convenience, we will decrement A before the call to Set-Up-Breakpoint. The full coded routine is given here.

The way that this routine ends needs a little explanation. When the program to be tested is running we do not need extra items on the stack, so we must make sure that the stack is empty when control is transferred to the program. We can clear any superfluous items off the stack in the main module, but if this routine is called by means of a BSR (to maintain consistency with other commands) the return address will have been placed on the stack. If we leave it there, then, in a long session (where the program may be restarted a number of times) the stack will keep growing. The solution we have used removes the address from the stack at the same time as control is transferred back to the program. It does this by replacing the return address on the stack by the start address. The RTS then pulls the return address, which is now the start address, off the stack, thus transferring control while resetting the stack.

The final command we will look at in this instalment is command M, to inspect and change memory locations. The idea here is to get an input address and to display the contents of that address on the screen. The user can then enter a new two-digit hex number to be placed in that location, or simply a Return. In either case, we move on to the next consecutive memory location. The user can stop the process by entering a dot. The routine GETHX2 was coded with this in mind, allowing the entry of two hex digits or a dot or a Return.

COMMAND M

Data:

Current-Location is the 16-bit address of the location being inspected
Current-Value is found in Current-Location. This is eight-bit
New-Value for the Current-Location. This is also eight-bit

Process:

```
Get Current-Location
Repeat
  Display Current-Value
  Get New-Value
  If New-Value is not a dot then
    If New-Value is not Return then
      Store New-Value in Current-Location
    Endif
    Increment Current-Location
    Display Current-Location
  Endif
Until Current-Location is a dot
```

For this command routine, Current-Location is stored in X, and the B register is used for both Current-Value and New-Value. A is used as a flag to indicate which of the three possibilities (hex number, dot or Return) was entered.

We have now to design and code three remaining commands — G, R and Q. However, these involve the use of an interrupt mechanism, which we have yet to look at. This, and the designing of the main module for the debugging program, are the subject of the next instalment.