BUG REPELLENT

To illustrate the techniques of the top-down design approach to Assembly language programming, we now begin to build up a debugging program. The first thing we must do is develop the control module, which has overall command of the lower-level modules that perform more specific activities.

We shall first take a brief look at the specification and design stages for the production of a debugger. The specification is reasonably straightforward; we have already looked at the functions we would expect such a program to provide (see page 739).

The inputs to the debugger will be:

1. A program to debug: We will assume that the debugger is loaded with the program it is to debug already in memory.
2. Commands: We must decide whether the commands are to be entered directly or as choices from a menu. We will enter single-character commands from the list given in the margin.
3. Addresses: These would presumably be entered in hex, so it will be necessary to convert a string of ASCII hex digits to a 16-bit binary number.

The outputs from the debugger will be:

1. 'Echoes' of input characters: Remember that keypresses do not automatically generate characters on the screen — the computer must be programmed to do this (this is called 'echoing').
2. Eight- and 16-bit numbers: These are accepted as strings of hex digits.
3. Strings: These are used to label the above.

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

There are many ways in which a program could be split up into modules and then into subroutines, but there must always be an outer module — the 'shell' — which ties all the others together. For our debugger program, this will take the form:

THE MAIN MODULE

Data:

Start-Address of program (16-bit)
Prompt for command entry (single ASCII character '>')
Command Character is a single ASCII character (do we allow lower-case characters?)
Break-Address is the address of the handler routine that services the SWI interrupt

Process:

```
Set up Interrupt
GET Start-Address
REPEAT
   DISPLAY Prompt
   REPEAT
      Get Command
   UNTIL Command is valid
   DISPLAY (Echo) Command
   IF Command = 'B' THEN
      Insert-Breakpoint
   ELSE IF Command = 'U' THEN
      Remove-Breakpoint
   ELSE IF...
Until Command = 'Q'
```

End of Main Module

From this we now have a good idea of the routines that will be required. A module is not the same thing as a subroutine, however. Clearly, there are several subroutines that logically go together in groups with shared data — one such module, for example, might deal with breakpoints. The next stage of refinement shows how we might design such a module:

MODULE BREAKPOINTS

Data:

Breakpoint-Table is an array of 16-bit addresses where breakpoint addresses can be stored
Removed-Values is an array of eight-bit values corresponding to the above table. The op-codes that get replaced by an SWI instruction at the breakpoint can be stored in this
Number-Of-Breakpoints is an eight-bit value containing the number of active breakpoints
is an eight-bit value, which contains the next breakpoint that will be encountered in the run
SWI-Opcode is an eight-bit op-code for the SWI instruction

Process1: Insert-Breakpoint

```
IF Number-Of-Breakpoints < MAX THEN
   Get-Address
   Add 1 to Number-Of-Breakpoints
   Store Address in Breakpoint-Table
   (Number-Of-Breakpoints)
ENDIF
```

End Of Process1

Process2: Set-Up-Breakpoint(N)

```
(N tells us which of the breakpoints in the table is to
   be set up)
Get-Address in Breakpoint-Table(N)
Get Op-code at that Address
Store it in Removed-Values(N)
Store SWI-Opcode at Address
```

End of Process2

Process2 is at the stage where we could begin coding it. There are four data values that must be manipulated: N, the parameter that tells us which breakpoint to use, is an eight-bit number in the