code column. Because DW automatically converts its operands into lo-hi form, it is most often used to initialise 'pointer' locations with addresses. LABL2, or location $D3A1, might be such an address — it points to location $98CE.

The third thing to consider is that the instruction DS $10 has the effect of adding $10 to the program counter. This is clearer in the symbol table than in the actual listing — LABL3 represents the location $D3A3 (the location following the previous instruction), though it appears from the listing that its value is $D3B3. This is actually the location address of the next instruction after the DS instruction, so DS $10 has reserved a block of 16 bytes (from $D3A3 to $D3B2 inclusive) between one instruction and the next. This is a process rather like putting long REM lines into a BASIC program to create unused space in the program text area that can then be POKEd and PEEKed as a machine code program area (see page 137).

Finally, the last instruction uses EQU to set one symbol equal to the value of another, so that DATA1 has the value $D3A3 (the value of LABL3). This is another source of possible confusion. LABL3 is the symbolic representation of the location address $D3A3, so DATA1 EQU LABL3 means 'the symbol DATA1 is to have the same meaning and value as the symbol LABL3'. The fact that the DB instruction has made the contents of $D3A3 equal to $5F has no significance for the meaning of the symbols LABL3 and DATA1. Keeping the distinction between a location and its contents clear in your mind is one of the most testing difficulties in the early stages of learning Assembly language programming. You may have had the same problem with BASIC program variables and their contents.

At first glance, the DB directive seems to duplicate EQU, but this is not the case. LABL1 means 'the location $D3A0', and DB $5F has initialised that byte with the value $5F, but, although the value of LABL1 is now fixed, the contents of the location it symbolises can be changed at any time (by storing the accumulator contents there later in the program, for example). Similarly, DATA1 is now a symbol whose value is fixed by the EQU instruction; its value cannot be changed by the program's execution. And again, LABL3 points to the start of a 16-byte data area, the contents of which can be changed in the program, but LABL3 is itself unchangeable.

This introduces, but does not exhaust, the possibilities of the new pseudo-ops. Consider this new version of the previous fragment:

```
                        ORG   $D3A0
D3A0   4D4553   LABL1   DB    'MESSAGE 1'
D3A9   CE98     LABL2   DW    $98CE
D3BB            LABL3   DS    $10
D3BB            DATA1   EQU   LABL3

SYMBOL TABLE:
LABL1 = D3A0: LABL2 = D3A9: LABL3 = D3AB
DATA1 = D3AB
ASSEMBLY COMPLETE — NO ERRORS
```

The DB instruction has a string, 'MESSAGE 1', as its operand, and the assembler has initialised the locations from $D3A0 to $D3A8 with the ASCII values of the characters within the single quotes. This can be inferred from inspection of the location address column in the listing, and is partly confirmed by the machine code column — the contents of the three bytes from $D3A0 to $D3A2 are shown to be $4D, $45, and $53, which are the hex ASCII codes for 'M', 'E', and 'S'.

This is a significant facility, not only because it relieves the programmer of the task of translating messages and character data into lists of ASCII codes, but also because it makes the listing much easier to read, and hints at the possibility of actually getting some screen output from our Assembly language programs. The latter is particularly significant because so far we have been restricted to storing results in memory and inspecting them using the Monitor program (see page 118). Naturally, we will be exploring screen-handling in the course, but there are still aspects of Assembly language that we need to investigate before going onto that topic. If, however, you think about our habit of storing results in memory, and if you understand already that memory-mapped screen displays are, in effect, only areas of memory, then you may be able to see a way of addressing the screen from a program.

The most important aspect of this new DB facility is that it confers on LABL1 the status of a

Exercises

1) The first program fragment in the main text uses the DS pseudo-op to reserve $10 bytes of memory starting from the address represented by the label LABL1. Write an Assembly language program that will store the numbers $0F to $00 in descending order in this block, one number per byte. This should be done using a loop, and indexed addressing techniques, for which you will need to use the DEX (decrement the X register) or DEC (IX+0) (decrement IX) instructions. The loop should continue as long as decrementing the index register does not cause the zero flag to be set, so use the BNE or JR NZ branch instructions.

2) Using the techniques of the previous exercise, write a program to copy the message stored at LABL1 by the DB pseudo-op (see the second program fragment in the main text) to a block of memory starting at the address stored at LABL2 by the DW pseudo-op. The address $98CE may not be suitable for your computer, so change the initialisation, but the program should work for any address, and for any length of message. To implement this, your program must use either the number of characters in the message as a loop counter, or it must be able to recognise the end of the message — you might put an asterisk, for example, as the last character of any message.