need to consider, so BTM could be set to MID. Slightly more efficient, however, is to set BTM to MID + 1, since we already know that ARRAY(MID) is not equal to the search key. Similarly, IF ARRAY(MID) > SCHKEY$, TOP may be set to MID - 1.

As an interim step towards developing a fully working routine, the program shown can take a dummy input (which needs to be in exactly the same format as the MODFLD$ fields) and will either print RECORD NOT FOUND if there is no match, or RECORD IS NO (MID) if there is a match. As the routine starts with line number 13000, it can be added on to the end of the program as presented on page 399, and will work as long as line 4040 is changed to IF CHOI = 1 THEN GOSUB 13000.

Line 13240 contains the STOP statement. This will stop the program temporarily as soon as the RECORD NOT FOUND or RECORD IS NO (MID) messages are displayed. The program can be re-started at the same line number, without losing data, by typing CONT. Without STOP, the program would rush on to the RETURN statement in line 13250 and the message would appear too briefly to be legible.

Let's consider this program fragment in more detail. Line 13100 sets BTM to 1, the position of the lowest element in the MODFLD$ array. TOP is set to SIZE-1 in line 13110. This is the position in the MODFLD$ arrays where the highest element is located. Line 13120 initialises a loop that will only be terminated when either a match is found or no match is known to exist.

Line 13130 finds the mid point of the array by halving the sum of the bottom and top index of the array (INT is used to round off the division, so that MID cannot assume a value such as 1.5). There's a chance that the contents of MODFLD$(MID) will be the same as the search key (SCHKEY$), but if they are not the same, as is likely, L will be set to 0, ensuring that the loop will be repeated. If the test in line 13140 fails, MODFLD$(MID) will either be lower or higher in value than SCHKEY$. The value of BTM will then be set to one more than the old value of MID (line 13150), or the value of TOP will be set to one less than the old value of MID. The reason the value of MID itself is not used is that the failure of the test in line 13140 has already demonstrated that MODFLD$(MID) is not the target we are searching for and there is no point in looking at that element of the array next time round the loop.

If no match is found, the value of BTM will eventually exceed the value of TOP. The loop can be terminated (line 13170) and a RECORD NOT FOUND message printed (line 13200).

This program fragment is presented for demonstration purposes and to enable the search routine to be tested. As it stands, its use is rather limited. Without the STOP in line 13240 we wouldn't even have time to see the message flashed on the screen. What is required is a display of the full record, as it was originally typed in. Once the record number is known, it is a simple matter to retrieve any of the additional information required — NAMFLD$, STRFLD$ etc. Below the display of the record, we would probably want a message such as PRESS SPACE BAR TO CONTINUE (back to the main menu) and perhaps further options such as PRESS "P" TO PRINT.

Not so easy, unfortunately, is deciding how to handle the input of *FNDREC*. In the program fragment, the input expected (in line 13020) must be in the standardised form — JONES PETER, for example. This is clearly not good enough. People don't think of names in inverse order, and it's an unreasonable burden on the user to have to enter the name in upper case letters. Additionally, the slightest deviation between the name input originally would result in a RECORD NOT FOUND. The first two problems could, one would expect, be handled by *MODNAM*. The third problem of how to cope with an approximate match is far more interesting, but very much harder to solve.

Before considering this problem, let us see why *MODNAM* will not solve the first two problems. If you go back and look at *MODNAM*, which starts at line 10200, you will discover a good illustration of one of the commonest traps into which programmers fall — lack of generality. This subroutine ought to be able to handle conversions from 'normal' names to 'standardised' names whenever this operation was needed. Even though it was written as a separate subroutine, it was clearly written with *ADDREC* in mind. It assumes that the name to be converted will always reside in NAMFLD$(SIZE) and that after conversion the modified name will always be stored in MODFLD$(SIZE). Faced with this situation, the programmer has three choices: either completely rewrite *MODNAM* to make it general, which would in turn involve further changes in other parts of the program. Or write an almost identical routine just to handle the input for *FNDREC*, which represents wasted effort and takes up more space in memory. Or resort to some bad programming technique to allow the unmodified *MODNAM* routine to be used. This last alternative is in some ways the least attractive. It will solve the problem, but the actual working of the part of the program that has been modified is likely to be unclear, even to the writer of the program, and a nightmare to anyone else trying to use the program.

The moral of the story is: make subroutines as general as possible, so that they can be called by any part of the program.

To illustrate bad programming technique, or 'dirty' programming as it is often called, and to show how unclear it can make the program, consider line 13020 of the program fragment, INPUT "INPUT KEY ";SCHKEY$ and then look at the modification or 'fix' that would allow *MODNAM* to be used:

```
13020 INPUT "INPUT KEY";NAMFLD$(SIZE)
13030 GOSUB 10200: REM *MODNAM*
   SUBROUTINE
13040 LET SCHKEY$ = MODFLD$(SIZE)
13050...
```