bound to pass) and the subroutine RETURNs to the main program skipping the rest of the *CHOOSE* subroutine since it is inappropriate.

You may have wondered why TEST$ is tested twice. This is to prevent the subroutine RETURNing to the wrong point in the program. Without line 3530, the program would continue on down the rest of *CHOOSE*, presenting the choice menu even though it is not needed. It also avoids the use of GOTOs, though IF TEST$ = "@FIRST" THEN GOTO 3850 would work just as well. GOTOs make the program messy and difficult to follow (programs making excessive use of GOTOs are referred to as 'spaghetti coding').

Before going on to look at *FIRSTM*, readers are referred back to *RDINFL* and the GOTO in line 1430. Since we have consistently argued against using GOTO, why has one been used here? It would have been perfectly easy to CLOSE the file and RETURN by simply testing the value of TEST$ in two separate lines. We used a GOTO here instead to illustrate one of the few instances where its use is excusable. This is within a very short and identifiable program segment, and its function is obvious (and made more so by the REM comment). GOTOs should never be used to jump out of a loop (this can leave the value of variables in an unpredictable state), never used to jump out of a subroutine (this will confuse the RETURN instruction unless a matching jump back into the subroutine is used), and never used to jump to remote regions of the program (this makes the program all but impossible to follow).

The *FIRSTM* subroutine is simple and straightforward: the screen is cleared and a message is displayed informing the user that a record will have to be entered. Line 3870 sets CHOI to 6 so that when control is passed back to *EXECUT* the *ADDREC* routine will be executed automatically. The code for *FIRSTM* follows:

```
3860 REM  *FIRSTM* SUBROUTINE (DISPLAY
     MESSAGE)
3870 LET CHOI = 6
3880 PRINT CHR$(12): REM  CLEAR SCREEN
3890 PRINT
3900 PRINT TAB(8);"THERE ARE NO RECORDS IN"
3910 PRINT TAB(8);"THE FILE. YOU WILL HAVE"
3920 PRINT TAB(6);"TO START BY ADDING A
     RECORD"
3930 PRINT
3940 PRINT TAB(5);"(PRESS SPACE-BAR TO
     CONTINUE)"
3950 FOR B = 1 TO 1
3960 IF INKEY$ <> " " THEN B = 0
3970 NEXT B
3980 PRINT CHR$(12): REM CLEAR SCREEN
3990 RETURN
```

The *ADDREC* subroutine, given on page 379, has two small but important changes from the version we encountered before. After the fields have been entered as elements in the various string arrays, the variable SIZE is incremented and TEST$ is set to a null string (see lines 10090 and 10100). SIZE is an important variable used in various parts of the program so that it knows which records are being operated on. SIZE was originally set to 0 as part of the *CREARR* subroutine. Later, in *SETFLG*, it is set to 1 if TEST$ = "@FIRST". This is done so that when *ADDREC* is first executed, the INPUT statements will put the data into the first element of each array. In other words, INPUT "ENTER NAME";NAMFLD$(SIZE) is equivalent to INPUT "ENTER NAME";NAMFLD$(1).

Line 10090 increments SIZE, so that it now becomes 2. If *ADDREC* is executed again, data will be entered into the second element of each array. Finally, *ADDREC* sets TEST$ to " " in line 10100. This is done because a record has now been entered (though not yet stored in the tape or disk data file). If *CHOOSE* is executed again, as it must be to save the data and exit the program, we will not want to be forced to add a new record again. If TEST$ were not cleared, the program would get stuck in an endless loop, and the only way to get out of it would be to reset or unplug the computer, and all the data would be lost.

By setting TEST$ to a null string, the tests in lines 3520 and 3530 of *CHOOSE* will fail and allow the options menu to be displayed. What then happens to SIZE will depend on which routine is executed. So far we have only ensured that SIZE = 1 if there is no valid data in the file, and that this is incremented by 1 each time a record is added. But what would happen if there had been a number of valid records in the file? To answer this we'll have to look at *RDINFL* again.

Line 1420 reads the first data item into TEST$. If it is not @FIRST, it is assumed to be a valid data item. The records in the file are always in the same order, namely: NAMFLD, MODFLD, STRFLD, TWNFLD, CNTFLD, TELFLD, NDXFLD, NAMFLD, MODFLD, etc. If the first record read out is valid data, it must belong in the first element of the NAMFLD$ array, so line 1440 transfers this data from TEST$ to NAMFLD$(1). The next two lines fill up the first elements in the other five arrays. We now know that we have at least one complete (database) record, so SIZE is set to 2. This value must be one greater than the number of valid records read into the arrays, otherwise *ADDREC* would write new data into elements already containing valid data.

Then a loop from 2 to 50 reads the records into all six arrays, incrementing the index L each time round. We have already made the decision to restrict our program to dealing with files of 50 names and addresses, and the DIM statements in the *CREARR* subroutine allocated space for this. However, when you first start using the program, you are unlikely to have a complete file of 50 entries, so we will need a routine in the program that can detect when this is the case, set the variable SIZE accordingly, and abort the reading-in loop.

Consequently, we have included line 1510 to provide a call to a 'SIZE' subroutine, which we will be developing later in the course. There are three ways in which this problem could be handled. First, when we write the data to tape, we could arrange that the first record to be written is the variable SIZE. The *RDINFL* subroutine could then be modified to read in SIZE first and then set up a loop of the form FOR L=1 TO SIZE to read in the records. The second, and preferable, method