real address book expects you to 'PRESS 1' to find a name. Nevertheless, a good user image involves well designed screen layouts and a consistent pattern to the operations. Prompts should always appear in the same position on the screen (some well known word processors, for example, display some prompts on the top line of the screen and some on the bottom line, apparently at random). A program with a good user image will also inform the user at any time where he is in the program. If you are in the ADDREC mode, there should be a message always visible to tell you this. If you have just entered a field (to add to a new record), there should be a message to say, for example, HIT RETURN IF ENTRY IS CORRECT, ELSE HIT ESCAPE (which brings us to the important subject of error recovery and reporting, which we will come to later).

Ideally, all formatting should appear on the screen, so that, for example, the record displayed on the screen will be of the same format as a record printed out by the printer. Many commercially available pieces of software incorporate 'help menus' which will tell you what to do next if you're not sure.

The user image of a program is best when it is concrete — a piece of typing paper or an index card — rather than abstract with 'sub-files', 'buffers' and so on. Many commercial database programs suffer in this respect; the user has constantly to keep in mind that certain pieces of information are in sub-files or temporary, hidden fields. These factors tend to make the use of such a program more of an intellectual burden.

Error recovery is also an important subject. What happens, for example, if you have just entered someone's name but realise that you made a typing error? Will you have to go ahead and then call MODREC to correct it, or will the program give you the option to 'quit' before you go any further? Most versions of BASIC will report errors in the entering of a program, either when the erroneous line is entered, or when the program is run. However, this is not part of the 'user interface'. BASIC does include a number of messages that re-prompt the user for a correct entry if an incorrect one is made (for example, the REDO prompt if an unsuitable entry is made in response to an INPUT statement).

Handling errors has two facets — error reporting and error recovery. One well known word processing program, for example, has good error reporting but poor error recovery; if you create a long document and try to save it on a disk that is already nearly full, the program gives you the helpful message DISK SPACE EXHAUSTED. It does not, unfortunately, allow the user to recover from this error — a new disk cannot be formatted without first destroying the text that you may have spent hours typing in!

Any operation performed by the user that could result in the loss of data (MODREC, for example) should always be queried before execution. Messages such as THIS WILL DESTROY THE RECORD. ARE YOU SURE? (Y/N) should always be provided. In a word processor, a similar message would be: THE 'SAVE' COMMAND WILL NOT KEEP A BACKUP OF THE OLD DOCUMENT. IS THIS OK? (Y/N).

Error handling (trapping and reporting) should be considered in the design of a program wherever there is a possibility of wrong data input, wrong menu choice, wrong commands and whenever data is to be modified or saved, especially if the save involves writing over old data.

You must pay attention to security — what happens to the program or data if there is a fatal error (such as a power failure). The program designer should consider how much data it is possible to lose and devise methods of recovering as much as possible or making the remaining data usable. One rather sophisticated word processor incorporates a program called RECOVER so that if there is a catastrophic error (a power failure, for example, or switching off the computer before saving the document), almost nothing is lost. Such advanced programming techniques, sadly, are beyond the scope of this course. The point is, though, to make your programs as secure as possible by anticipating all possible fatal errors beforehand (that can be reasonably dealt with), and writing routines designed to cope with them.

Adaptability, the ease with which the program can be customised, is also important. We have touched on this topic a number of times already. At the simplest level, always leave plenty of room between line numbers (in BASIC) and incorporate plenty of empty REMs that can be filled later with statements and GOSUBs if necessary. When creating arrays, at least one redundant array should be built in to allow for future expansion. It is a cardinal rule of program writing that future requirements cannot be anticipated. The only thing certain is that a good program can always be made better, and making it better is likely to mean writing more code.

Basic Flavours

ZX81 SPECTRUM

See the program on page 318 for the Spectrum version of lines 1300 to 1370, and the Basic Flavours box for the ZX81 version. On the Spectrum, lines 3750 to 3780 may cause problems because keys repeat if held down for more than a fraction of a second. The Spectrum handbook recommends that this type of INKEY$ loop contain an extra line to avoid this:

```
3755 IF INKEY$ <> "" THEN GOTO 3755
```

Sinclair BASIC supports VAL(A$), but will crash the program if the first character of A$ is non-numeric; in this program the problem can be avoided by:

```
3790 LET CHOI=CODE A$-48
```

but this is not a complete solution — it works only when A$ is a single character (as it must be in the program). Sinclair BASIC does not support ON...GOSUB, but it does allow you to write GOSUB (numeric expression) as well as simply GOSUB (line number); line 4010, therefore, may be replaced by