The Technology of the Quad Editor

by
Jim Kramer
Hewlett-Packard Co.
St. Louis, Missouri

I. Introduction

The Quad editor is a simple text editor that is being contributed to the Users Group library at these meetings. It has several features which make it notable and useful, the most important of which are that it texts files instantaneously and that it can undo any or all editing changes. The purpose of this paper is to explain the technology behind these and other features.

II. A Brief History and Description of Quad

Quad was created to avoid the high overhead of Edit/3000's text and keep commands, which are essentially file copy operations.

Early versions of Quad simply opened the texted file and made changes directly to it. This precluded the possibility of adding or deleting lines from the file. It also made editing a somewhat risky business, since changes were being made directly to the file rather than to a work file copy ("QUAD" originally meant QUick And Dirty). Nonetheless Quad was very fast for looking at files and making simple changes.

The current version of Quad retains most of the advantages of the early versions and eliminates the main deficiencies. A file is still texted just by opening it, so that the overhead of a file copy is eliminated. However changes are kept in a work file, so that a user is not committed to them until he does a keep of the file.

Having separate text and work files requires Quad to logically merge the two to give the user the illusion of working on a single file. However it also makes it possible to cancel any and all changes just by removing them from the work file; Quad's undo command returns all lines within a specified range to their original state.

In general Quad must, when keeping the edited file, create a new file which merges the text and work files. However if the keep is back to the texted file and no lines have been added or deleted, then the keep is done just by updating existing records in the text file. Thus whenever possible the file copy on keeping is eliminated also.

It is important that Quad be able to find lines in the texted file quickly. Quad starts out with no knowledge of the location of lines in the file, and must find requested lines using binary search. However Quad keeps a record of all blocks read during the search process and uses this record to shorten subsequent searches. The method is described in a paper titled "A New Tool for Keyed File Access (Sometimes)" in the proceedings of the Users Group's 1980 North American meeting.

III. The Work File

Quad creates a work file only when the user first makes a change to the texted file. This can be any type of change -- adding or deleting a line or modifying an existing line.

The work file is a keyed file which allows both keys and data to be variable length. It can contain two types of entries -- deletes and changes.

Quad allows deletion of a range of records -- for example "D 2/9" is a command which deletes line numbers 2 through 9. To do the deletion, Quad makes a single entry in its work file as follows:

D0000200000009000

This is just a 17 character key. The first character is a "D" (for delete), the next 8 characters are the lower line number in the range, and the last 8 are the upper line number

Since deletion is achieved with a single work file entry, it is very fast, and the speed is independent of the number of lines being deleted.

Now suppose that the command was given to delete records 8 through 14. This command would normally result in an entry of D8/14 into the work file. However this range overlaps an already existing delete range of D2/9, so that Quad would combine the two into a single entry of D2/14.

If the user now undid changes over the interval from 5 to 8, it would be necessary to "undelete" over this range. Therefore the entry D2/14 would have to be split into two -- D2/4.999 and D8.001/14.

The other type of entry in the work file is a change entry. There is a change entry for every line added during editing and every line modified.

A change entry consists of both a key and data. The key is just the letter "C" followed by the 8 character line number, and the data is the line of text corresponding to that line number.

IV. The Structure of the Work File

The work file used by Quad was originally designed for another purpose -- a different editor. The desire was for a file access method which gave random access to variable length records, and re-used space from deleted records.

The solution is a file access method which I call ticket files.

With most file access methods, the user who wants data stored specifies where it is to be stored -- a record number. With ticket files the user does not specify; instead he just supplies the data to the access method and receives back a "ticket" telling him where the data has been stored. In order to retrieve the data, he just supplies the ticket.

It is important to recognise that this technique gives enormous flexibility to the file access manager. The data can be put in the most convenient spot, for example a block that is already in a buffer in main memory. Within the block the record can be placed wherever there is space. With ticket files a record need not even be placed contiguously within the block -- it can be broken into pieces.

Although ticket files might at first seem unnatural or even clumsy, they turn out to be perfectly suited to those applications in which data is found through pointers; tickets are really just pointers.

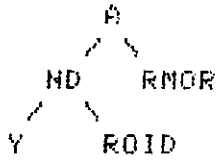
Image detail data sets are an example. If we neglect serial access, a detail data entry is found through pointers which are stored in other detail entries or in a master entry. Ticket files could in fact be used to implement Image details and would relieve Image of the burden of keeping track of free space. Further they would add a capability that Image details currently do not have -- variable length records.

KSAM files are another example -- all data in a KSAM file can be found through pointers if only the location of the root block in the key file is known. Ticket files will, by the way, remember one ticket for the user.

In order to make ticket files satisfactory as work files, it was necessary to implement a keyed sequential access method based on ticket files. The implementation is significantly different from KSAM and actually more powerful: both keys and data can be variable length, space is re-used, and keyed sequential access can be either forward or backward.

The keys are stored in a tree-structure called a trie. In a trie a key value is actually distributed through various levels of the tree structure; branching occurs when two keys which are identical for some number of beginning characters first differ.

For example the keys "ANDY, "ARMOR", and "ANDROID" would result in the following structure:



Generally this structure will have more levels than KSAM's B tree. On the other hand its structure is a function of the data itself, not of the order in which the data is loaded. Therefore tree re-organization is never necessary during loading, whereas with KSAM it usually is.

When a key is stored, a ticket is stored with it. The ticket points to data. Thus storing data by key is a two-step process:

1. Store the data and receive a ticket.

2. Store the key and the ticket.

Retrieving data by key reverses the two steps:

1. Supply the key and receive the associated ticket.

2. Use the ticket to retrieve the associated data.

V. The Help Facility

From the start I wanted Quad to be a single program file not requiring any auxiliary message or documentation files. The reason is that I wanted the acquisition and use of Quad to be as simple and foolproof as possible.

This meant that Quad had to have a very good help facility -- a built-in manual. Quad does in fact now have quite an extensive menu-driven help facility. All of its text can be printed offline to make an adequate user manual.

A help facility presents some technical problems. In the first place it should be very easy for the programmer to write the help text. Moreover the text must be prevented from making the program enormous: text takes up program space very quickly. Therefore blank compression is very desirable.

The solution adopted in early versions of Quad was simple but not fully satisfactory. A line would be written to the screen by

doing an SPL move of a literal to a buffer followed by a call to a procedure to write the buffer to the screen. By using SPL defines it was possible to make the code to write a line look like the line itself bordered on both sides by a bit of auxiliary text. Thus the code

```
BQQ "This is a line to go to the screen" EQQ
```

was shorthand for the code:

```
MOVE MSG:=("This is a line to go to the screen",0);
WRITE'TO'SCREEN(MSG);
```

This made help text very easy to write. Unfortunately it also wasted code space. Each line to be written required its own move and call, and there was no blank compression (although trailing blanks could be suppressed).

An efficient solution to save code space is to have each screen of help text stored in a PB-relative array in a compressed form. (A PB-relative array is an SPL array which is part of a code segment). The procedure containing the array would fetch the text line by line in a loop and call another procedure to write each line to the screen. In this way there is only one set of move and call code for the entire screen rather than one set per line.

The problem is that it is very difficult to write code that initializes arrays with blank-compressed text. It would be far better to just write:

```
(*    Help'proc : Help'seg      *)

  This is a block of help text, which we would like to be
  converted to a space efficient procedure named Help'proc
  (for the segment Help'seg)

(**)
```

The solution of course is to have a program which converts such blocks to the desired procedures. Quad's help facility was written using such a program which served as a pre-processor to the SPL compiler.

VI. The Command Interpreter

There were two main objectives in the implementation of Quad's command interpreter:

1. To catch as many errors as possible during command analysis, rather than command execution.

2. To emit the most meaningful possible error messages.

As an example, editors must have commands which include file names. The editor could be designed do no checking on the file name; any errors will be detected later on by the file system. Quad, however, assures that the name is syntactically valid first. For example it checks that there are no more than three parts to the name (file, group and account), that each part has between 1 and 8 alphanumeric characters with the first being alpha, that there is no more than one lockword, that the lockword follows the file part, etc. Any error found results in a message which describes that particular error.

One aspect of emitting meaningful error messages is to point out where in the command the error occurred. To this end Quad points to the error, just as MPE does for command errors.