To: MTB Distribution

From: Gary C. Dixon

Date: July 31, 1978

Subject: Goals and Policies of a New Editor

This MTB attempts to take a step backward in the process of designing a new editor. I have long felt that the goals of a new editor are not well understood by many people interested in its design. This MTB attempts to state boldly the goals which have been implied in previous editor MTBs (MTB-334, MTB-339 and MTB-368).

Although work on a new editor has been tabled for the present, I have gathered together my thoughts on editor objectives, goals and policies for reference in future editor design projects.

If you disagree with the goals or policies stated in this MTB, or if you can suggest further goals or policies, feel free to comment by Multics mail to:

GDixon.Multics (System M)

POTENTIAL USERS OF A NEW EDITOR

We begin by defining the potential users of the new editor, for it is their needs which motivate the design of a new editor.

* secretaries, typists and other potential Multics users
  - who are unfamiliar with computers
  - who are unfamiliar with timesharing and online editing
  - who need a simple, easy to teach/learn/use editor which provides a moderate set of editing functions, and full protection of the novice user from his mistakes

* beginning programmers
  - who are unfamiliar with Multics and with timesharing and online editing
  - who need a simple, easy to teach/learn/use editor which provides a moderate set of editing functions, and some protection from mistakes made by the user

* intermediate programmers and typists
  - who have used editors to some extent
  - who have a lot of editing to do and are concerned about amount of work involved in editing and total time editing will take
  - who need a fairly sophisticated editor which provides complete editing functions and simple edit programming functions

* sophisticated programmers
  - who are skilled in use of editors and writing of editing programs (request files)
  - who want an editor which is easy and fast to use, and which provides a comprehensive set of editing and editor programming functions

Multicians have much experience with intermediate and sophisticated programmers so they understand the needs of these users fairly well. However, we have not had much experience with secretarial users, or beginning programmers and typists. Consequently the needs of these users are not well understood. Suggestions about what such users need/want (or don't need/want) in an editor would be welcomed.

GOALS OF A NEW EDITOR

To meet the needs of the potential users listed above, a new editor must meet the following goals.

Goal 1. The editor should provide a user interface which is adaptable (tailored) to the needs of each class of user named above.

Goal 2. Basic editing should be easy to teach and to learn. The time required for a novice to begin practical editing tasks should be minimized.

Goal 3. The editor should provide functional editing and edit programming capabilities comparable to those available in edm, qedx, ted and teco.(1)

Goal 4. The editor should by as easy, fast and safe to use as possible.

Goal 5. The editor should provide reliable editing services.

Goal 6. The editor should be as efficient as possible, to have the smallest possible impact on total system load.

Goal 7. The editor should take full advantage of the hardware and software features of Multics in performing its functions.

Goal 8. The editor should be designed and implemented in a way which permits/encourages functional extension of editing capabilities as new editing needs are identified. It should be easy/fun to add new features to the editor.

Goal 9. The editor should interface well with the existing Multics command environment, and incorporate the various features of this environment in its functional capabilities. In other words, it should be well integrated with existing Multics features and facilities.

Goal 10. The editor should provide a subroutine interface, to allow it to be used within other subsystems (such as the mail command, etc).

Goal 11. The basic editor interface should manipulate only single-segment, stream-oriented files.

---

(1) The interfaces to these capabilities will probably differ in any new editor, especially in cursor-controlled editors which provide a different editing environment to the user.

Goal 12. To the extent possible, it should be easy to change from one of the existing Multics editors (qedx, ted, teco) to the new editor. Some retraining of habit patterns may be required, but it should be possible to accomplish such retraining within one month's editing usage.

Little attempt has been made to justify the goals stated above. Hopefully, justification for them is self-evident. However, it is important that all parties interested in the design of a new editor agree on the goals of that editor before undertaking the design process.

CURSOR-CONTROLLED EDITING VERSUS TTY-ORIENTED EDITING

In recent years, the attention of the timesharing industry has been directed to inexpensive (to build, purchase and operate) video terminals (also called CRT terminals, scope terminals, TV terminals, etc). While some of these terminals have a true graphics capability with operations for drawing vectors and conic sections on the screen, the least expensive models provide only ASCII text display capabilities. Characteristics and features of these terminals vary.

1. Most use TV type screens that must be constantly refreshed from a local character buffer. Some use static storage tube screens that involve no local buffer but which cannot be dynamically updated without erasing and redisplaying the entire screen.

2. Displays come in a variety of sizes. Some terminals can display only four 32-character lines, while others display forty-five or more 80-character lines.

3. Most provide a cursor which indicates where the next letter will appear when the user types or the computer prints. Usually, function keys are provided to move the cursor to the right or left by one character, and up or down by one line. Movement of the cursor causes special cursor move characters to be transmitted by the terminal to the system in some terminals. In other terminals, cursor movement is a local function hidden from the system.

4. Some terminals transmit each character to the system as it is typed. Most TTY terminals do this, and such terminals are identified as a class by the name video TTY terminals (vtty).

   Others store characters in their local buffer memory as they are typed, and transmit only when a special TRANSMIT key is typed. Such terminals are referred to as video information processing terminals (vip). TRANSMIT causes some vip terminals to send only a line of data, specifically the line which the cursor points to. In some cases, only the characters which precede the cursor on that line are transmitted. TRANSMIT causes other vip terminals to transmit the entire screen (the entire local buffer). This allows the user to use local editing functions provided by the terminal to properly shape the data before it is transmitted to the system. TRANSMIT causes still other terminals to transmit only the data fields which have been modified since the system last printed the screen contents. This feature allows the system to print a form on the screen which the user will fill in. TRANSMIT causes still other terminals to send all data fields on the screen which have not been protected by system-controlled protecting characters included by the system when the screen was originally printed. Many terminals provide several of these transmission techniques, allowing the user or the system to choose which is most appropriate for the current application.

5. Some of these terminals provide special function keys (besides ESC and CTRL) which are mapped into ASCII control characters. Some even allow the user to specify what character or character sequence is to be transmitted when a particular function key is typed.

With such a variety of video terminals in the marketplace, and especially with the increasing word processing marketplace, it behooves us to consider making the new editor a cursor-controlled editor.

Experiments with a variety of cursor-controlled editors indicate that cursor-controlled editing is often faster than tty-oriented, request-controlled editing. Also, cursor-controlled editing is usually easier to learn than tty-oriented editing.

However, the developer of a cursor-controlled editor faces significant difficulties. The most obvious is that cursor-controlled editing is usually designed to be done on vtty terminals. Assuming this is true for a Multics cursor editor, a significant portion of the existing Multics community (having hardcopy terminals) would not be able to gain the full advantages of cursor editing. Thus, development of a cursor-controlled editor would not fulfill the goals hardcopy users have for a new Multics editor.

Second, too little is known about the variety of video terminals out in the marketplace, or even about the video terminals used by our current customers. Significant terminal research would have to be carried out before we could begin to design an editor which would match the characteristics of most terminals.

Third (and even more significant), we Multicians do not have enough experience with cursor-controlled editing to design a good editor (or even to judge such a design). Significant research into existing cursor-controlled editors(1) is required before we try to design our own. Otherwise, we may miss the boat in the marketplace, or worse, design a cumbersome editor which is difficult to learn and use.

It should be clear from the discussion above that it will be several years before we can provide a demonstratably excellent, cursor-controlled editor. However, we must have an editor which is simpler than qedx and more powerful than edm in the near future to meet the needs of the editor users listed above.

I propose that we continue the research into cursor-controlled editing to gain experience and user feedback with experimental cursor editors; at the same time, we should take advantage of our existing expertise in TTY-oriented editing by developing a simpler, enhanced TTY-oriented editor.

The remaining sections of this MTB expand on some of the editor goals stated above, based upon experiments with a variety of TTY-oriented editors. A series of policies are outlined which might be followed to help achieve some of these goals. While these policies were envisioned in the context of a TTY-oriented editor, many of them could also be applied to the development of a cursor editor.

---

(1) An example of such research is Bernie Greenberg's experimental EMACS editor which is a Multics implementation of an existing cursor-controlled editor developed by the MIT Artificial Intelligence Laboratory. The Multics EMACS experiment is described in MTB-373.

CONCEPT-ORIENTED EDITOR

Difficulties in learning to use an editor come from the number of concepts which the user must understand to be able to use the facilities of the editor. Although use of the ted editor has become widespread, only a few people understand all of the concepts embodied in this editor. Because many of the concepts are implemented by ted requests unfamiliar to most users, a simple typing error can cause the user to perform some unknown operation on his file. This makes the ted editor too dangerous for many users to use.

To meet the goals for tailored user interfaces (Goal 1), for minimizing the teaching/learning time for basic editing (Goal 2), and for ease and safety in editing (Goal 4), the editor must be designed in such a way that users can quickly learn a small amount of information about basic editing and immediately begin using the editor. Then, as the need arises, and their ability and confidence improves, users can add new/advanced editor concepts at their own pace of learning.

Let us enumerate some of the concepts involved in editing, based upon the goal of providing editing capabilities similar to those of qedx, ted and teco (Goal 3).

Editor Syntax
- mode of entering input lines
- mode of entering edit requests
- basic syntax of editor requests.
- basic editing operations which are supported.
- syntax of nonconforming (idiomatic) editor requests.
- scenario of user interactions (prompting, queries, command invocation, etc)

Line-Oriented Editing
- basic operations for manipulating text on a single line or for printing/reading/writing an entire file.
- multi-line text manipulation operations (line addressing).
- escaping Multics commands from editor request level

Context-Oriented Editing
- simple contextual searching (no special characters in search expressions).
- special characters in search expressions.
- multi-component addresses (eg, /abc/+3).

Macro-Operation Editing
- line movement operations within a single buffer.
- macromanipulation operations such as the qedx substitute, global and exclusive requests.
- partial line editing (character addressing).

Multiple-Buffer Editing
- multi-buffer line movement operations (buffer addressing).
- editing several files at once.
- buffer copying

Edit Programming
- buffers containing edit requests (edit subroutines, edit functions)
- executing edit requests in a buffer (invoking edit subroutines)
- file and buffer attribute expressions (buffer length, file pathname, file writability, etc)
- logical variables, expressions and operations
- conditional execution of edit requests
- arithmetic variables, expressions and operations
- looping through edit requests
- passing arguments to edit subroutines
- returning a value from an edit function
- invoking and using the result of an active function
- using canned edit programs to create a special editing environment (edit request files)
- programmed recovery from user errors
- error diagnostics

Special Editing Facilities
- tab insertion and deletion; elementary table formatting.
- upper- and lowercase shifting operation.
- underlining operation.
- speedtype expansion of input.
- use of abbreviations in edit requests.
- character translation.
- buffer sorting.
- buffer composing.
- caller-provided extensions and tailoring of editor subroutine
- user-provided extensions to the basic editor

Some of the concepts above are fundamental to all editors of a given type. For example, Editor Syntax and Line-Oriented Editing concepts specify the fundamental ways in which a line-oriented editor is used and the types of editing that can be performed. A cursor-controlled editor would have a different editor syntax and mode of operation than a line-oriented editor. These fundamental concepts must be understood by all users of the editor before any editing can be performed.

Other concepts need not be understood to perform basic editing functions. Macro-Editing, Multiple-Buffer Editing and Edit Programming are examples of concepts which can be added to the fundamental editor concepts as the user's confidence grows and her editing abilities increase. Because there are so many different concepts, these concepts should probably be grouped together in clusters of similar (or equally-useful, or equally comprehensible) concepts. The clustering shown in the partial list of concepts above is only one of several possibilities, and is probably not the most useful clustering.

During the period when (the fundamental or added) concepts are new to the user, the editor must carefully watch for mistakes in using the new concepts, giving thorough diagnostic messages when definite mistakes are found and querying the user when possible mistakes might cause drastic actions (such as deleting the entire buffer or substituting for every character in the line) to be sure the specified action is intended. As the user becomes accustomed to the new concepts, the number of queries and verboseness of error diagnostics should decrease, but additional error diagnostic information should be available on request.

The following policies are defined to summarize the features of the concept-oriented editor described above.

Policy 1. Documentation and teaching of editing should be organized around the clusters of editing concepts.

Policy 2. Documentation and teaching should begin with the fundamental editing concepts. By learning these fundamental concepts, the user should be able to perform basic editing functions.

Policy 3. Additional concept clusters should be documented separately from the fundamental concepts, and should be learned by the user only when he is ready to begin using them.

Policy 4. Additional concept clusters should be independent of one another, so that they can be learned and used in any order. Any dependencies which are absolutely necessary should be clearly identified in the documentation. The editor should diagnose attempts to use a cluster without its dependent cluster. Of course, these additional concept clusters will be dependent upon the fundamental editing concepts.

Policy 5. The user should have the ability to specify when a new cluster of concepts should be added to his editing interface, and what that cluster will be. While the concepts are new to the user, the editor should give verbose error diagnostics and warn the user of potential mistakes.

Policy 6. The user should have the ability to specify when she is comfortable with a concept cluster. At this point, brief error diagnostics should be given with additional information available as needed. The user should be warned of fewer potential mistakes.

EDITOR REQUESTS

The heart of any editor is its request language. The syntax of requests can make it easy or difficult to learn a new editor, and will strongly affect how easily and how fast a particular editing task can be accomplished.

The goals for minimizing the teaching/learning time for basic editing (Goal 2), and for ease and speediness of use (Goal 4) suggest several policies affecting the request language. Much of the justification for these policies stems from the results of my experiments in editor writing. Techniques for emphasizing Goal 4 (and to a lesser extent, Goal 2) played a major part in the experimentation. I hope that the results of this experimentation can be applied in the design of the new editor.

Simple Syntax...

The first result of the experimentation was that the basic syntax of all editor requests should be simple. Simpler syntax involves fewer concepts to be learned, thus shorting the learning time for basic editing.

Uniform Syntax...

Of equal importance for short learning times is the need for a uniform request syntax used by all editor requests. Fewer concepts must be learned when only a small number of request formats (preferably only 1) are employed in the syntax of requests.

Smaller Set of Requests...

The total number of requests also affects the time required to learn the entire editor request language. This will primarily affect the speed and ease of use of the advanced editor user, since the novice user will not need to learn the entire request language in order to begin editing (Policy 2).

The number of requests tends to increase as additional functionality is added to an editor. This has been especially true in ted, were most of the letters of the alphabet (both upper- and lowercase) and printing symbols have been used for request names. This plethora of requests is one factor which makes ted so difficult and dangerous to use.

One way to offset the request explosion is to note that many different requests do about the same thing, but in a slightly different way. In ted for example, the 'p' request prints lines while 'P' prints lines preceded by line numbers. 'P' can be eliminated by applying a "with line numbers" modifier to the 'p' request (eg, 1,5pl) as long as the syntax of requests requires that white space separates one request from another. This approach of request modifiers was proposed for the Kissel editor (MTB-339). However, because of the many different types of modification involved, the number of modifiers increases almost as fast as the number of requests in ted. What's more, all modifiers are not applicable to every request, so the user is stuck with the significant task of learning which modifiers go with each request.

One possible escape from the request/modifier explosion is to initially design a more powerful set of requests for the essential editing functions (so that fewer requests are required). The key to this approach is to make requests perform double duty. This approach was used to some extent in Kissel's editor, where a generalized addressing scheme allowed the 'a', 'c' and 'i' requests to append, change or insert lines in any buffer, not just the current buffer. Recent design experiments carried out with Jim Falksen indicate that this approach can be carried even further, to obtain a rather elegant functionality from a minimal number of requests. This approach should be strongly considered for the new editor.(1)

Another way to minimize modifiers may be to provide user-settable options which control the way requests work. For example, a "with line numbers" option which caused the 'p' request to always print with line numbers might eliminate the need for a "with line numbers" modifier.

---

(1) These design experiments are described in a forthcoming MTB written by Jim Falksen.

Experience with ted has shown that the number of requests increases dramatically as more specialized functions are added to the editor. Examples include: inserting and removing tabs; shifting from upper- to lowercase; dumping buffer contents in octal; testing, conditional branching and looping facility; etc. Many of these facilities have proved highly useful in ted, and should probably be considered for the new editor. However, instead of implementing them directly inside the editor with single-letter request names, it might be better to implement them as a library of external editor functions which have longer, more mnemonic names. This is probably feasible since these functions are not often used. The editor function library concept also allows the user to write his own editor functions, and thus provides a user-extensible editor request language.(1)

Requests such as testing, conditional branching and looping requests which must be implemented directly in the editor because they control the execution of edit programs should probably have longer, more familiar and mnemonic names. Since these functions are used in edit programs, they don't have to be typed often. We can afford more keystrokes in such cases for the sake of edit program clarity, and ease of learning/using these functions.

The following policies are defined to summarize the discussion of edit requests described above.

Policy 7. The general syntax of all edit requests should be simple.

Policy 8. A uniform syntax should be used for all edit requests. Idiomatic request formats should be avoided whenever possible.

Policy 9. The total number of essential editing requests should be minimized. A library of mnemonically-named editing functions may be provided to supplement these basic editing requests.

Policy 10. If request modifiers are used, they should be few in number and they should be defined in a way which makes it obvious which modifiers can be used with which requests. Editing options should be considered as an alternative to modifiers.

---

(1) This idea was originally proposed by Jim Falksen, Tom VanVleck and others, and has been included here for the sake of completeness.

Policy 11. The set of essential editing requests should be as powerful and as elegant (in the sense of APL operator elegance) as possible.

Policy 12. Nonessential editing requests which must be integrated into the editor (rather than being editing functions) should have longer, more mnemonic names. These requests should follow the uniform request syntax whenever possible.

SPEED AND SAFETY OF EDITING

The goal for an editor which is easy, fast and safe to use (Goal 4) suggests several more policies affecting the editor request language.

Minimizing Keystrokes...

To accomplish a particular editing sequence as quickly and easily as possible, the user should have to type as little as possible. However, care must be taken lest a policy for a minimum-keystroke editor produce an obscure, difficult to learn editor. Many people feel that qedx and ted have minimized keystrokes to the great detriment of understandability and ease of learning. Clearly, a carefully considered compromise is required in this area.

Upper/Lowercase Shifting Failures...

Whenever a user is trying to perform some sequence of operations quickly, the likelihood of typographical errors increases. The editor request syntax must provide the user some safety from the potential ravages of typos. One of the most common typos involves a failure to properly shift from upper- to lowercase at the appropriate time.

Several steps can be taken to protect the users from case shifting errors. The most obvious is to minimize the number of case shift operations required in performing editing functions. This policy stems from the idea that the user will make fewer errors if given fewer chances to do so.

Another step which can be taken is to reduce the importance of shift failures in the editor request syntax. Whenever possible, upper- and lowercase characters appearing on the same key should have the same meaning to the editor. This is easily accomplished for alphabetic letters, since the upper- and lowercase versions of these letters appear on the same key on all keyboards. This policy is more difficult to implement for numbers and special characters, since such characters are paired differently on different keyboards. However, giving upper- and lowercase alphabetics a common meaning in the editor request syntax has a beneficial affect (as shown is several editors including teco). This policy should be strongly considered for any editor.

Minimizing Use of Easily-Mistyped Request Sequences...

The most frequent typos involve mistyping a single letter or duplicating a letter (because of typewriter key switch bounce, etc). Ill affects from such typos can be avoided by limiting the number of single character request names. Single character requests which make some change to the file should require some special surrounding context (in addition to their single letter), or should query the user before making the change. We may want to consider using two letter request names for requests which are not surrounded by a context. The delete request in qedx is a good example. Perhaps this request should be 'dl' or 'delete' rather than just 'd'.

It is clearly important to separate requests from one another and from their operands. Keyboard switch bounce could cause 'dd', which qedx would interpreted as "delete 2 lines". qedx would interpret 'ww' as "write into file w". If white space were required between requests, 'dd' would be diagnosed as an invalid request. Similarly, if delimiters were used to surround pathnames (eg, w/path/), then 'ww' would be diagnosed as an error.

Minimizing Special Case Letters...

Another problem which makes an editor difficult to use involves misuse of special-cased letters becuase the user didn't know or had forgotten that these characters were special. For example, the '.' and '*' special characters in qedx regular expressions occur far too frequently in the text being edited. They are bad choices for letters which have a special meaning by default, and which must be escaped to gain their usual meaning.

Clearly, the number of such special characters used in the editor should be minimized. When special characters are required, they should be chosen from the set of letters which occur least frequently in the text to be edited. Perhaps the user should have the option to change these letters to be more appropriate to his particular editing. He should certainly have the option of turning off the special meaning of such letters (by turning off a concept/option/etc).

The following policies are defined to summarize the discussion of editing speed and safety given above.

Policy 13. To make editing fast, the number of keystrokes required to perform the most frequent editing sequences should be minimized. Care should be taken when applying this policy to prevent the editor request syntax from becoming obscure.

Policy 14. Extremely powerful requests in the editor language should have longer names which are difficult to mistype, or should be required to appear in an elaborate context which is unlikely to occur in a mistyped sequence of characters, or should query the user for permission to drastically modify the file.

Policy 15. The number of case shifting operations should be minimized in frequent editing operations.

Policy 16. Uppercase and lowercase letters should have the same meaning in the editor request language. In addition, any other sets of characters paired together on most keyboards should have the same meaning in the editor request language, especially if one of these characters appears before or after a case shift in editor requests.

Policy 17. Editor requests should be separated from one another by some delimiter. Operands of editor requests should be separated from the request name by some delimiter, especially operands which follow the request name.

Policy 18. A minimum number of letters should have a special meaning in editor requests or input strings (outside of the ordinary meaning letters typically have in that context). Letters chosen to have a special meaning should occur infrequently in the contexts in which they are typically used.

Policy 19. The should should be able to specify which letters have a special meaning in particular contexts, or should be able to turn off the special meaning of such characters (temporarily by escaping the meaning, or permanently by turning off an option or concept).

EDITOR SUBROUTINE INTERFACE

The goal for a subroutine interface to the editor (Goal 10) is intended to allow the new editor to be used from a subsystem. Examples of existing subsystems which could use an editor subroutine interface are: the mail command (to edit mail being sent and incoming letters); the merge_ascii command (to edit pieces of the multiple input files being merged); the various special-purpose editors used for system administration (eg, ed_mgt, ed_installation_parms, edit_proj, edit_reqfile, etc); the fast, dfast and linus subsystems (which employ editors to edit their special files).

In addition, an editor subroutine could be used for a variety of new applications: in the abbrev processor (allow a mistyped or invalid command line to be edited); in an I/O switch auditing I/O module (to edit previously-typed input lines and resubmit them as new input, or to edit output lines for use as subsequent input lines, etc); in the new bug file maintenance tools soon to be proposed (to allow editing of a single record in the bug file, or editing of a single field within a record, or scanning of a single field in all records for a contextual match, etc); in specialized word processing subsystems such as a forms editor, a simple paragraph-by-paragraph letter editor or a document management subsystem (to allow editing of a particular field in a form, paragraph in a letter or document part without exposing other fields/paragraphs/parts to accidental destruction).

I am sure other applications will come to mind; I have given the brief list above to identify a range of applications so that the implications of subsystem editing on the editor design can be explored.

The advantages of sharing a common editor interface in all Multics subsystems should be obvious. Users of many different subsystems will not have to learn a new editor for each subsystem. The subsystems will share editor code, so development and maintenance costs and introduction of bugs will be minimized. If a flexible, easy-to-use editor subroutine interface can be designed, the disadvantages of such an interface will be minimal.

Editing a Window...

One obvious implication of providing an editor for subsystems such as merge_ascii, SysAdmin editors and a bug file editor is that it must be possible for the editor subroutine interface to edit a piece of a segment (a window looking into part of the segment) without exposing the remainder of the segment to the user.

One implementation might copy the window into a separate editor buffer, invoke the editor subroutine with just that buffer, and then copy back the result.

A more efficient implementation might copy the entire segment into an editor buffer (assuming that many parts of the buffer would be edited using different windows throughout the editing session), and then define a pseudo-buffer which overlays the window in the original buffer (without copying the viewed data). For example, in merge_ascii the various input files could be treated as read-only editor buffers. The pieces of these input buffers to be merged at any given time could then be overlaid with windowing pseudo-buffers. The merge_ascii editor could then make only these windows available to the user whenever editing is required; the user could then select pieces from one/all of these windows plus type in original input lines, perform substitutions, etc to create the merged output window. The merged output window would then be appended to the output segment merge_ascii is building.

It should be noted that this scheme has the advantages of standard editor buffer referencing conventions and full editor request language which the current, specialized merge_ascii editor lacks.

---

(1) On the other hand, you may want to allow the reader to read (all or part of) another segment for use in the merged output.

Limiting Request Language...

Another obvious implication for subsystem editors is that not all requests defined for the standard segment editor will have meaning in a subsystem context. For example, read and write requests may *not* have meaning in a merge_ascii context, where the segments being merged are specified only in the command line.(1)

Some mechanism must be available for the subsystem to control which editor operations are allowed (and perhaps to what extent such operations are allowed). This could be specified by some editor mode setting or by a control structure in the editor subroutine interface.

Extending Request Language...

The complement of a subsystem limiting the editor request language is its extension to include specialized functions required by the subsystem. For example, the audit I/O module editor needs a function to resubmit a set of lines as input. A bug file editor needs the function of performing a keyword search through a particular field in some/all records to find a record containing some or all of the given keywords.

It should be possible for the subsystem to provide the editor with an interface which can be called when an unknown (or disallowed) request is encountered so that the subsystem can implement its own specialized functions. The subsystem would then have the option of implementing the request in its own way or of diagnosing the request as a real error.

The interface should have the ability to accept an address range, the name of the request found by the editor (using its standard rules for parsing the editor request language), and the remainder of the request line (to obtain other operands).

Making Editor Utility Functions Available...

It should be clear from the above discussion that a full set of editor utility routines must be available to the calling subsystem so that it can construct buffers and pseudo-buffers, implement specialized editing functions, and interface cleanly with the editor.

The set of available utilities should include: buffer definition and manipulation routines; addressing routines; buffer contents changing routines; error message printing routines; help routines; segment read/write routines; request operand parsing routines; input line reading routines; plus higher-level editor request functions such as a global substitution function, etc.

Policy 20. The editor should provide and support buffer windowing functions and window pseudo-buffers.

Policy 21. The editor should provide a way to limit the requests defined in the request language. This limiting mechanism should probably be available as an editor request for use in editor macros.

Policy 22. The editor should provide a mechanism which allows the subsystem to provide specialized requests which extend (or replace) the functions provided by the standard editor.

Policy 23. The editor's utility functions should be available for use by calling subsystems and by user-written editor functions.

EDITOR OPERATIONAL STRATEGIES

The remainder of this MTB describes several different buffer manipulation strategies which might be used to implement a new editor. While this topic may not be of interest to all readers, the work undertaken in this area of internal editor implementation deserves to be documented and considered, for each of the existing Multics editors has chosen a different buffer manipulation strategy. The paragraphs below try to describe these various strategies, highlighting their relative advantages and disadvantages, in the hope that some insight can be gained into the type of strategy most appropriate for a new editor.

Since editors are the most frequently used Multics commands and are the commands active for the longest time in the average process, it is important that the load generated by the new editor on the total system resources be as small as possible. The editor must be as efficient as possible (Goal 6) and must make the best use of Multics hardware and software features (Goal 7) if it is to provide optimal performance (fast response) using a minimum amount of system resources.

In analyzing editor performance, it is useful to identify the steps involved in processing a typical editing operation. The general format of an editing request in qedx or ted(1) is:

<address range> <operation> <operand>

For example, take the request to change a group of lines:

```
.,/abc/c The new
input which replaces the
changed lines.
\f
```

Such a request is processed by performing the following steps:

1. Determine the range of lines (or characters) to be operated upon by the request (ie, evaluate the address range of the request).

2. Branch to the code which processes the named operation. This code must validate the address range as one which is sensible to use with this request.

3. Obtain any operand(s) required to perform the request. This may involve reading input lines from the user, or reading a pair of substitution strings, or there may be no operands required.

4. Perform the requested operation on the given address range using the given operand(s).

Steps 1, 2 and 3 above are reasonably straight-forward. The steps involve fairly well-defined algorithms with little choice in their implementation beyond a selection of an efficient coding style and reasonable representations for the data involved. Of the three, step 1 offers the most chance for optimizing efficiency since determination of the address range may involve searches of large parts of the buffer (to determine absolute line numbers or to search for a string). Efficiencies in step 1 come from reducing the amount of the buffer which must be examined to reduce the working set of this code. For example, buffer scanning can be reduced when absolute line numbers are given in the address range by maintaining the line number of the current location as various operations are performed on the buffer, rather than recomputing it whenever an absolute line number is given.

---

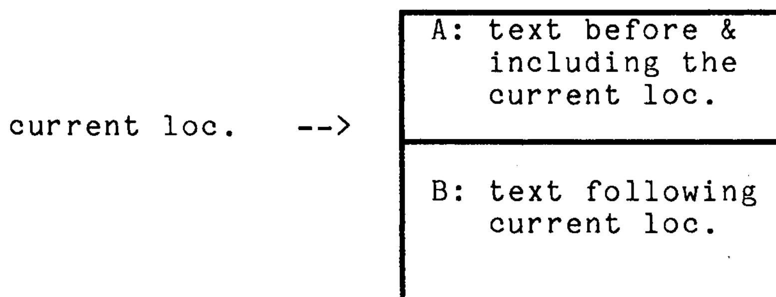
(1) Other editors such as teco or edm may use a different format for their requests, but the requests are processed by performing steps similar to those described for qedx and ted above.

Step 4 offers the greatest possibility for optimization because of the many possible algorithms which can be used to implement an operation on the buffer. Of the four major Multics editors (edm, qedx, ted and teco), each uses a different algorithm for operating on the buffer. The discussion which follows will describe each of these algorithms and highlight the advantages and disadvantages of each. My experimental editor uses a fifth algorithm which implements certain editor operations more efficiently. This algorithm will also be described and compared with the other algorithms.

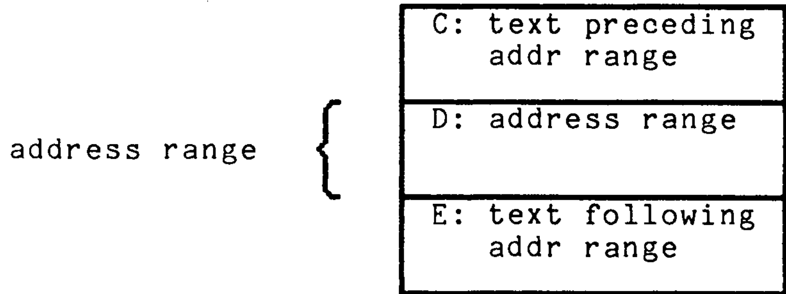
Introduction of Buffer Management Strategies...

We begin by describing the basic kinds of buffer operations which can be performed. We can then show, for each of the five algorithms, how this operation is achieved.

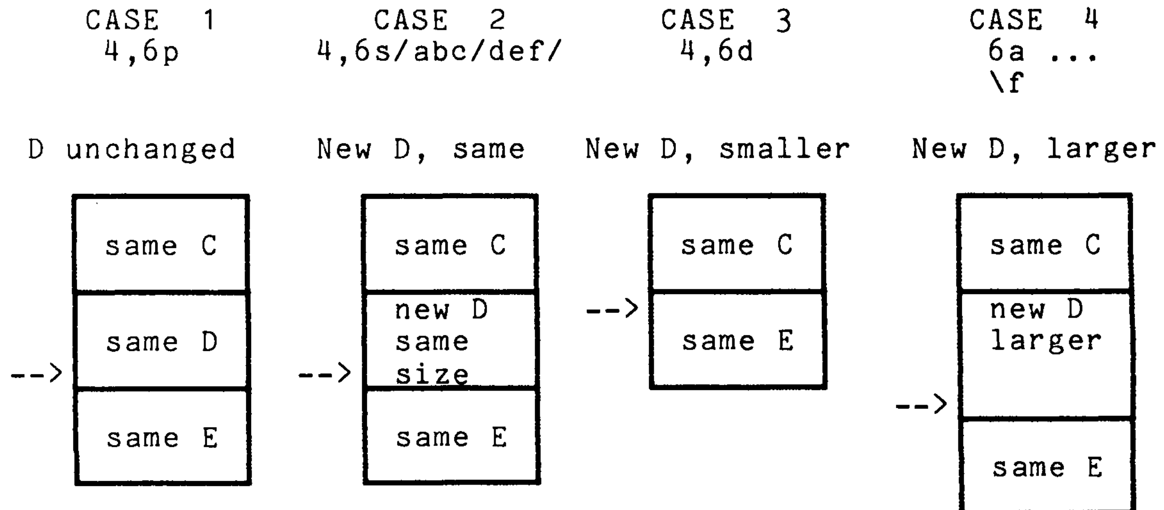
A given buffer is conceptually divided into two parts by the pointer to the current location in the buffer. This current location may be a particular line (the current line) in the buffer, or it may be a particular character (the current character) in the buffer. edm, qedx, and ted line-mode employ current line pointers, while teco and teco string-mode employ current character pointers. This situation may diagrammed schematically, as shown below.



The first step in processing a request is to identify the address range to be operated upon. This address range may or may not be related to the current pointer (eg, .,.+5 is related to the current pointer but 1,5 is not). Determination of the address range conceptually divides the buffer into three parts, as shown below.



All operations leave parts C and E untouched. A particular operation: (1) may leave part D untouched as well (eg, 4,6p); (2) may modify part D while leaving its size in characters unchanged (eg, 4,6s/abc/def/); (3) may modify part D and reduce its size in characters (eg, 4,6d); or (4) may modify part D and increase its size in characters (eg, 6a ...\f). These possibilities are illustrated below.



The main reason why there are so many different algorithms for changing buffers is the restriction in PL/I that, when a character string is assigned to another character string, the storage occupied by the two strings may not overlap. We will see that techniques for avoiding the overlapping string problem usually lead to maintaining two copies of the buffer, a before-the-change copy and an after-the-change copy.

qedx Buffer Management...

qedx employs the simplest buffer management algorithm. It maintains a before- and after-the-change copy of a buffer.

Each time the buffer is changed: part C (from the diagrams above) is copied from the before-copy to the after-copy; the new version of part D is appended to the after-copy; finally, part E is appended to the after-copy; then pointers to the before- and after-copies are interchanged. The same kind of copying occurs for CASES 2, 3 and 4 above.(1) No copying occurs for CASE 1, in which the buffer is not changed.

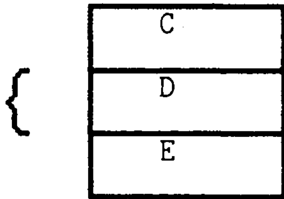
---

(1) Jim Falksen informs me that qedx and ted (when not in -safe mode) both perform the request 2,5d (CASE 3) totally inside the Before Copy without using the After Copy; however, 2,5s/abc/a/ is performed as shown above. Even though the string movement for 2,5d involves overlapping source and target strings (a violation of PL/I language rules), the code generated by PL/I works in this case of a smaller New D.

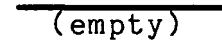
qedx - CASES 2, 3, 4

```
3,4c
...........
...New D...
...........
\f
```

Before Copy

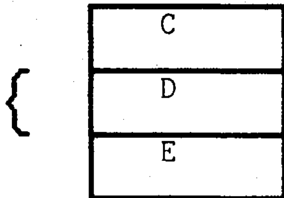


After Copy

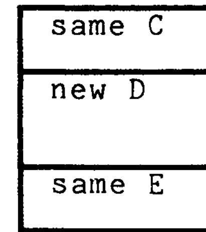


IS CHANGED TO

Before Copy

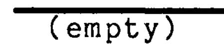


After Copy



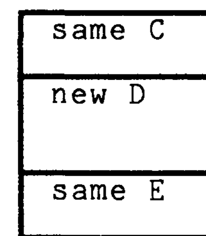
IS CHANGED TO

New After Copy



New Before Copy

-->



It should be obvious that this algorithm avoids the overlapping string problem by insuring that the source and target strings are in different segments. It also has the advantage that, in global requests (eg, 1,$s/abc/defg/), a single after copy can be built incrementally as changes are made on many different lines. Also, the buffer is never in an inconsistent state; the currently-used buffer reflects either before-the-change or after-the-change, never any state in between.

This algorithm has a very significant disadvantage. It has a basic working set which averages twice the size of the current buffer (size(before-copy) + size(after-copy)). We will see below that this working set size is larger than it needs to be, producing slowed editor response and a greater load on total system resources.

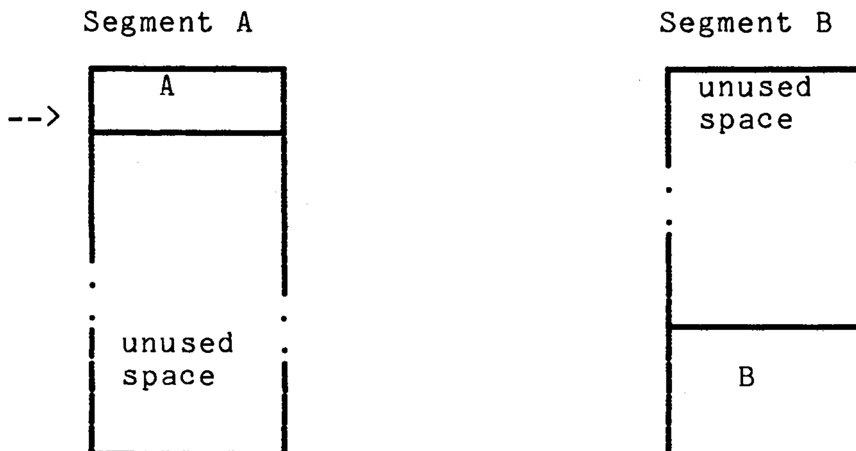
ted Buffer Management...

ted uses the qedx buffer management algorithm, but goes one step further by optionally placing the buffers in a user-specified directory, rather than in the process directory. Thus, in the event of a process or system failure, editing in progress can be restarted using the saved buffers. Taking advantage of the always consistent nature of the buffers, the user will lose, at most, the last operation in progress when the system/process failed, even if the buffer has never been written into the original file. This has proved to be a very safe method of editing which is much used and much in demand.

Of course, ted shares qedx's disadvantage of a large working set.

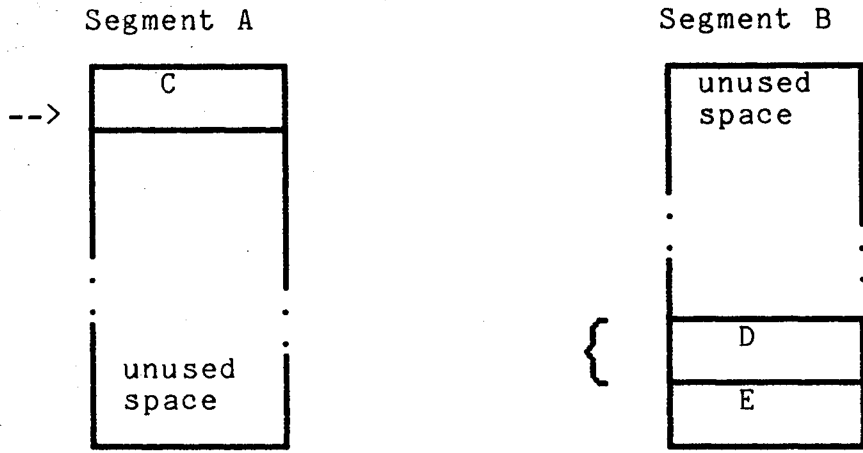
teco Buffer Management...

teco is a character-oriented editor. It maintains a pointer to a current character, irrespective of line boundaries. teco uses two segments to contain the current buffer: segment A contains part A of the current buffer, stored at the beginning of the segment; segment B contains part B of the current buffer, stored at the end of the segment. This buffer management system is illustrated below.

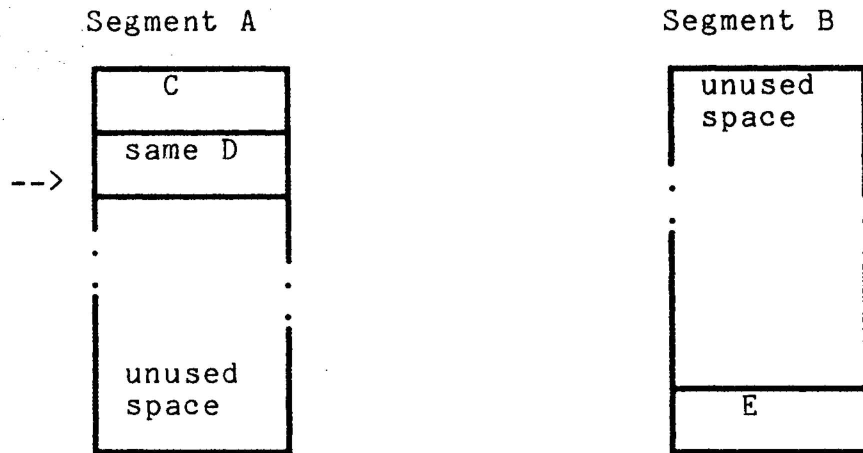


Movement of the current character pointer involves copying characters from the end of part A to the beginning of part B, or vice versa. This algorithm is illustrated below.

teco - CASE 1 2LT$



IS CHANGED TO

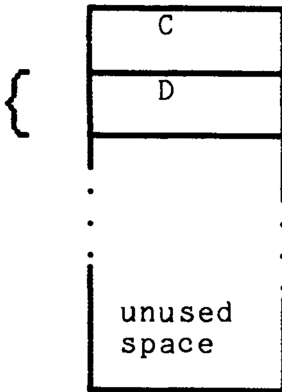


A change involves appending, changing or deleting the characters at the end of part A (since all additions to the buffer occur immediately after the current character), and/or deleting characters at the beginning of part B. This is illustrated below.

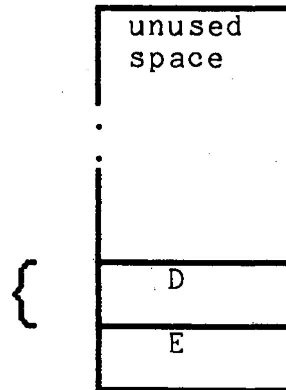
teco - CASES 2, 3, 4

```
-2K2Ki/...........
...New D...
...........
/$
```

Segment A

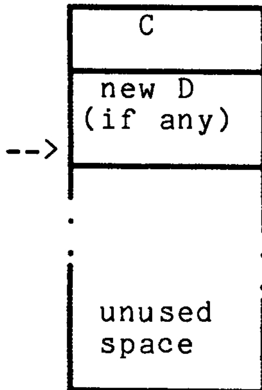


Segment B

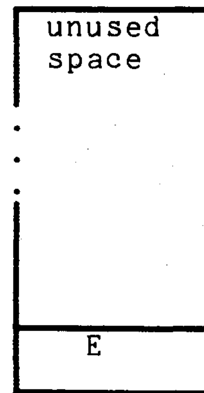


IS CHANGED TO

Segment A



Segment B

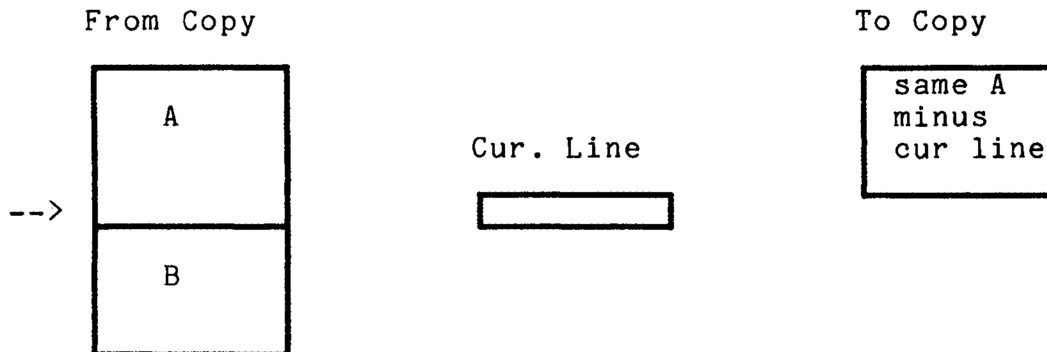


The teco algorithm has an advantage over the qedx/ted algorithm. While is does use two segments, it involves a smaller working set than qedx/ted (size(old D) + size(new D), for the worst case).

However, the teco algorithm has many disadvantages. It causes characters to be moved for the common case of moving the current pointer. It uses the end of Segment B, forcing the use of a 255K AST entry.(1) Operations which straddle the current pointer are more difficult to implement since they must be done in two pieces. The teco buffers are inconsistent during the period in which modifications are underway. These disadvantages are rather significant.

edm Buffer Management...

edm is a line-oriented editor, but it uses a two-buffer management scheme similar to qedx, plus a separate buffer to hold the current line. edm uses its buffer segments as follows: the From Copy holds part A (including the original version of the current line) and part B; the To Copy holds just part A. This scheme is illustrated below.

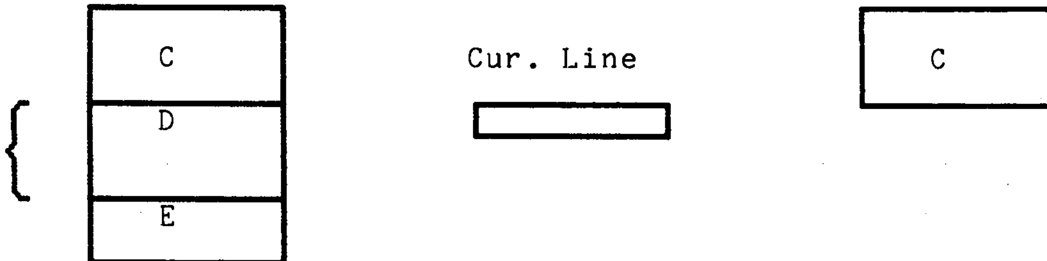


(1) teco could be optimized to use the smallest possible AST entry size for Segment B, and could use larger sizes only when necessary. However, this would involve the poor coding practice of building a knowledge of AST entry sizes into teco.

Moving the current line forward involves storing the current line buffer at the end of the To Copy, moving the current line pointer of the From Copy to the appropriate line in part B, copying lines passed over (excluding the new current line) from the From Copy to the To Copy, and copying the new current line into the current line buffer. These steps are illustrated below. Moving the current line backwards involves moving the current line pointer of the From Copy back to the appropriate line, removing the lines passed over from the To Copy (the new current line is removed, but the old current line was never in the To Copy so could not be removed), and copying the new current line into the current line buffer.

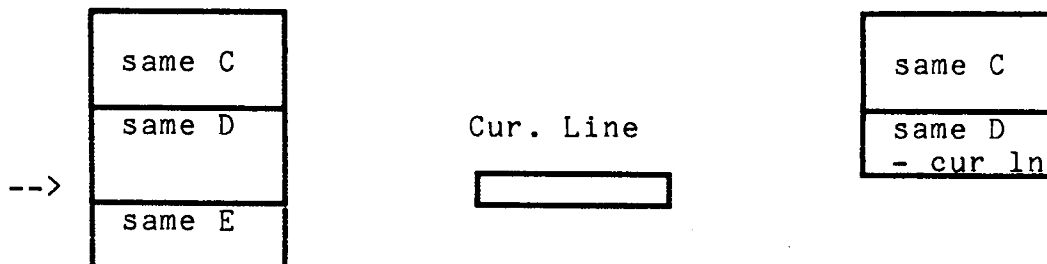
edm - CASE 1 n 2

From Copy To Copy



IS CHANGED TO

From Copy To Copy



Changing the current line (CASE 2) involves changing only the current line buffer. No change is made in either the From or To Copies. The following request is an example of CASE 2:

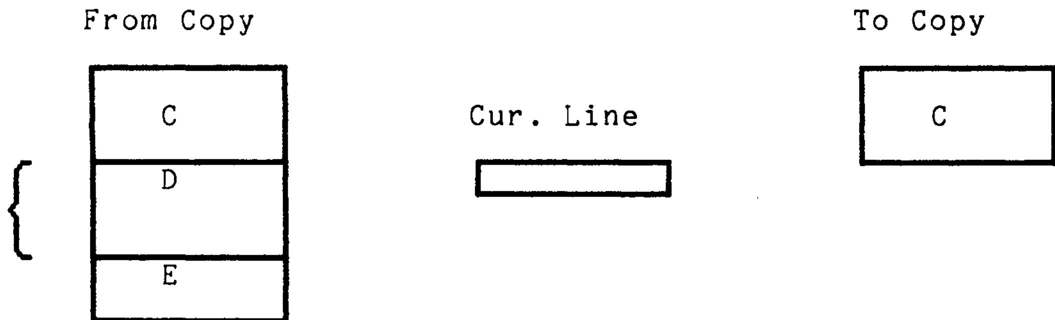
c /abc/defgh/

Changing several consecutive lines (CASES 2, 3, 4 special cased) involves moving to the first line to be changed (as described above for CASE 1), making changes in the current line buffer to this line, copying the current line buffer to the To Copy, moving forward to the next line to be changed, and repeating the process. An example of these special CASES 2, 3, and 4 is:

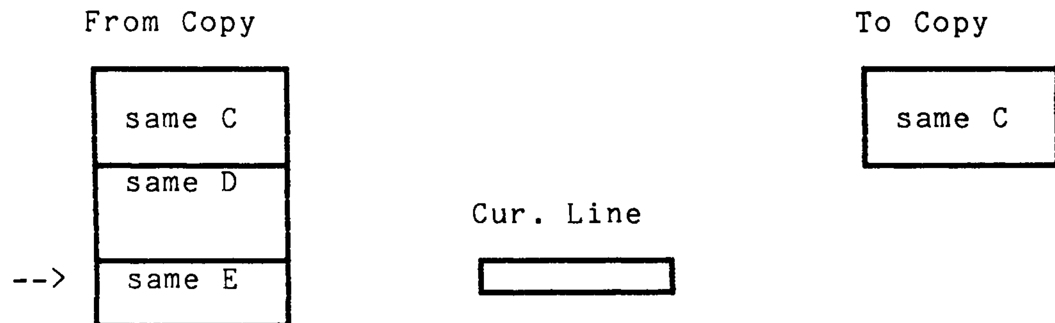
n 2
c 4 /abc/defgh/

Deleting lines below (including) the current line (CASE 3) involves moving the current line pointer in the From Copy without copying passed over lines to the To Copy, and then copying the new current line into the current line buffer. This is illustrated below.

edm - CASE 3 d 3



IS CHANGED TO



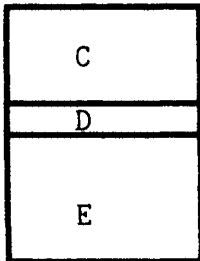
Similarly, appending lines below the current line (CASE 4) involves copying the current line buffer to the end of the To Copy, adding the new lines to the end of the To Copy, and copying the final new line into the current line buffer. This is illustrated below.

edm - CASE 4

.
Input.
.
....
...New lines...
....
.
Edit.

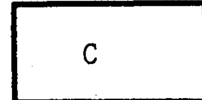
From Copy

{



Cur. Line

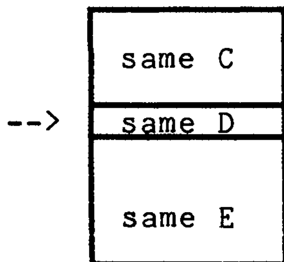
To Copy



IS CHANGED TO

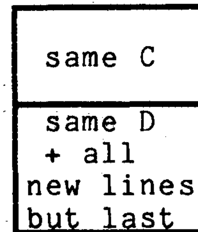
From Copy

-->



Cur. Line

To Copy



It should be obvious that edm shares many of the advantages and disadvantages of the teco buffer management strategy. edm has a small working set for most operations. Often, the working set is limited to the current line buffer. At worst, it includes the current line buffer, plus the size of the From address range (size(old D)) and the size of the new part of the To Copy (size(new D)).

edm shares teco's disadvantages of causing character movement when the current pointer is moved; of having buffers in an incomplete/inconsistent state during the period when modifications are underway; and of having operations complicated by data straddling two segments. Even worse than teco, edm has data straddling three segments. These are significant disadvantages.

Experimental Editor (ed) Buffer Management...

The experimental editor (ed) which I have developed is a combination line- and character-oriented editor similar to ted. However, it employs an entirely different buffer management strategy. Whereas all of the other editors avoid the overlapping string restriction of PL/I by using two segments, ed avoids it by not using PL/I to move its strings. Instead, it calls the appropriate entry point of an ALM procedure to perform an MRL or MLR hardware instruction, depending upon the direction in which the string is being moved. Given two strings, X and Y, of equal length as shown below:

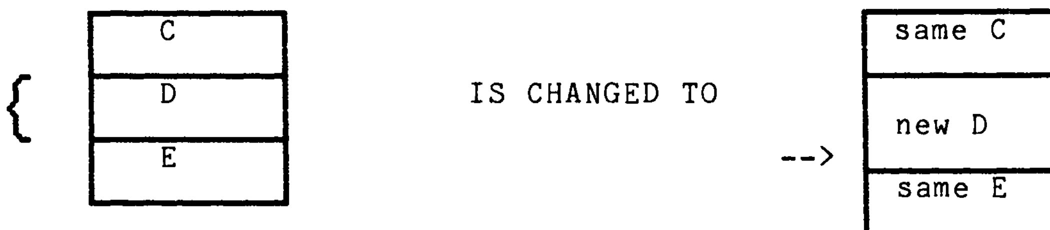
an MLR instruction can safely move the contents of string Y into string X. Similarly, an MRL can safely move the contents of string X info string Y.

The ALM procedure which performs this movement does not push a stack frame (it shares its callers stack frame) and does not require argument descriptors. Therefore, its basic cost of invocation is the cost of storing pointers to its arguments in an argument list. Thus, it can be invoked cheaply to perform the desired buffer movement operation.

As with qedx and ted, no copying is required to move the current pointer (CASE 1) using ed's buffer strategy.

Each time the buffer is modified, part E is moved up or down to accommodate the new size of part D; then the new D (if any) is inserted into the space which was made. This strategy is illustrated below.

```
ed - CASES 2, 3, 4          3,4c
                            ...........
                            ...New D...
                            ...........
                            \f
```



As mentioned above, this buffering strategy avoids the overlapping string problem by using an ALM MLR/MRL procedure to move overlapping strings. It has the advantage of a smaller working set than qedx or ted (size(E) + size(new D)). Also, since the entire file is contained in a single buffer, coding of editor operations is simpler than that of edm or teco.

The main disadvantage of ed is that its buffer is not always in a consistent state, as are the buffers of qedx and ted. During the period of a modification, part E may be damaged if a system/process failure occurs while it is being moved. Failure after part E has been moved but before the new Part D has been inserted may result in incorrect data for part D. Though these periods of inconsistency are short, they do exist and prevent this strategy from being totally safe across system/process failures. Also, ed's algorithm sometimes has a larger working set than that of edm or teco, especially when making modifications at the beginning of a very large file (when part E is very large).

Conclusion of Buffer Management Strategies...

It is difficult to judge which of the buffer management strategies described above is best for the new editor. In fact, it is doubtful that an optimum strategy for all types of editing even exists. For example, editing that involves selectively printing lines without modifying the segment can be implemented and performed most efficiently using the experimental editor's (ed) buffer management technique. On the other hand, editing which involves many changes and additions to the segment can be performed most efficiently using the split buffer technique of teco (as modified to minimize the AST entry size of the buffers) or of edm. If many changes are to be made to the current line (perhaps by a novice users who makes a changes and prints the line, then makes another change, etc), then the current line buffer technique of edm is most efficient.

It is tempting to avoid the coding complexities of the split buffer technique in an already complicated editor design. However, use of split buffers could significantly reduce the working set of the editor. This would have a significant affect upon the total system load generated by editing. Since editing is a significant part of the workload at most sites, the proper buffer management strategy could improve overall system performance and response.

I am not going to choose a particular buffer management policy in this MTB. Instead, I ask for your comments in this area. All of the buffering techniques can be coded in such a way that editing can be restarted after a process/system failure with consistent buffers and minimal data loss.(1) Therefore, the basic question is one of performance, of types of editing to be optimized, and of coding complexity. Your comments would be appreciated.

---

(1) The experimental editor (ed) buffer strategy cannot guarantee consistent buffers if the process/system failure occurred while the buffer was being modified. However, since the periods of modification are short, the likelihood of failure during such periods is small. Note that, in input mode, the buffer is not modified until the end of input mode is encountered. The input lines are then added to the buffer as a whole.