THE TXDT PACKAGE
INTERLISP TEXT EDITING PRIMITIVES

BY J Strother Moore

JANUARY 1981

CSL-81-2

ABSTRACT

The TXDT package is a collection of INTERLISP programs designed for those who wish to build text editors in INTERLISP. TXDT provides a new INTERLISP data type, called a buffer, and programs for efficiently inserting, deleting, searching and manipulating text in buffers. Modifications may be made undoable. A unique feature of TXDT is that an address may be "stuck" to a character occurrence so as to follow that character wherever it is subsequently moved. TXDT also has provisions for fonts.

KEY WORDS AND PHRASES

string searching, word processing, fonts, text representation, undoing, INTERLISP, text editing

CR CATEGORIES

3.70, 3.73, 3.74

XEROX
PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

1. Historical Preface

In 1971, at the University of Edinburgh, Robert S. Boyer and I developed a method of representing text that was economical in use of storage, efficient for many common operations, and permitted the implementation of "undoable" modification. Using that representation, we implemented an editor, called the 77-editor (named after the disk sector on which it resided), in the programming language POP-2. The editor is described in [Ref 1].

In 1973, I moved to Xerox Palo Alto Research Center where I implemented the editor in INTERLISP-10. That early implementation became the basis of an experimental text editor patterned after POET implemented by Warren Teitelman. Warren gradually evolved the idea of a friendly display-based editor into a full-fledged interface to Interlisp [Ref 2] and I evolved my editor primitives to suit his needs. The entire system of editing primitives was implemented several times from the ground up and in 1975 I documented (in an unpublished PARC report) an early version of the "TXDT package".

In 1976, I left PARC and joined SRI but Warren and I continued informally in our established roles for several years. Many features were added during that time, including multiple buffers and fonts. Eventually the package stabilized to what is described here.

In 1979, while working as an SRI consultant to Xerox PARC, I instructed Bill Laaser of PARC in the implementation of TXDT; Bill has repaired the occasional bugs discovered since and has implemented TXDT on the Dolphin/Dorado series of machines. Richard Burton, of Xerox PARC, carefully read a manuscript of this manual and suggested many improvements.

2. Introduction

The TXDT package is a collection of INTERLISP programs designed to support those who wish to build text editors in INTERLISP. Strictly speaking, all TXDT provides is a new INTERLISP datatype called a "buffer" which, like a file or string, contains a sequence of ascii characters. Unlike INTERLISP files and strings, buffers are efficiently expanded and contracted as characters are inserted and deleted.

Arbitrary locations in a buffer may be addressed and may be the target for insertions, deletions, etc. A unique feature of TXDT is that addresses can be "stuck" to characters so that they follow the character wherever it is moved. All modifications made to buffers are undoable. Finally, where possible, all operations are symmetric with regard to direction of scan through a buffer. That is, one can move, search, substitute, etc., backwards as well as forwards.

It should be understood from the outset that there is much to a good text editor beyond what is provided here. For example, buffers have no structure TXDT sees them only as character sequences, not sentences, paragraphs, words or S-expressions. TXDT knows nothing about i/o devices or user interaction. It maintains no screens and monitors no mouse buttons or keys. The designer/implementor of a TXDT-based text editor is responsible for implementing these higher level features using INTERLISP and the TXDT primitives.

This document is organized as follows. The first few sections informally describe how TXDT looks to the user and what its most important features and limitations are. Then follow a series of more precisely written sections which specify the functions in the TXDT package. Functions are grouped according to the kind of operation they perform. A table of contents and index are provided. After the main TXDT functions are specified, the underlying implementation is described so the user can have a better idea of where the inefficiencies lie and why certain things can or cannot be done. Also specified are several functions which expose a certain amount of the implementation but which are sometimes useful. The document concludes with sections listing the TXDT error messages, variables of interest to the user, and datatypes used by TXDT. It is sometimes necessary to distinguish the PDP-10 implementation of TXDT from the Dolphin/Dorado implementation. We therefore sometimes talk about INTERLISP-10 or INTERLISP-D as opposed to INTERLISP, and TXDT-10 as opposed to TXDT.

3. Buffers and Boxed-addresses

An arbitrary number of buffers may be maintained at once. It is possible to modify or otherwise operate upon any buffer at any time. To operate upon a buffer, one must have an "address" into it, indicating where the operation is to take place. The most efficient and primitive form of address in TXDT is the "boxed address", which is an instance of an INTERLISP user datatype named TXDTADDR.

Unlike many editors (e.g. TECO), where an "address" into a buffer is just a character count from the top to the given point, TXDT boxed addresses are actually attached to the text. In fact, they are attached to the single character to which they point. It is convenient to construct the following model of a buffer in order to provide an intuitive explanation for what a boxed address is and how it behaves. A buffer is analogous to a row of toy blocks, each block representing some letter or character. To insert some new text at some location, the new text is assembled as a row of blocks, the old text is parted at the appropriate place, and the new is inserted. To delete some segment of text, the buffer is parted at each end of the segment, the segment is slipped out, and the gap is closed by pushing the two halves of the buffer together.

A boxed address is just an identification of a particular block. That is, if we search the buffer and find some occurrence of the word "FOO", and obtain the boxed address, ADDR, of the first character, "F", we can imagine ourselves having tied a string to the particular block in the buffer representing that "F". Now let us consider what happens under various operations on the buffer.

If we insert text anywhere in the buffer we will find that ADDR still references that very same "F", in the sense that the string is still tied to the same block. It does not matter that the block may have changed its relative position from some point, or that it may no longer be followed by "OO".

If we delete a segment of the buffer, ADDR will still point to the "F", if it is still in the buffer. If the "F" has been deleted, an attempt to use ADDR will generate the error TXDT ADDRESS INTO DELETED AREA.[1] The way this is done is as follows: when a segment of the buffer is

1. The TXDT error messages are reassuringly verbose. A complete list of the TXDT error messages is included in the section "Error Messages".

deleted, the characters in that segment are marked as such. Any attempt to use a boxed address which references a marked character causes the error.[2]

Finally, TXDT supplies the necessary functions for grabbing a segment of the buffer (deleting it from the buffer but returning a pointer to it) and later inserting that segment (that is, the very same row of blocks) elsewhere. In this case, a boxed address referencing the text deleted is considered invalid until the text has been reinserted; once the segment has been reinserted, all addresses into it are valid again, and reference the very same characters as before. For example, if the text around the "F" to which ADDR points is grabbed and later inserted elsewhere, ADDR would reference that same "F" in its new location.

The section "Implementation Functions" describes the actual representation of buffers and boxed addresses.

4. Specifying Addresses

Given a boxed address into a buffer, the buffer may be the subject of any TXDT operation. However, TXDT permits one buffer to be distinguished as the "current" buffer. A buffer is designated as the current one using the functions TXDTCURBUFF. The current buffer may be addressed with boxed addresses (as usual) but it may also be addressed with any of the following forms of "unboxed addresses":

TOP denotes the address of the first character in the current buffer.

BTM denotes the address following the last character in the current buffer.

n, where n is an integer denotes the beginning of the nth line in the current buffer, counted from the top. Nonpositive values of n are treated as TOP. Excessively large values of n are treated as BTM.

(n . c), where n and c are integers denotes the character addressed by moving to the nth line (in the sense described above) and then moving forward or backward c 1 characters, depending on the sign of c. Thus (20 . 15) is the 15th character after the beginning of the 20th line which may or may not be the 15th character of the 20th line, depending on whether the 20th line has at least 15 characters in it.

(NIL . c), where c is an integer denotes the address of the cth character in the current buffer, counting from the top. Nonpositive values of c address TOP and excessively large values of c address BTM.

All TXDT functions which take addresses as arguments accept either boxed addresses or the objects described above. When an unboxed address is used, the address implicitly refers to the current buffer. Because characters have to be fetched and inspected before a line or character address can be used, boxed addresses which point directly into the buffer are considerably more efficient than line or character addresses. The user should therefore pass boxed addresses whenever possible. All TXDT functions which return addresses actually return boxed addresses.

---

2. The delete routine does not have to visit every character deleted to achieve the effect of marking them all. Also, the marking is undone if the delete is undone, thereby rendering the address valid again.

The function txdtbox takes an address of the type described above and produces a boxed address that points to the location indicated. The function txdtunbox maps boxed addresses into line or character specifications.

Note that BTM is a very special address. Unlike all other addresses it does not reference a character; it points to immediately behind the last character in the buffer. BTM is supplied because the insertion functions insert text immediately before the character referenced by the "target address." The only way to insert text after the last character in the buffer is to use BTM as the target address.

When the buffer is empty, TOP and BTM reference the same location. Either may therefore be used as the target address for the first insertion into the current buffer.

5. Buffers

A TXDT buffer is an instance of the datatype TXDTBUFFER. Buffers are constructed by the function txdtcurbuf. There may exist any number of buffers. However, one buffer is always designated as the "current" one. The current buffer is the one used when addresses are specified as line and character numbers ("unboxed" addresses). The current buffer has no other distinguished role. Any buffer may be modified at any time.

In addition to the text in the buffer, each buffer contains three fields used to help decode line and character addresses when that buffer is the current buffer. The names of these fields are TXDT$, TXDTPOETDOTADDR, and TXDTPOETDOT. TXDT$ must contain the number of line terminations in the buffer. TXDTPOETDOTADDR must be a boxed address in the buffer. It may be thought of as the current "cursor position" in many editors. TXDTPOETDOT must be a pair, (l . c), giving the unboxed address equivalent to TXDTPOETDOTADDR. When the buffer is the current one, the contents of these three fields are stored in the global variables txdt$, txdtpoetdotaddr, and txdtpoetdot. If the global flag txdtpoetflg is nonNIL, these three globals are used to make the decoding of line and character addresses more efficient. (See txdtbox.) txdtpoetflg is initially NIL.

It is up to the implementor of the larger editing system to maintain these global variables and the contents of the three fields TXDT$, TXDTPOETDOTADDR, and TXDTPOETDOT in each buffer. The fields may be updated with the replace operator of the record package. The values of the variables are important only when unboxed addresses are used while txdtpoetflg is on. The values of the fields in a buffer are important only when the buffer is made current. At that time, the contents of the three fields are stored in the three global variables for use by the TXDT unboxed address decoding function, txdtbox.

Note that TXDT itself never sets the contents of the fields in a buffer (except as noted in txdtcurbuf). When the user of TXDT makes a new buffer the current one, the user must update the contents of the old buffer's fields, if desired. When a TXDT function is used to modify a buffer that is not current, the user must update the buffer's fields, if desired. TXDT does not assume responsibility for these fields because they may not always be necessary. For example, the implementor may know that a certain buffer will never be addressed except with boxed addresses or when txdtpoetflg is NIL. When a noncurrent buffer is modified, the implementor may choose to

delay the computation of the correct field contents until just before that buffer is made current. To aid the implementor in maintaining these fields and variables, all TXDT functions which modify buffers can be asked to keep track of how many lines and characters are added or deleted.

txdtcurbuf[buf;undoably;defaultflg;msg]

If buf is NIL, a new empty buffer is constructed. The message msg is attached to an "invisible" address just beyond the top of the buffer (see the section "Messages and Fonts"). This has the effect of propagating msg as the message governing any otherwise ungoverned text inserted at the top of the buffer. The TXDT$ field of the new buffer is set to 0; the TXDTPOETDOTADDR is set to the bottom; the TXDTPOETDOT field is set to line 1, character 1. Then the new buffer is made current by setting the global variable txdtcurbuf to the buffer and appropriately setting the txdt$, txdtpoetdotaddr and txdtpoetdot variables. These settings are made undoable (via INTERLISP's UNDO command) iff undoably is nonNIL. The new buffer is returned.

If buf is not NIL, it is assumed to be a buffer previously constructed by txdtcurbuf. If defaultflg is nonNIL, the TXDT$, TXDTPOETDOTADDR, and TXDTPOETDOT fields of buf are set to default values computed by TXDTCURBUF. TXDT$ is set to the number of line terminations in buf; TXDTPOETDOTADDR and TXDTPOETDOT are set to the bottom of the buffer. Note that these fields of buf are smashed and the old values are lost. Then buf is made current as described above, undoably if so specified. Buf is returned as the value.

txdtkillbuf[buf;undoably;currentflg]

If buf is a buffer, it is killed in the sense that all the text in it is deleted and all the structures referenced by the representation of the text are freed for garbage collection. All fields of buf itself are smashed to the atom KILLED. The operation is undoable iff undoably is nonNIL.

If buf is NIL or a list, every element of buf is killed. If buf is the atom ALL, all buffers are killed.

If an attempt is made to kill the current buffer an error is caused unless currentflg is on (or buf is ALL). If currentflg is on, or buf is ALL, the current buffer will be killed with unspecified consequences if TXDT is directed to use the current buffer before a new current buffer is designated.

To detect whether a buffer has been killed, one may ask if (fetch TXDT$ of buf) is the atom KILLED.

txdtbufp[buf]

Returns buf if buf is a buffer and NIL otherwise.

txdtemptyp[buf]

Returns T or NIL according to whether buf is empty or has been killed. If buf is NIL, it defaults to the current buffer.

txdtwhereis[addr]

If addr is a valid address, txdtwhereis returns the buffer containing addr. Otherwise, txdtwhereis causes an error.

6. Windows into the Buffer

Some of the TXDT functions operate on the segment of text between two addresses. Such segments of the buffer are called "windows". For example, txdtdelete takes two addresses and deletes the text in the window defined by then.

Usually the first address defaults to TOP (of the current buffer) when it is NIL, and the second to BTM. Departures from this will be noted in the descriptions of the functions concerned.

The text in a window is defined to be that segment of the buffer starting with the character referenced by the first address and ending at (but not including) the one referenced by the second address.

Windows are therefore somewhat like open intervals. The window defined by two addresses includes the character referenced by the first address, but not that referenced by the second. This is actually quite natural, as use of the TXDT package will confirm. (Addresses could be thought of as pointing to the interstices between characters but then it is a little harder to model how boxed addresses are stuck to characters.)

For example, if one obtains the address of line 30, and the address of line 31, and gives these two addresses to txdtdelete (or equivalently, if one just executes txdtdelete[30;31]), then all of line 30 is deleted, and line 31 remains unchanged.

If the two addresses are to define a window, the second must be greater than or equal to the first, in the sense that the second is encountered in a forward scan of the buffer starting at the first. Otherwise the window is ill-defined. Note that both addresses must be into the same buffer if the window is to be well-defined.

Most of the TXDT functions which operate over windows do not verify that the window is well-defined. Instead they start their operation at one of the addresses and continue until hitting the other or the top or bottom of the buffer (depending on the direction of scan).[3] The only functions which verify that the window is well-defined are those that modify text in the window. An error will occur if these functions are given addresses which do not well-define a window.

3. This was an implementation decision made for efficiency and relying on the assumption that the TXDT functions will be embedded in a fairly sophisticated system designed by the user. In this case, the user will often know that a window is well-defined because of the process used to fetch the two addresses. For example, the first will be obtained, and then the second will be obtained with some operation that starts at the first and moves forward through the buffer. Two addresses obtained in this way must well-define a window.

Occasionally the user will have two addresses and not know which follows which. Two predicates, txdtequal and txdtgreaterp, are provided for ordering addresses in the sense defined above.

Finally, those functions which operate on windows decode the two addresses independently. Thus, txdtdelete[30;31] makes two calls to txdtbox, one to count from the top to line 30, and one to count from the top to line 31. If some relationship between the two addresses can be exploited to save time, the user is responsible for exploiting it. For example, in the case just cited, it would be more efficient to first obtain the boxed address of line 30 and then move one line down from there to obtain the boxed address of line 31.

The relationship between two addresses given as line numbers is not automatically exploited by the windowing functions because in general both addresses will not be line numbers.

7. Objects That May be Inserted

Character sequences to be inserted into a buffer (either via the insertion or substitution functions) may be obtained from a variety of sources. The various INTERLISP objects which may be inserted and their interpretation are as follows:

grabbed object. According to our toy block analogy for buffers, a grabbed object is a row of blocks previously removed from some buffer. Technically, grabbed objects are records of type TXDTGRABBEDOBJ and are constructed by the function txdtgrab. A grabbed object represents the text in the window that was grabbed. See txdtgrab for more details.

file segment. A file segment is a listp object whose car is eq to the value of the variable txdtinsertfilekey. Such a list denotes a window into the file named by the second element of the list. The window starts at the byte position indicated by the third element and extends to that indicated by the fourth element. For example, to insert the portion of the file <MOORE>FOO.BAR from byte positions 1000 to byte position 7000, the appropriate file segment could be constructed by:

```
list[txdtinsertfilekey;
  <MOORE>FOO.BAR;1000;7000]
```

If the third element is NIL (or nonexistent) it defaults to the current file pointer for the named file. If the fourth element is NIL (or nonexistent) it defaults to the end of file pointer for the file. Note that list[txdtinsertfilekey; file;0] causes the entire file to be inserted. An ill-formed segment is equivalent to an empty segment.

listp other than a file segment. A listp other than one whose car is txdtinsertfilekey denotes the sequence obtained by calling prin3 on each element in the list. (Note that this is not the same as calling prin3 on the list itself the outer parentheses and separating spaces are omitted.)

other. Any other object denotes the sequence obtained by calling prin3 on the object.

When the sequence denoted by an object must be obtained with prin3, the printing is done to a scratch file maintained by TXDT. Then TXDT behaves just as though a file segment were to be inserted.

8. Caveats About Files and SYSOUTS

Pieces of many different files may be inserted at any one time.[4] Any given file may be inserted as many times as desired. TXDT contains some fairly sophisticated software to handle the insertion of files. When a file is read into the buffer, the characters are not actually copied into core. In fact, the characters are not inspected at all. Instead, each page of the file is referenced through an object not unlike an INTERLISP string pointer, which gives the file name and page number concerned. Whenever the characters of a particular page must be inspected, that page is PMAPped (by a TXDT paging mechanism) into one of several scratch pages maintained by TXDT for this purpose.[5]

The effect of this arrangement is that during operations that scan large amounts of a buffer, such as searches from top to bottom, the pages of inserted files will be PMAPped in and out. But when a particular area has been singled out, it and neighboring pages are actually in core (for all practical purposes).

Changes made in a file being edited are not actually visited upon the file itself. The changes are actually represented by the state of the buffer. Hence, changes can be easily undone by restoring the state of the buffer. However, it is dangerous to undo operations out of order. (See the section "Implementation Functions".) Should the system crash during a session in which TXDT functions were being used to edit a file, all of the changes made since the last write will be lost. Of course, a buffer may be written to a file at any time, and the file written will contain all of the modifications made.

Because files which have been inserted may be PMAPped in, the normal INTERLISP file closing primitives will not operate properly. Furthermore, since inserted files are referenced by their JFNs, closing and reopening a file (which will most likely assign it a new JFN) typically has disastrous consequences on all buffers using the old JFN. Finally, because of these JFN and PMAP issues, the INTERLISP function sysout does not produce a useable core image. TXDT has facilities for dealing with these issues. The reader should see the section "File handling".

9. Line Terminations

The INTERLISP-10 version of TXDT must contend with a war between TENEX and INTERLISP-10 over the line termination protocol.

TENEX uses the carriage return character followed by line feed (CR/LF) to terminate lines in files. INTERLISP-10 uses the end of line (EOL) character in strings and CR/LF in files. When

4. INTERLISP-10 files must remain open while they are being edited. Since INTERLISP-10 restricts the number of open files to 16, at most 16 different files may be manipulated by the INTERLISP-10 version of TXDT.
5. A page is not PMAPped into one of these areas if it already occupies one; steps are taken to prevent a page which has recently experienced heavy use from being swapped out.

INTERLISP-10 prints an EOL to a file (as with PRIN3), it deposits a CR/LF pair. Unless the user has inserted single CR's or LF's, all lines in TXDT buffers and in files created by TXDT-10 terminate with CR/LF.

When counting lines, TXDT-10 treats LF as the line terminator.

When searching for a string containing an EOL, TXDT-10 accepts CR/LF as a match.

TXDT-D, on the other hand, naturally uses EOL uniformly to define line terminations.

10. Messages and Fonts

To facilitate the handling of special information like fonts, TXDT permits a "message" to be stored at an address. A message is either NIL, a single character atom (other than the one whose ascii code is 0), or a list of up to 126 such atoms.

Conceptually the messages in a buffer divide the text into regions; each character or address in a given region is considered "governed" by the message stored at the address at the front of that region. Typically, messages can be used to specify the font of the region governed.

Unless otherwise specified, text inserted into a buffer inherits the message governing the point of insertion. Each buffer has a message stored "above the top" that governs otherwise ungoverned insertions at the top. (See txdtcurbuf.) However, text to be inserted may contain special characters, called "message sequences", which specify the messages and message regions of the inserted text.

A message sequence is a sequence of characters beginning with the character contained in the variable txdtescapechar (hereafter called the "escape character"). Suppose the escape character occurs in text to be inserted. Here is how the message is parsed.

If the character following the escape character has ascii code 0, then the character code, n, following the 0 is taken as the message length and the message is a list containing the next n characters. If the character following the escape characters has ascii code 127, it is a signal that the escape character itself is to be inserted (and the 127 is deleted). If the character following the escape character has an ascii code other than 0 or 127, then that character is the message.

For example, suppose that @ is the escape character and that the following symbols have the associated ascii codes[6]

symbol	ascii code
#	0
$	127
2	2

6. We have here associated the codes 0, 127, and 2 to ascii characters other than their true ones because their true ones usually require more than one space to write down.

Then inserting the following sequence

@ATHE @#2AIESCAPE CHARACTER@A IS @#.

has the effect of inserting the text

THE ESCAPE CHARACTER IS @.

and attaching the message A at the T in THE, the message (A I) at the first E in ESCAPE and the message A at the space before IS.

This interpretation of inserted characters is disabled if txdtescapechar is set to NIL.

To store a message at an address in a buffer the function txdtputmsg may be used.

When text from a buffer is printed to a file (usually the terminal) with txdtprint, the messages in it are passed to the function txdtprintuserfn as they are encountered. Initially txdtprintuserfn is a noop but it may be defined by the user (e.g., to cause subsequent text to be printed in a different font). See txdtwrite and txdtprint for details.

11. Interrupts

During certain critical sections of TXDT code, unexpected interrupts could result in mangled data structures and cause irreparable damage to the current TXDT start up. Thus, certain critical TXDT operations either temporarily disable interrupts or take steps to recover from them after the fact. The user should arrange for the function txdtresetformfn to be called after any error or CTRL-D interrupt inside a TXDT function and before any other TXDT function is called. It is safe to call it anytime. One suggestion is to keep a call of txdresetformfn[] on the INTERLISP reset list. The function returns NIL.

12. Getting Started

Before any other TXDT function can be called txdtinit must be called.

txdtinit[]

> This function initializes the TXDT package. It declares the TXDT user datatypes if necessary; opens the scratch file as a temporary io file and stores the name in the global variable txdtscratchfile; initializes the swapping buffers; closes all previously inserted files; kills all existing buffers; sets up one new empty buffer; makes it the current buffer; and stores that buffer in the global variable txdtcurbuf.

If TXDT has already been started, txdtinit has the effect of releasing all the space associated with the old start up and starting over.

13. File Handling

txdtread[file;addr;behind;countlc;oldbox]

Reads the file file into a buffer. Equivalent to
txdtinsert[object; addr; behind; countlc; oldbox]
where object is list[txdtinsertfilekey; file].

txdtclosef[file]

Closes the file named file (in TXDT-10, if file is an integer it is treated as a JFN). Returns the full file name of the file closed. If the file was not open, NIL is returned.

txdtcloseall[]

Closes all files ever inserted into a TXDT buffer.

Because of PMAP considerations (see "Caveats About Files"), txdtclosef and txdtcloseall are the only proper ways to close files that have been inserted into a TXDT buffer.

The rest of this section pertains to TXDT-10 only.

A buffer may not be used while any file inserted in it is closed. An attempt to use such a buffer may cause arbitrary chaos, including the dreaded TRAP AT LOCATION.

txdtunpmap[]

UnPMAPs all files currently mapped in by TXDT. After this operation they may be closed normally.

txdtsubstjfns[alist;buflst]

alist must be a list containing elements of the form (oldjfn . newjfn) where oldjfn is the JFN of a file that was previously inserted into some buffer and then closed and newjfn is the JFN assigned when that file was reopened. It is assumed that each file is open and was reopened in exactly the same mode (eg INPUT, OUTPUT, BOTH).

buflst must be a buffer, a nonempty list of buffers, or NIL which means all unkilled buffers.

txdtsubstjfns simultaneously substitutes each new JFN for every occurrence of each old JFN in each buffer in buflst.

For example, suppose it is desired to make a SYSOUT of a TXDT job. One should first save the names, JFNs, and modes of every open file. Then the files should be closed with txdtclosef and the SYSOUT made. Upon restarting that SYSOUT, one should reopen all the files in exactly the same mode and then txdtsubstjfns the new JFNs for the old JFNs in all buffers. Such a sysout facility is not provided explicitly in TXDT because it is expected that the editor implementor will have additional invariants to maintain.

Caution: buffers are not the only TXDT objects that reference JFNs. Both grabbed objects and undo information may contain JFNs. Thus, after a SYSOUT of the kind described above, previously grabbed objects must not be used nor should previously executed operations be undone. This limitation has been found acceptable because txdtsubstjfns is usually used merely to save an initialized core image for user convenience. Editing sessions are generally saved as partially edited files.

14. Modification Functions

txdtinsert[object;addr;behind;countlc;oldbox]

Inserts the characters of object in front of the character referenced by the address addr. If behind is NIL, txdtinsert returns the address of the first character inserted. If behind is the atom BOTH txdtinsert returns a dotted pair, consisting of the address of the first character inserted and the address immediately following the last character inserted. Otherwise, it returns the address of the character immediately following the last character inserted.

If oldbox is a boxed address, it is reused (i.e., its fields are smashed) to represent (one of) the answer address(es).

Before object is inserted, the message governing addr is stored at addr (creating two adjacent regions with identical messages). Then the characters in object are inserted. Finally, if txdtescapechar is nonNIL and object is not a grabbed object, any message sequences in it are processed and deleted as described in the section "Messages and Fonts".

countlc may be used to determine the number of lines or characters inserted. If countlc is CHARS, the global variable txdtdelta is set to the number of characters in the insertion. If countlc is LINES, the total number of lines (line terminations) in the insertion is counted and put in txdtdelta. If countlc is BOTH, txdtdelta is set to (l . c) where l is the number of lines inserted and c is the number of characters inserted on the last line. If countlc is NIL, txdtdelta is left unchanged. Note that characters and lines are counted after any message sequences have been processed. If object is a grabbed object, it must not currently be inserted or the error ATTEMPT TO REINSERT INSERTED GRABBED OBJECT will be generated.

If addr is a boxed address, the address of the character immediately following the last character inserted is txdtequal to addr: they reference exactly the same character. However, the two addresses will not be eq and the address returned by txdtinsert will be slightly more efficient to use after the insertion.[7]

A grabbed object is actually a segment of the buffer which has been removed. It represents the text contained in that segment and may be inserted with the effect of inserting that text. However, a grabbed object may be inserted only once, because it contains pointers which link it to the adjacent text and each of these can only point to one place. When a segment of text is grabbed, it is deleted from the buffer. When it is deleted, it is marked as such. When a grabbed object is given to txdtinsert, a check is made to insure that it is marked as deleted. If not, an error is generated. If the object is marked, txdtinsert unmarks it before inserting it. The process of unmarking prevents the object from being inserted a second time. When a window is grabbed, the messages in it remain attached to their respective addresses. Thus, the message regions in a window are preserved when the window is grabbed and reinserted.

7. This is explained in the section "Implementation Functions". The toy block model of the buffer and boxed addresses fails to provide an analogy here. Essentially though, after the insertion, addr is slightly out of date. However, it contains sufficient information to (1) make it easy to detect that it is out of date, and (2) decode it in the new context.

Note that since the message initially governing addr is stored at addr before the insertion, the last message in object does not "spill over" to characters beyond addr. Note also that if object is not a grabbed object and contains no message sequences, then its characters are all governed by the message initially governing addr.

txdtsubst[object;str;$addr_1$;$addr_2$;back;count;countlc;oldbox]

Substitutes object for str in the window defined by $addr_1$ and $addr_2$. The error ILL-DEFINED WINDOW occurs and no changes are made if the window is not well-defined. If back is NIL, the scan starts at $addr_1$ and proceeds forward to $addr_2$. If back is nonNIL, the scan starts at $addr_2$ and proceeds backward to $addr_1$. If count is NIL or negative, every occurence of str in the window is replaced by object. Otherwise, count is assumed to be the maximum number of substitutions allowed. countlc may be used to determine the total change in the number of lines or characters in the buffer due to the substitution. If countlc is CHARS, the total change in the number of characters is put in the global variable txdtdelta. If countlc is LINES, the total change in the number of lines is put in txdtdelta. If countlc is BOTH, txdtdelta is set to (l . c) where l is the total number of lines inserted and c is unspecified. If countlc is NIL, txdtdleta is not changed.

txdtsubst returns the address of the final substitution made. If proceeding forward, this will be at the end of the final insertion of object. If proceeding backward, it will be at the beginning of the final insertion of object. If oldbox is a boxed address, it is reused to represent the answer address. Finally, the global variable txdtsubstcnt is set to the total number of substitutions actually made.

Object may be any insertable object; however recall that a grabbed object can only be inserted once. If str is not a string, it is converted to a string with mkstring. $addr_1$ and $addr_2$ default to TOP and BTM respectively.

Note that txdtsubst[object; str; $addr_1$; $addr_2$] replaces all occurrences of str in the window by object, while txdtsubst[object; str; $addr_1$; $addr_2$;NIL;1] replaces the first occurrence of str by object.

The empty string cannot be substituted for. (It can be found everywhere.) If str is empty, the results are as if it could not be found in the window. Similarly, str is not searched for within the newly inserted object.

If object is empty, the effect is that of deleting the appropriate occurrences of str.

In order to detect whether any substitutions were performed, inspect txdtsubstcnt. It will be 0 if str was not found (or if count was 0 to begin with). In the case where no substitution is made, the resulting address is that of the starting point, either $addr_1$ or $addr_2$, depending on the direction of the scan.

If the countlc option is used, txdtdelta will be set to the total number of lines or characters gained or lost due to the substitution. If txdtdelta is negative, that many lines or characters were lost from the buffer. If txdtdelta is positive, that many were gained. Note that if txdtdelta is 0, it means there was no net change in the number of lines or characters; however, it does not mean that no substitutions were made.

txdtdelete[$addr_1$;$addr_2$;countlc;oldbox]

Deletes the text in the window from $addr_1$ to $addr_2$. If countlc is CHARS, the global variable txdtdelta is set to the negation of the number of characters deleted. If countlc is LINES, txdtdelta is set to the negation of the number of lines deleted. If countlc is BOTH, txdtdelta is set to (l . c) where l is the negation of the number of lines deleted and c is unspecified. If countlc is NIL, txdtdelta is unchanged. The function returns the address of the character immediately following the last character deleted (reusing oldbox if it is a boxed address). If the window is not well-defined, an ILL-DEFINED WINDOW error is generated and no deletion occurs. $addr_1$ defaults to TOP and $addr_2$ defaults to BTM.

If $addr_2$ is a boxed address, the result returned by txdtdelete will always be txdtequal to $addr_2$. However, it will usually be slightly more efficient to use the address returned rather than $addr_2$ after the deletion.[8]

When text is deleted it is marked as such. Any subsequent attempt to use an address referencing deleted text will cause an error.

txdtgrab[$addr_1$;$addr_2$;countlc;oldbox]

Deletes the text in the window from $addr_1$ to $addr_2$ and returns it as a "grabbed object". If the window is not well-defined, the error ILL-DEFINED WINDOW is generated, and no deletion occurs. countlc functions just as in txdtdelete. The global variable txdtgrabaddr is set to the address of the character immediately after the last character deleted (reusing oldbox if it is a boxed address). $addr_1$ defaults to TOP and $addr_2$ defaults to BTM.

Grabbed objects can be recognized by the predicate txdtgrabbedp. They are not addresses and may not be used as such.

A grabbed object represents the text in the window deleted, in the sense that it may be inserted with txdtinsert with the same effect as inserting a string containing the text in the window. In addition, any addresses into the window will be valid once the grabbed object is inserted, and all messages in the window are preserved.

However, a limitation is that a grabbed object may only be inserted once. This is because it contains pointers that are used to link it to the text on either side of it in the buffer, and these can only point to one place. txdtinsert will cause an error if an attempt is made to insert a grabbed object that has already been inserted.

If it is necessary to move text from one place to another, txdtgrab is exceedingly well suited. Since the text is already in the form required for insertion into a buffer, the only cost is boxing it up long enough to pass it to the user.

If it is necessary to insert the text several times, txdtgrab is not so well suited. Suppose the task is to copy a given window and insert the copy elsewhere, leaving the original in place. In general, this can be done in several ways with TXDT functions. The function txdtprint may be used to print

8. See the discussion of this under txdtinsert or the section "Implementation Functions".

the window to a file and the resulting file segment may then be inserted as many times as desired. Less efficiently, the function txdtmkstring can be used to generate a string containing the text in the window, and that string may then be inserted as many times as desired. A third way is to txdtgrab the window, use txdtcopy to copy the grabbed object, then insert the original grabbed object in one place and the copy in the other. (Addresses into the grabbed object will point into the grabbed object only and not into the copy.)

Which of these methods is most efficient in terms of storage depends on the amount of text involved and the extent to which it has been modified since it was originally read or inserted. In most circumstances, the txdtprint method will be the most efficient. The section "Implementation Functions" should make the details clearer.

It should be recalled that the context of this discussion is that more than one copy of a window is to be inserted. txdtgrab is unexcelled if the text is to be merely moved to a new location since it requires virtually no overhead.

txdtgrabbedp[x]

Returns GRABBED if x is a grabbed object that is not currently inserted. Returns GRABBED&INSERTED if it is a grabbed object that is currently inserted. Returns GRABBED&UNDONE if it is the result of an undone grab. If x is not a grabbed object, it returns NIL.

15. Addressing Functions

The following functions may be used to obtain and manipulate addresses.

txdtfind[x;$addr_1$;$addr_2$;back;behind;count;anchor;oldbox]

If anchor is NIL, txdtfind searches for the countth occurrence of the string x (or mkstring[x] if x is not a string) in the window from $addr_1$ to $addr_2$. If back is NIL, the search proceeds forward from $addr_1$ to $addr_2$ (or the bottom if $addr_2$ is not encountered). If back is nonNIL, the search proceeds backwards, from $addr_2$ to $addr_1$ (or the top if $addr_1$ is not encountered).

If anchor is nonNIL, txdtfind determines if x occurs starting at addr, (or, if back is nonNIL, ending at $addr_2$). Such an occurrence must be entirely within the window defined by $addr_1$ and $addr_2$. count is ignored.

If the appropriate occurrence of x is found, the address of the beginning or end (or both) of that occurrence is returned. If behind is NIL, the address of the first character is returned. If behind is BOTH, a dotted pair, consisting of the address of the first character and the address of the character immediately following the last character, is returned. Otherwise, the address of the character immediately following the last character is returned. In all cases, oldbox is used to represent one of the answer addresses if oldbox is a boxed address.

If an appropriate occurrence of x is not found within the window, NIL is returned.

Whether or not count occurrences of x are found, txdtfindcnt is set to the number of occurrences found.

$addr_1$ and $addr_2$ default to TOP and BTM respectively. count defaults to 1.

It should be pointed out that only characters strictly in the window are inspected. Recall that the character at $addr_2$ is not in the window, and hence, is not inspected. That is, if a forward search must look at the character at $addr_2$, then the search has failed; a backward search begins at the character immediately preceding $addr_2$.

If x is the empty string or count is 0, the search will succeed immediately at its starting point.

If count is negative, back is negated, and abs[count] occurrences are found. Thus, the direction of the search can be specified with either back or the sign of count.

txdtmove[n;c;addr;flg;oldbox]

Roughly speaking, this function returns the address of the character arrived at by starting at addr and moving past n lines and then past c characters. Precisely: If n is a positive integer, the "line move" from addr moves past n line terminations, and stops immediately after the nth one. Then, c characters are counted off, backwards or forwards, depending on the sign of c. If n is a non-positive integer, abs[n]+1 line terminations are counted off in the backward direction, stopping immediately to the right of the last one. Then the same type of character move described above is made. In all cases, oldbox is reused to represent the answer address if oldbox is a boxed address.

If n is NIL, no line move is made; the resulting address is just c characters from addr. If c is NIL, 0 is used (i.e., no character move is made). If addr is NIL, it defaults to TOP or BTM according to n and c: If n is zero or positive, addr defaults to TOP. If n is negative, addr defaults to BTM. If n is NIL, addr defaults to TOP when c is zero or positive, and to BTM when c is negative.

Finally, if the move would exceed the bounds of the buffer, the result returned depends on flg. If flg is T, NIL is returned. If flg is NIL, an address txdtequal to either TOP or BTM is returned, depending on which was exceeded. If flg is the atom BOUNDARYERR, an error is caused (via ERROR!).

Note that when n is 0, the line move is backwards to the first character in the line containing addr. Thus, txdtmove[0;0; addr] is the address of the beginning of the line containing addr, txdtmove[1;0; addr] is the address of the beginning of the previous line and txdtmove[1;0; addr] is the address of the following one.

When n is NIL, txdtmove is just a character mover. txdtmove[NIL; 5;addr] is the address of the character 5 to the left of the one at addr, and txdtmove[NIL;5;addr] is that 5 to the right. Note that the character move is not bounded by the current line. Thus, txdtmove[NIL;5000] is the address of the 5001st character in the buffer, and txdtmove[NIL; 5000] is the address of the 5000th character from the end. (Note how addr defaults to TOP in the first case, and BTM in the second.)

flg should be set to nonNIL in situations in which a positive check on the validity of the move is desired. That is, if the buffer only has 100 lines in it, and the user wishes to move past 200 lines, then txdtmove[200;NIL; addr] will move to the bottom (immediately behind the last character in the file), while txdtmove[200;NIL; addr;T] will return NIL.

Character moves are very efficient, since the characters may be counted off a page (2560 characters) at a time. To make line moves, the characters must be inspected.

In TXDT-10, one bizarre aspect of txdtmove is that it avoids returning an address that points to the line feed of a CR/LF pair. If after the character move, the character addressed is such a line feed, txdmove moves forward (or backwards, depending on the direction it has been moving) one additional character so as not to leave the address between the CR and its LF.

The function txdtmove is used to move over a given number of lines from a given place in any buffer. The following function, which is just a special case of txdtmove, is used to find the address of a particular line in the current buffer.

txdtgoto[n;c;flg;oldbox]

Returns the address of the cth character from the beginning of the nth line in the current buffer. The lines are counted from the top. If n is NIL, no line move is made and the address returned is that of the cth character, counted from the top. The character move is backwards or forwards depending on the sign of c. If c is NIL, no character move is made. flg plays the same role as in txdtmove, permitting a check on whether the move exceeds the limits of the current buffer. When oldbox is a boxed address, it is used to return the answer address.

Because of the convention of numbering things starting at 1 rather than 0, the nth line is arrived at by moving past n 1 line terminations. A similar remark can be made about the cth character. Thus, the 1st character in a line is the one arrived at by not moving from the beginning of the line. Therefore, it follows that the 0th character of a line is actually the result of moving back one.

txdtcountlc[$addr_1$;$addr_2$;charflg]

Counts the number of lines and/or characters between $addr_1$ and $addr_2$. If charflg is the atom CHARS, the result is an integer, which is the number of characters separating the two addresses i.e., the character count one must give txdtmove to move from $addr_1$ to $addr_2$. If charflg is the atom LINES, the result is the number of line terminations between the two addresses. Otherwise, the result is a dotted pair, car of which is the number of lines, and cdr of which is the number of characters from the beginning of the last line to $addr_2$. The two addresses default to TOP and BTM respectively.

Thus, txdtcountlc can be used to find out "where" a given address is. Note however that txdtcountlc counts the number of lines and characters between the two addresses; it does not return a line number when $addr_1$ happens to be TOP. Thus, if txdtcountlc[TOP; addr] is (45 . 4), then addr points to the 5th character from the beginning of line number 46. Interpreted the other way, it means there are 45 line terminations between addr and (in this particular case) TOP. And there are 4 characters separating the last line terminator and addr itself.

txdtbox[addr;flg;oldbox;float]

This function returns a boxed address equivalent to addr, reusing oldbox if oldbox is a boxed address.

If addr is an unboxed address, it is decoded with respect to the current buffer. flg is used during the decoding in precisely the way flg is used by txdtmove to permit a positive check on whether the bounds of the current buffer have been exceeded.

If addr addresses the bottom of its buffer and float is nonNIL, the boxed address returned is very special: the address is actually tied to the last character of the buffer but has a mark on it that causes it to be moved forward by one whenever it is used. The first time such an address is used and the move forward does not touch the bottom, the address is smashed so that it is afterwards attached to the character encountered.

Normally a boxed address pointing to the bottom of its buffer will forever point to the bottom. (Imagine the buffer as a list; boxed addresses point to successive cdrs; a boxed version of the bottom address is analogous to NIL and denotes the end of the list no matter how many insertions and deletions are made.) A boxed version of the bottom address with the float flag on "floats" off the bottom as inserts are made at the bottom.

Calling txdtbox on a boxed address (with the float flag off) has two uses. First, it is a way to smash oldbox with the contents of addr. Second, if the region addressed by addr has been heavily modified since addr was obtained, the result of txdtbox will be equivalent to addr but more efficient to use. See the section "Implementation Functions".

txdtbox is the function used by all TXDT functions to decode line and character addresses. When other TXDT functions call txdtbox, they specify flg and float to be NIL. Thus, if the current buffer contains 100 lines, a line address of 200 is equivalent to BTM.

txdtbox has an optional feature which makes the decoding of lines and character addresses more efficient in the vicinity of a particular point in the current buffer. If the global variable txdtpoetflg is nonNIL, the feature is activated. In this case, the global variable txdt$ is assumed to be set to the number of line terminations in the current buffer. The global variable txdtpoetdotaddr is assumed to be set to some boxed address in the current buffer and txdtpoetdot is assumed to be set to a pair (l . c) giving the line and character location of txdtpoetdotaddr. (That is, txdtpoetdot is an unboxed address equivalent to txdtpoetdotaddr.)

If txdtpoetflg is set, then whenever a line and character address is used, the necessary number of lines are counted from the TOP, BTM, or txdtpoetdotaddr, whichever is the fewest number of lines away. However, this efficiency requires that the implementor properly maintain the globals txdt$, txdtpoetdot and txdtpoetdotaddr as insertions and deletions are made into the buffer. (The insertion and deletion functions can be made to count the number of lines and characters added or deleted.) See txdtcurbuff.

When txdtpoetflg is NIL, line and character addresses are always counted from the top.

txdtunbox[addr;charflg;flg;oldpair]

This function returns the line and character address of the character referenced by addr. If charflg is T, a pair of the form (NIL . c) is returned, where c is the character number of the given character. Otherwise, a pair of the form (n . c) is returned, where n is the line number of the line containing the character and c is the character number within that line. If addr is itself a line and character address, then flg is used to monitor boundary errors while it is being decoded. If oldpair is a listp, it is smashed to represent the answer pair. In TXDT-10, if the car of oldpair is a big number box, it is smashed to represent the line count.

If txdtpoetflg is set and addr is in the current buffer, txdt$, txdtpoetdotaddr and txdtpoetdot are used to reduce the number of lines counted. Note that if addr is not in the current buffer, then the line and character address returned is useless until addr's buffer is made current.

One reason for calling txdtunbox on an unboxed address is to "normalize" the line count or to convert a character address to a line address or vice versa. For example (7 . 100) may actually be an "unnormalized" address equivalent to (8 . 20).

txdtcopy[x]

x may be an address or a grabbed object. It is copied and returned.

The user should copy addresses if he wants to call a TXDT function that smashes the address but wants to keep a copy of the original. It is as efficient to use a copy of an address as to use the original.

Grabbed objects must be copied if they are to be inserted more than once. If txdtcopy is given a grabbed object that has already been inserted, it will cause an error (because at that time it is impossible to reconstruct the limits of the window grabbed). Copying a grabbed object is similar to copying a list at the top level only. Since references to the characters are second level pointers, such an object can be copied without inspecting the characters it "contains". Note that boxed addresses into a grabbed object do not point into copies of the object.

It is merely notational economy to supply one function for copying both kinds of objects. Addresses should not be confused with grabbed objects.

txdtaddrp[x]

Returns T if x is a boxed address and NIL otherwise.

txdtequal[$addr_1$;$addr_2$]

The equality predicate for addresses. Returns T if the two addresses reference the same character, and NIL otherwise.

txdtgreaterp[$addr_1$;$addr_2$]

Returns T if the character referenced by $addr_2$ strictly follows that referenced by $addr_1$. Otherwise returns NIL.

txdtvalidp[addr;flg]

This function decodes addr and returns T if it is a valid address. If addr is not an address or references deleted text, the function returns NIL. If flg is the atom BOUNDARYERR and addr references a non-existent line or character (e.g., line 200 while the current buffer has only 100 lines), the function returns NIL. If flg is anything else and addr references a non-existent line or character, T is returned (since such a use of addr would default to TOP or BTM).

txdtclosest[addr;addrlst]

addr is supposed to be an address and addrlst is supposed to be a list of addresses. txdtclosest searches addrlst and returns either the address on addrlst which is closest to addr (in the sense of being in the same buffer and separated by the fewest number of characters)

or returns the atom TOP or BTM if one of those two addresses is closer than any address on addrlst. In all cases txdtclosestforwflg is set to T if the answer is forward from addr and NIL otherwise.

Note that if addr points into a buffer, buf, other than the current one and txdtclosest returns TOP or BTM, then that result cannot be used to address buf until buf is made current.

16. Printing Functions

txdtprint[$addr_1$;$addr_2$;ptraddrs;ptrchars;file;mask]

Prints the text in the window from $addr_1$ to $addr_2$ to the file file.

ptraddrs and ptrchars may be used to mark certain characters in the window. ptraddrs is supposed to be a list of addresses in ascending order (as ordered by txdtgreaterp). ptrchars should be a list of atoms or strings. Whenever a character indicated by an address on ptraddrs is about to be printed, the car of ptrchars is printed to file (with prin1) first and ptrchars is cdred. If ptrchars is NIL, txdtptrchar is printed. If ptrchars and txdtptrchar are NIL, nothing is printed in response to finding the indicated character.

If ptraddrs is an address rather than a list of addresses, ptraddrs is set to list[ptraddrs]. If ptrchars is an atom instead of a list, txdtprint acts as though ptrchars were a list as long as ptraddrs each element of which was the given ptrchars.

If mask is nonNIL, it is assumed to be an integer and is treated as an n-bit mask (n =36 in TXDT-10 and n =32 in TXDT-D). If the ith bit (counting from 0 on the left) is on, then when the character about to be printed has i as its ascii code, the character is not printed. Thus, mask may be used to mask out of the printing all occurrences of any of the first n ascii characters. This includes the standard control characters. If mask is NIL, 0 is used (which means no character is masked out).

Finally, whenever the character about to be printed is governed by a nonNIL message (different from the message governing the previous character) and txdtescapechar is nonNIL, the function txdtprintuserfn is applied to the message and either the JFN of file (in TXDT-10) or the full filename of file (in TXDT-D). If txdtprintuserfn returns a value other than NIL and that value is not the message it was given, the value is treated as a message and overwrites the message previously stored at that address. All messages are compared using eq. txdtprintuserfn is initally defined as a noop but may be defined by the user. If the variable txdtprintuserfnbox is nonNIL, it is assumed to be a boxed address and is smashed to contain the address of the character about to be printed. This address may be inspected by txdtprintuserfn.

txdtprint returns the last character printed.

Note that simply txdtprint[$addr_1$;$addr_2$] will print the text in the window to the terminal.

If the user wishes to print a mark indicating the location of some address in the window, ptraddrs may be used. For example let the text be:

> Tantallon is on a high cliff above the North Sea, two miles south of the village of North Berwick. Opposite the castle, two miles out to sea, loom the white cliffs of the Bass Rock. On approaching the castle, the visitor is struck by the size of the curtain wall, stretching nearly 300 feet across the promontory, and nearly 50 feet tall.

Let addr point to the "s" in "visitor", and let txdtptrchar be bound to the atom _. Then

```
txdtprint[ txdtmove[ 1;0;   addr]
           txdtmove[2;0; addr]
           addr]
```

would print:

> Berwick. Opposite the castle, two miles out to sea, loom the white cliffs of the Bass Rock. On approaching the castle, the vi_sitor is struck by the size of the curtain wall, stretching nearly 300 feet across the promontory, and nearly 50 feet tall.

If the user wishes to print a "cursor" in some other way, he must do that with multiple calls to txdtprint or by using the function txdtmapchars.

txdtwrite[file;$addr_1$;$addr_2$]

> Writes the text in the window from $addr_1$ to $addr_2$ to a new version of the file named file and returns the full file name of the file created. The addresses default to TOP and BTM.
>
> If txdtescapechar is nonNIL, then whenever the character about to be printed is governed by a different (neq) nonNIL message from the previous one, an appropriate message sequence is written out before the character.

If file does not include an extension, the default extension is the setting of the global variable txdtextension. This is initially NIL, which means no extension is supplied.

When txdtwrite is called, a new file is created and opened. Pages for this new file are created by PMAPping them in and depositing bytes from the buffer window. When done, the file size is computed, and the file is closed. txdtwrite is considerably faster than txdtprint.

If it is desired to write to an already open file or to not close the file when done, txdtprint with the appropriate JFN should be used.

If txdtwrite is interrupted (by CTRL-E or an error) after it has begun writing and before it has finished, the file is closed and deleted. The deleted file should not be undeleted by the user since the size field in its file descriptor block will not have been set.

If many changes have been made since the last write, it might be considered wise to use txdtwrite to save the current version. If this is done, it would also speed subsequent searches, etc., if the buffer were then emptied and the newly produced file read back in. (This would allow the search function, for example, to scan whole pages at a time rather than spend a lot of time skipping over insertion/deletion boundaries. See the section "Implementation Functions".) However, if this is done, (1) any boxed addresses being kept would be meaningless after the old buffer had been deleted and (2) it would not be possible, after the deletion, to undo changes made prior to the deletion without undoing the deletion first. Boxed addresses could, of course, be saved by converting them to line or character addresses with txdtunbox before clearing the buffer.

txdtmkstring[$addr_1$;$addr_2$;rplstring;strptr;mask]

Makes a string out of the characters in the window from $addr_1$ to $addr_2$, and returns it. TXDT-10 converts CR/LFs wholly contained in the window into EOLs, following the conventions on INTERLISP-10 strings. The two addresses default to TOP and BTM respectively.

Messages encountered are written to the string as message sequences under the conditions described in txdtwrite. mask is used to mask out characters just as in txdtprint.

rplstring and strptr are used in the construction of the answer string. Recall that an INTERLISP string is represented by a "string pointer" which contains a byte count and points to an address in pname space where the characters are stored.

If rplstring is NIL, the string returned is composed of entirely new structure (both from pname space and string pointer space). If rplstring is T, the pname space for the answer is obtained from an internal TXDT buffer. Thus the characters in the answer string in this case will be smashed when a subsequent call to txdtmkstring is directed to use that buffer again. If rplstring is a stringp, the pname space for the answer string is that of rplstring itself. If the entire window will not fit in the space provided, all new structure is returned but the characters in a stringp rplstring may have been smashed.

If rplstring is nonNIL, and the window fits in the space provided and strptr is a stringp, then strptr is smashed to represent the final answer; otherwise a new string pointer is constructed. Note that if strptr is smashed, then the answer string is a substring of rplstring (or TXDT's internal buffer) and may be smashed by string operations on that parent string.

txdtmkstring is useful for copying text which must be inserted in several different places. (Unlike grabbed objects, strings may be inserted as many times as desired.) Note that unlike txdtgrab, txdtmkstring does not delete the text from the buffer.

17. Message Functions

txdtgetmsg[addr]

If addr is a buffer, txdtgetmsg returns the message above the top of the buffer. Otherwise, addr must be an address and this function returns the message governing addr.

txdtputmsg[addr;msg]

If addr is a buffer, this function sets the top message of the buffer to msg. Otherwise, addr must be an address and this function stores the message msg at addr. In both cases, the message replaced is returned.

txdtgetmsglst[$addr_1$;$addr_2$]

This function returns a list of pairs. Each pair is of the form ((NIL . c) . msg) where (NIL . c) is a character address and msg is the message stored at that address. There is one such pair for $addr_1$ and for each subsequent address in the window from $addr_1$ to $addr_2$ at which a message is stored. The pairs are in ascending order by address.

It is sometimes convenient to write out a buffer to a file without cluttering the text in the file with message sequences. txdtgetmsglst permits one to save elsewhere sufficient information to restore the messages after the file has been reinserted. txdtgetmsglst uses character addresses in its answer for two reasons: such addresses can be decoded quite efficiently and (in contrast to the situation obtained had boxed addresses been used) the answer list can itself be written to a file.

txdtmapmsg[$addr_1$;$addr_2$;fn;arg_2]

This function scans the window from $addr_1$ to $addr_2$. At $addr_1$ and each time it encounters an address at which a message is stored, it applies fn to the message and arg_2. Provided the result is a legal message, it is undoably stored in place of the original one. When the window is exhausted, NIL is returned.

If fn is NIL, txdtmapmsg is just txdtgetmsglst[$addr_1$; $addr_2$].

18. Character Functions

txdtchar[addr;charcodeflg;moveflg]

Returns the character at address addr. If charcodeflg is NIL, the atom having the corresponding single character as its pname is returned; if charcodeflg is T, the integer character code is returned. In TXDT-10, if the character at addr is CR, followed by LF (or LF preceded by CR), EOL is returned.

Finally, if the address supplied is a boxed address and if moveflg is non-NIL, the address is destructively modified to point to the adjacent character. If moveflg is T or 1, the address is modified to point to the character following the one returned. If moveflg is 1, the address is modified to point to the one preceding the one returned. In TXDT-10, if moving forward and the character at addr was a CR followed by a LF, then the address is incremented by two (and EOL is returned); if moving backward over a CR/LF, the address is decremented by two.

txdtchar, when moveflg is set, is analogous to gnc and glc. The ability to obtain the character at an address and simultaneously move the address pointer is useful when one must treat the buffer as a stream of characters.

Recall that the bottom of a buffer is distinguished by not pointing to any character. If txdtchar is given an addr pointing to the bottom, NIL is returned. If moveflg specifies a forward move, addr is not modified. If a backward move is specified (and addr is a boxed address) then addr is lifted off the bottom and the next txdtchar at addr will return the last character in the buffer.

There is one bizarre feature of txdtchar. If a backward move is specified and addr is a boxed address pointing to the top (i.e., txdtaddr returns the first character in the buffer), then addr is modified to be a special address that points just above the top. A subsequent call of txdtchar on that addr will produce NIL. If such a call specifies a backwards move from such an addr, addr is not modified; a forward move from such an addr makes addr point to the top again. Such a special addr will cause an illegal address error if given to any TXDT function other than txdtchar or txdtmapchars.

This special treatment of boxed addresses at the top and bottom means that one can detect that the top or bottom of a buffer has been reached by asking whether the character returned is NIL.

txdtmapchars[$addr_1$;$addr_2$;charcodeflg;moveflg;untilfn]

This function scans the window defined by $addr_1$ and $addr_2$ and successively applies untilfn to each character (or if charcodeflg is T, character code) encountered. If moveflg is not 1, the scan starts at $addr_1$ and moves to $addr_2$ (or the bottom if $addr_2$ is not encountered). If moveflg is 1, the scan starts just before $addr_2$ and moves backwards to $addr_1$ (or the top).

When untilfn returns nonNIL, the scan stops and txdtmapchars returns the address of the character just beyond the one upon which untilfn returned nonNIL. (That is, the answer address is incremented (or decremented) before untilfn is called.) If untilfn never returns nonNIL, the scan stops when the window is exhausted and the address returned is just beyond the last character seen. In all cases, if $addr_1$ is a boxed address, it is reused to represent the answer.

Warning: There are limitations on which TXDT functions can be called by the mapping function untilfn. In particular, txdtfind, txdtsubst, txdtwrite and txdtmapchars should not be called by untilfn. An error may occur if one of these functions is called inside txdtmapchars. If no error occurs, txdtmapchars works correctly. The problem arises from the fact that each of these functions ties down one or more of TXDT's PMAP swapping buffers and when called in combinations they exhaust the supply.

txdtmapchars will accept and may produce a special address above top (as described above in connection with txdtchar). txdtmapchars treats that address just as txdtchar does: the "character" seen there is NIL. Note that txdtmapchars moves forward if moveflg is NIL while txdtchar does not move at all in that case.

In TXDT-10 txdtmapchars is roughly 20 times faster than the equivalent function written in terms of txdtchar.

txdteolp[]

Returns T or NIL according to whether the last two characters in the current buffer are CR/LF (TXDT-10) or EOL (TXDT-D).

txdteolp is somewhat more efficient than the equivalent functions defined in terms of txdtchar. In fact, if no modifications have been made at the bottom of the buffer since the last time txdteolp was called, the computation involves only comparisons and three global variables.

19. Implementation Functions

A buffer is actually a two-way list of records. Each record specifies some sequence of characters, a message, and the two adjacent records. The concatenation of all of the character sequences is what appears as the text in the buffer.

The character sequence in a record is specified by three fields: The first contains a file name and file page number.[9] This component therefore points to some source of characters. The other two components specify what segment of this source is to be considered "in" this record. These other two components are just integers that give the offsets from the beginning of the page of the beginning and end of the segment.

For example, let #1 and #2 represent file names and page number pairs that point to the following character sources.

#1 = ABCDEFGHIJKLOMNOP . . .

#2 = abcdefghijk . . .

#3 = 012345 . . .

Let the buffer consist of the following four records (named as indicated):

T_1 = (**TOP** 0 0 msg_1)

T_2 = (#1 0 9 msg_2)

T_3 = (#2 2 11 msg_3)

T_4 = (**BTM** 0 0 0)

There are two special records which represent the beginning and end of the buffer. (It is important to remember that the records are linked with pointers. For example, T_3 points "forward" to T_4 and "backward" to T_2, and they reciprocate.) This buffer would print as:

ABCDEFGHIcdefghijk

The top message of the buffer is msg_1. The characters ABCDEFGH are governed by msg_2 and the remaining characters by msg_3.

The boxed address of TOP is a pair containing T_2 and 0. The boxed address of BTM contains T_4 and 0. The special address above TOP contains T_1 and 0.

The boxed address of the character "C" contains T2 and the integer 2.

9. In TXDT-10, the file name is encoded as a JFN.

In order to insert the string "012345" in the buffer at the address of "C", we would change the buffer to the following:

T_1 = (**TOP** 0 0 msg_1)

T_2 = (#1 0 2 msg_2)

T_5 = (#3 0 6 msg_2)

T_6 = (#1 2 9 msg_2)

T_3 = (#2 2 11 msg_3)

T_4 = (**BTM** 0 0 0)

Note that we had to "break" T_2 into two parts, one to contain characters 0 to 2, and the other to contain 2 to 9. We created a new record for the second part, T_6. This would not have been necessary had the insertion occurred at the beginning or end of a record. Note also that a record, T_5, was created to hold the string inserted. This buffer would print as:

AB012345CDEFGHIcdefghijk

It should now be clear that in order to undo this insertion, we merely need to restore the third component of T_2 to 9, and relink T_2 and T_3.

This example illustrates why it is more efficient to use the boxed address returned by txdtinsert than to use the old boxed address of "C" after the insertion. If behind were T in the call to txdtinsert, the address returned would be a box pointing to the character in position 2 of T_6. Recall that before the insertion the boxed address of "C" is a box pointing to the character at position 2 of T_2. But after the insertion, 2 is not a legal position in T_2, because after the insertion, T_2 contains only the characters at positions 0 and 1. If given this boxed address however, the low-level TXDT address decoding function can quickly recognize that it is out of date, since the position specified in the boxed address is not consistent with the offset information stored in the triple pointed to by the boxed address. To decode the address, the TXDT decoding function looks for a record with the indentical first component (in this case a first component of #1) with offset components that include the position in question. In the current example, this would yield the triple T_6.

Now let us illustrate a deletion. Suppose we wish to delete the segment "EFGHIcde" from the buffer shown above. This creates the buffer:

T_1 = (**TOP** 0 0 msg_1)

T_2 = (#1 0 2 msg_2)

T_5 = (#3 0 6 msg_2)

T_6 = (#1 2 4 msg_2)

T_3 = (#2 5 11 msg_3)

T_4 = (**BTM** 0 0 0)

This buffer prints as:

AB012345CDfghijk

Note that in this case all we had to do was reset the terminal offset of T_6 and the initial offset of T_3. In general, we might also have to relink two records so as to delete a segment of records.

It is clear how to undo a delete. However, consider what happens if we undo the insertion without undoing the deletion. Recall that undoing the insertion consists of restoring the third component of T_2 to 9, and relinking T_2 and T_3. If we do this we get a buffer which prints as:

ABCDEFGHIfghijk

which shows that in addition to the insertion, part, but not all, of the deletion was undone. (The "EFGHI" was restored, but the "cde" was not.) Undoing out of order is therefore dangerous, since it can affect text that does not overlap or even adjoin with the text modified by the operation undone.

From this discussion of the representation, it should be obvious that insertion and deletion are fairly high level operations that are not concerned with the number of characters involved. This is why reading in large files, for example, is so fast. The work must be done by those functions which must scan this structure and pretend it is a continuous stream of characters.

In order to make this pretense reasonably fast in TXDT-10, these functions are all written in LAP. These functions include the search function, the character and line counters, the printing functions, and others. These functions construct byte pointers into the segment of text contained in the current triple, and use byte manipulation instructions to inspect successive characters. When the end of the segment is reached, they climb over to the next record and continue.

When these functions start on a new record, they call a special function that returns a base address for the source of characters for that record. This base address is used by the processing function to create byte pointers. The function that computes this base address asks if the relevant page of the relevant file is currently PMAPped in, and if so, returns the base address of the scratch page it is on. If it is not currently in, it is brought into a free scratch page and the address of that page is returned to the processing function.

Note that the efficiency of window scanning functions is diminished as the text in a buffer gets fragmented into records. At most one page of text may be contained in a record (so that the offsets may be coded as small integers). Thus each page of inserted text requires a new record. Since the only way to store a message is in a record, every txdtputmsg or message sequence in inserted text requires a new record. Finally, every insertion and deletion may cause records to fragment. The function txdtcontigify is provided to overcome this problem.

The details of this representation were worked out and implemented by Robert S. Boyer and J Strother Moore, in the "77 Editor", at the University of Edinburgh in August, 1972.

The following functions expose a certain amount of the implementation of TXDT.

txdtpiece[addr;oldbox]

Returns the address of the first character contained on the record containing addr. Reuses oldbox if oldbox is a boxed address.

txdtnextpiece[addr;oldbox]

Returns the address of the first character contained on the record after the one containing addr. Reuses oldbox if oldbox is a boxed address. If addr points to the bottom, the answer is equivalent to addr.

txdtprevpiece[addr;oldbox]

Returns the address of the first character contained on the record before the one containing addr. Reuses oldbox if oldbox is a boxed address. If addr is in the record containing the first character of the buffer, then the answer points to the first character of the buffer.

Because of the implementation of messages, txdtpiece is a way of finding the address at which the message governing a given address is stored. txdtprevpiece and txdtnextpiece permit one to move to the next possible address at which the message might change.

txdtcountpieces[$addr_1$;$addr_2$]

Counts the number of pieces in the window from $addr_1$ up to *and including* $addr_2$.

txdtcontigp[$addr_1$;$addr_2$;oldcons]

If all the characters in the window from $addr_1$ to $addr_2$ are contained contiguously on some file, this function returns a pair of the form (file . c) where file is the full file name of the file containing the window and c is the byte position (a la setfileptr) of the first character in the window. If the window is not contiguous on one file, NIL is returned. When a pair is returned, oldcons is reused (if it is a listp) to represent the answer and its cdr is smashed with setn if it is a big number.

txdtcontigify[$addr_1$;$addr_2$;file;behind;oldbox]

Writes the window from $addr_1$ to $addr_2$ to the end of the file file, deletes the window and inserts the just written text. Returns the address of the beginning or end of the new window (according to whether behind is NIL or T), reusing oldbox if it is a boxed address.

Note that txdtcontigify is a way to minimize the number of pieces in a window or buffer. However, any boxed address into the window (including $addr_1$) will be invalid after the operation since that area was deleted. $addr_2$ will be unaffected since it is (just) outside the window. $addr_1$ can be saved by setting behind to NIL and using $addr_1$ for oldbox.

In general, boxed addresses into the window can be saved by the user by unboxing them (with charflg T for maximum efficiency) before the operation and boxing them after. See txdtunbox and txdtbox.

Note that if the window contains messages and TXDTESCAPECHAR is nonNIL, then the window "contigified" will not actually be txdtconfigp.

txdtfileposition[addr;oldcons]

Returns a pair (file . c) where file is the full file name of the file containing the character at addr and is the byte position of that character in file. If oldcons is a listp, it is reused to represent the answer cons and its cdr is smashed with setn if it is a big number.

20. Variables

The following variables are of interest to the normal user. The reader should see the descriptions of the relevant functions for more precise specifications of the roles of these variables.

txdtdelta set by the functions described in the section "Modification Functions" to indicate how many lines and characters were added or lost due to a given insertion or deletion.

txdtextension the default file name extension for files created by txdtwrite. Initially NIL, which means no extension.

txdtptrchar the default pointer character used by txdtprint. Initially ^.

txdtsubstcnt set by txdtsubst to indicate how many substitutions were made.

txdtpoetflg used to speed up the decoding of line and character addresses in the vicinity of a "cursor" in the current buffer. See the section "Buffers".

txdtpoetdot used to speed up the decoding of line and character addresses in the vicinity of a "cursor" in the current buffer. See the section "Buffers".

txdtpoetdotaddr used to speed up the decoding of line and character addresses in the vicinity of a "cursor" in the current buffer. See the section "Buffers".

txdt$ used to speed up the decoding of line and character addresses in the vicinity of a "cursor" in the current buffer. See the section "Buffers".

txdtclosestforwflg indicates the direction from addr of the closest address found by txdtclosest.

txdtfindcnt set by txdtfind to the number of occurrences of the search string found.

txdtscratchfile always set to the full file name of the temporary file to which TXDT prints strings prior to insertion. The user may also write text in this file provided the file pointer is manually set to the bottom of the file before each write. The file must always be open for both input and output. The user may set the file pointer arbitrarily to read from the file and may leave the file pointer anywhere.

txdtcurbuflst set to the list of all buffers constructed and not yet killed.

txdtcurbuf set to the current buffer.

txdtinsertfilekey the item used to mark lists that are to be interpreted upon insertion as file segments. See "Objects That May be Inserted". Initially a string "TXDTINSERTFILEKEY".

txdtescapechar the single character atom used to mark message sequences in text to be inserted. Initially set to NIL, meaning message sequences are not to be looked for. See "Messages and Fonts."

txdtprintuserfnbox used by txdtprint to store the current address at the time txdtprintuserfn is invoked.

21. Error Messages

Below are listed the TXDT error messages a normal user may expect to see. Since TXDT functions do not always check that their arguments are of the right type, normal INTERLISP errors will be generated under some conditions as in attempting a nonnumeric nonNIL line move. In addition TXDT has several error messages caused by internal consistency checks. These should not occur in normal usage and their generation indicates either bugs in TXDT itself or a smashed core image. Internal errors are always preceded by the phrase INTERNAL TXDT ERROR and are not listed below. Their occurrence should be reported to Moore (currently MOORE@SRI-KL) along with enough information to reproduce the error from a clean core image.

Here are the normal TXDT error messages with parenthesized explanations when necessary.

TXDT ADDRESS INTO DELETED AREA

TXDT ADDRESS NOT RECOGNIZED (caused when an object other than a boxed address, TOP, BTM, a line number, or a line and character address is passed to a TXDT function in place of an address.)

CANNOT COPY INSERTED GRABBED OBJECT

CANNOT COPY RESULT OF AN UNDONE GRAB

CANNOT COPY INVALID ADDRESS

ATTEMPT TO REINSERT INSERTED GRABBED OBJECT

ATTEMPT TO INSERT THE RESULT OF AN UNDONE GRAB

ATTEMPT TO INSERT MYSTERIOUSLY MUNGED GRABBED OBJECT

ATTEMPT TO INSERT A SEGMENT OF FILE T

ATTEMPT TO INSERT ILLEGAL MESSAGE

ILL-DEFINED WINDOW (generated by txdtdelete, txdtgrab and txdtsubst when the two addresses supplied do not properly define a window i.e., when the second address is not greater than the first.)

TXDTWRITE INTERRUPTED FILE CLOSED AND DELETED (caused in response to an error or user interrupt while TXDTWRITE is transfering characters to the file.)

CANNOT REINSTATE KILLED BUFFER

CANNOT KILL CURRENT BUFFER WITHOUT EXPRESS PERMISSION

UNRECOGNIZED BUFFER LIST (caused by txdtsubstjfn when the buffer list supplied is neither a buffer, a listp, or NIL.)

22. Datatypes

The TXDT package declares four user datatypes.

TXDTRECORD used to represent the two-way linked lists representing text. The user should never see an object of this type.

TXDTADDR used to represent boxed addresses. The user may use type? to recognize boxed addresses (or txdtaddrp) but should not access or set the components of such a record.

TXDTGRABBEDOBJ used to represent grabbed objects. The user may use type? to recognize grabbed objects but should not access or set the components of such a record.

TXDTBUFFER used to represent a buffer. The user may use type? to recognize buffers and may access and set the following three fields TXDT$, TXDTPOETDOTADDR, and TXDTPOETDOT as described in the section "Buffers".

References

(1) R. S. Boyer, J S. Moore, D. J. M. Davies, "The 77-Editor," Technical Report 62, Department of Computational Logic, University of Edinburgh, 1973.

(2) W. Teitelman, "A Display Oriented Programmer’s Assistant," Proceedings of the 5th International Joint Conference on Artificial Intelligence, Department of Computer Science, Carnegie-Mellon University, pp. 905-915, 1977.

Index