ELECTRONIC PUBLISHING, VOL . 3(2), 3–29 (MAY 1991)

vi.iv, a bi-directional version of the vi full-screen editor

URI HABUSHA (אורי חבושה) AND DANIEL BERRY (דניאל ברי)

Computer Science Department
Technion
Haifa 32000
Israel

SUMMARY

This paper describes the semantics, design, and implementation of vi.iv, a bi-directional revision of vi, the standard, full-screen editor available on UNIX™ systems. The main goal was to produce the new program with as little change as possible to the semantics and implementation of the original vi.

תקציר

עבודה זו מתארת את הסימנטיקה, התכנון והישום של **vi.iv**, הגירסה הדו–כיוונית של **vi**, עורך הטקסטים הסטנדרטי, בעל מסך מלא, הקיים עבור מערכת ההפעלה UNIX. המטרה העיקרית היתה ליצור תכנית חדשה, עם שינוים מעטים ככל האפשר בסימנטיקה ובישום של **vi** המקורי.

KEY WORDS Bi-directional Editing Full-screen editor Multi-lingual Vi

INTRODUCTION

The authors live in a country in which bi-directional writing is a basic fact of life. The two official national languages, Hebrew and Arabic, in order of frequency of use, are written with non-Latin alphabets from right to left. Communication with the rest of the world requires use of European languages, principally English, which are written with the Latin alphabet from left to right. Even without the use of left-to-right languages, just writing so-called Arabic[1] numerals and mathematical formulae causes left-to-right writing in the midst of right-to-left text. The same situation, albeit with different national languages, exists in other Middle Eastern and Asian countries, in which Arabic, Farsi, Kazak, Kirgiz, Mongol, Uighur, Urdu, etc. are spoken[1].

Even outside the Middle East, there is an occasional need for bi-directional text. For example, text books on Middle Eastern languages written, say, in English or French, naturally have bi-directional text. This paper, containing both Hebrew and English, being written for eventual publication in an English language journal is another example.

Thus there is a need for a full complement of tools for bi-directional text processing. There already exist two batch-oriented bi-directional formatters, ditroff/ffortid™[2] and TEX™/XET[3] based on the two most popular formatters in use in the computer science world, ditroff[4] and TEX[5]. To complement these a terminal-independent, full-screen bi-directional editor is needed that runs on the same machines that support these bi-

[1] The designation of the standard numerals written in Latin alphabetic text as Arabic is misleading, as these numerals bear no resemblance to those actually written in the Arabic language.

0894–3982/91/020003–27$13.50 *Received 1 March 1990*
© 1991 by John Wiley & Sons, Ltd. *Revised 1 August 1990*

directional formatters. There exist two popular terminal-independent full-screen uni-directional editors that run on these systems, namely vi[6] and emacs[7].

There exists at least one bi-directional WYSIWYG (what you see is what you get) formatting system based on the Xerox ViewPoint system™[8]. However, our institution does not have these machines, having staked its investment in UNIX machines. On UNIX systems in Israel, there exists at least one mostly uni-directional editor, vih[9] designed for editing Hebrew text with an occasional Latin, left-to-right text item. It is not fully bi-directional in that it cannot handle left-to-right text that spans line boundaries. On the same systems, there is at least one true bi-directional editor, ded[10]. However, its authors say that it is not complete and is designed to work only with the Hebrew version of the DEC VT100. There exist a number of bi-directional editors that run on personal computers, such as alef-bet[11], Einstein[12], MacInHebrew[13], Multi-Lingual Scribe[14], Rav Ctav[15], and WORDMILL[16]. These are not usable on UNIX systems. Many of these are tied to their own WYSIWYG formatters. Additionally, many of these store the text in the order it is seen on the screen and not in the order in which it was input. (See the section on 'TIME ORDER' for a fuller discussion of the problems that this method of storing the text causes.) Of the above systems, the two batch-oriented formatters, the Xerox ViewPoint system, ded, and Rav Ctav store the text in the order it was input.

To the authors' knowledge there exists no fully bi-directional, terminal-independent, full screen editor available for use on the same systems that support the formatters available to the authors.

Therefore, the authors embarked on a project to develop vi.iv, a bi-directional version of the now standard vi full-screen editor found on every UNIX system that these authors know about and on many others as well. The design and implementation was done by the first author for his M.Sc. thesis under the second author's supervision. During the design and development, the second author played customer to the programmer played by the first author. Both authors tested vi.iv thoroughly in the normal course of their Hebrew and English writing. In addition, both this paper and a companion Hebrew version, the first author's M.Sc. thesis[17], were edited with vi.iv. Both of these papers were typeset by use of ditroff/ffortid with ditroff macro packages designed for their required styles. The implementation was carried out on the version of vi distributed in source form with University of California Berkeley's UNIX 4.3bsd.

The plan for the rest of the paper is to discuss the goals for the construction of vi.iv, the basic concepts underlying bi-directional editing, and the design and structure of vi and its implications on the design and structure of vi.iv. Then the paper turns to the actual construction of vi.iv and the various semantic problems encountered. The paper concludes with an evaluation of the effort and recommendations for future work. There is a user's manual[18] available giving a detailed functional description.

All examples of file contents, screen appearances, and entered commands or text are shown in constant-width fonts. File contents are always shown from left to right and have a file-beginning symbol, '⊣', at the left end. Screen appearances are marked by vertical bars on the left and right sides serving as boundaries of an imaginary screen whose width is 30 characters. An L-R screen view is left justified on the page and an R-L screen view is right justified in the page. Entered text or commands are always shown from left to right with no special markings at either end; each character shown is part of the input. In a file contents or an input, ‘ˆ’ represents pressing CONTROL simultaneously with the following character.

GOALS

There were a number of goals adopted early on with the closely related aims of making the bi-directional vi.iv easy to use by one familiar with the uni-directional vi and easy to program given the code for vi. These goals were to produce a bi-directional vi.iv

1. which is indistinguishable from the uni-directional vi when one is working with a uni-directional file, especially for an entirely left-to-right (e.g., English only) file, and possibly even for an entirely right-to-left (e.g., Hebrew only) file,
2. for which the extension into bi-directional processing is as orthogonal as possible, possibly using a single new command that causes all existing vi commands to be applied in the opposite direction,
3. which is language independent in that it works unchanged with any reasonable terminal for any right-to-left language, such that the Latin alphabet is available as an option, and
4. which can be built as a slight modification to an existing implementation of vi, with changes isolated into as few modules as possible in order to allow easy porting to other implementation of vi.

The last goal is critical. vi is a large program, which has been ported to a large number of different CPUs and operating systems. The code of vi on all of these systems is not the same. Attempting to port vi.iv to all of these systems will be a nightmare *unless* it can be done by porting the modifications that we have made. If we cannot port the modifications, we would have to replay the porting history of vi itself as we adapt the full vi.iv to all the systems to which vi has been ported. If our modifications are too extensive or the modules that we modified do not exist relatively unchanged in the other versions of vi, we will be forced into the nightmare.

Consistent with the goal of language independence, in the following discussions about the treatment of mixed directional text, left-to-right text shall be considered as written in the language LR using the LR alphabet, and right-to-left text shall be considered as written in the language RL using the RL alphabet. In contrast, the terms ‘L-R’ and ‘R-L’ are occasionally used as adjectives meaning ‘left-to-right’ and ‘right-to-left’, respectively.

TIME ORDER

It is clear that RL should be printed or displayed from right to left and LR should be printed from left to right. It is also clear that a human being would prefer to enter all text in what is called *time order* (also called *logical order*), i.e., each letter of any language in the text goes into the machine in the order that it would be pronounced if the text were mumbled out loud as it were entered. Of course, the text is not printed in this order. It is printed in what is called *visual* or *printing order*, which is a function of the *current document direction*. For example, the current document direction is L-R if one is to read the document in a general L-R flow. In an L-R document, such as this paper, paragraphs are indented from the left, the beginning of a line is the leftmost character of the line, and lines end to the right of the beginning. Moreover, the main language in an L-R document is, in all likelihood, one that is written from left to right, e.g., LR. The Hebrew version of this paper would be an R-L document. Perhaps the term ‘document’ in this context is misleading, because the concept of L-R or R-L document may actually apply to parts of what is normally considered one document. For example, a book in Arabic about Shakespeare’s works might contain large quoted English language passages that would best be displayed as L-R (sub)documents interspersed among the explanatory R-L (sub)-documents written in Arabic.

For the purpose of this discussion, a *streak* is defined as a maximal length string of text within a single line all of whose characters are in languages of the same direction. So in the line (`He said "`*`shalom to you`*`" to Uri.`),

```
He said "שלום לך" to Uri.
```

the three streaks are

```
He said "
שלום לך
" to Uri.
```

The basic invariant governing the reading of bi-directional text in visual order is that one should not move on to the next line until all the text on a given line has been read. One bounces around within a line in order to read off the streaks in an order agreeing with the current document direction and to read each streak its own direction. So, in the example above, if the document direction is L-R, one first reads

```
He said "
```

from left to right, then

```
שלום לך
```

from right to left, and finally

```
" to Uri.
```

from left to right. Note that the order in which the characters are read is precisely the time ordering of the line,

```
He said "ךל םולש" to Uri.
```

That is, a reader cognizant of the conventions concerning visual order reconstructs time order in reading the text. It is the job of the displaying and printing software to construct the visual ordering of a file from its time ordering.

This process of converting text from visual to time order is called *laying the text out*, or more simply, *layout*[19]. The following line-by-line layout algorithm assumes a device that prints from left to right:

```
for each line in the file do
        if the current document direction is L-R then
            reverse each contiguous sequence of RL characters in the line
        else (the current document direction is R-L)
            reverse the whole line about;
            reverse each contiguous sequence of LR characters in the line
        fi
od
```

The algorithm must or can be varied if the device prints from right to left or in both directions. If there exist a line whose length is longer than the physical line length, then the time-ordered line is folded into pieces that fit the physical line length first, and then the pieces are subjected to layout as if each were a line itself; the pieces are interpreted in the same document direction as the original line.

The issue facing any designer of bi-directional text-processing system is *when* layout is performed. There appear to be two schools of thought on this issue: Layout is done

1. as the text is entered, or
2. as the text is printed.

The former is the scheme followed by most commercial mass-market Hebrew word processing software, such as MacInHebrew, Multi-Lingual Scribe, WORDMILL, etc. and the vih editor. The latter is followed by Becker's multi-lingual WYSIWYG Xerox ViewPoint system, by Buchman and Berry's ditroff/ffortid, which in turn is based on Gonczarowski's hbtroff[20], by MacKay and Knuth's bi-directional $\TeX$/XET, by Raz and Gordon's ded, which was supervised by Gonczarowski.

The implication of layout-while-entering is that the file is stored as you see it and of layout-while-printing is that the file is stored in time or input order.

The drawback with layout-while-entering and storing the file as it appears is that the appearance is a function of the line length. If you change the line length, e.g., to show the message on device different from the entry device, it is difficult and in some cases impossible due to punctuational ambiguities (if the device does not have two sets of punctuation symbols, one for each language or the editor does not remember in which language a punctuation symbol was entered) to reconstruct the original input form in order to calculate the new appearance. Thus, layout-while-printing and storing the files in the input order is more general.

The drawback with layout-while-printing is the time spent to lay the file out each time the file is printed. However, given current hardware costs, workstation CPU utilization, CPU speed vs. printer speed, etc., this is not a burdensome cost; the user cannot even detect the extra filter.

Thus, layout-while-printing is more general and is not too much more expensive.

It is interesting that the first three groups that opted for layout-while-printing did so independently of each other; i.e., they each had written at least a draft of their work before any had published any of their work.

It turns out that if layout-while-printing is adopted system-wide, then all eighth-bit-clean[2] programs that input and output lines can be converted into their bi-directional versions simply by changing the basic operating-system supplied output device drivers to perform the layout algorithm on all lines that go by them[21]. If one ignores the problem of changing the user messages into RL, the application software does not have to be touched! Thus, the technique is very powerful. Note that this line-by-line technique does not work if the application does not write lines. An example of an application that does not write lines is a full screen editor such as vi or any application using curses[22] as its screen-writing interface. Hence the the first author had work to do.

INTERNAL CODES

In order for the algorithm of the previous section, or any variant of it adapted to a full-screen editor, to work, the editor must be able to determine

1. for each character in which language, LR or RL, it is, and
2. the current document direction.

Using the proper binary code for each letter solves the first problem. Specifically, each of the Latin, Arabic, Farsi, Hebrew, Urdu, etc. alphabets has few enough characters to represent with 7-bit codes. In fact, at least Latin, Arabic, Farsi, and Hebrew have standard 7-bit codes. Assuming that the terminal can deal with only one right-to-left language plus Latin, two 7-bit alphabets are needed. An 8-bit code suffices, with the eighth bit distinguishing the language or direction and the remaining seven bits being the standard code. The emerging standard in Israel at the moment is ESCII in which Latin has the eighth bit off and Hebrew has the eighth bit on. All of the newer terminals produce ESCII code from the keyboard and display the proper Latin or Hebrew character when sent ESCII code. What is particularly nice about this code is that each character that is in both alphabets, e.g., digits and punctuation symbols, appears twice, once in the eighth-bit-equals-zero half and once in the other half, and their codes differ by only the eighth bit. Thus, it is easy for the editor to remember the language in which the characters were entered. This eighth-bit scheme appears also to be the standard in at least those parts of the Arab world that subscribe to the ISO-8859/6 or the ASMO-708 standard.

The use of data codes in which the eighth bit is significant occasionally causes problems with software written under the assumption of a seven-bit ASCII code. Such programs may use the eighth bit to store program-relevant data, especially two-state information about each character. For example, in vi, it was discovered that in the screen image, the eighth bit is used to mark which blanks were inserted into the screen image to implement tab expansion and are not actually in the file. Such programs have to be modified so as to leave the eighth bit of the data alone, i.e., to be *eighth-bit clean*[23, 21].

The eighth-bit method of distinguishing alphabets is satisfactory only when two languages with small alphabets are involved. A fully general solution requires 16-bit characters[8], or more[24], leaving ample room in each character for both a language code and the character code itself. However, for now, the realities of existing hardware dictates the eighth-bit solution.

[2] The software must be eighth-bit-clean (leave the eighth bit of data untouched) in order not to mangle the data which are in the RL language, which generally has codes with the eighth bit on.

STRUCTURE OF vi

Figure 1 shows the basic abstract structure of vi. The program vi can be considered to

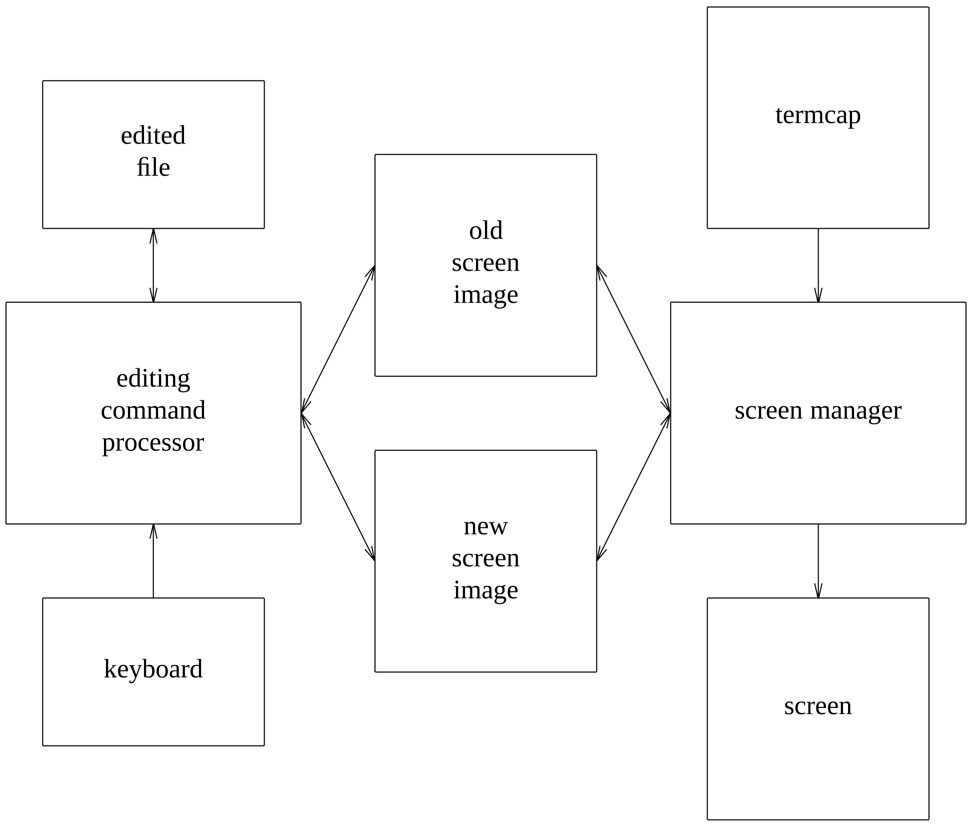

Figure 1. Structure of vi

have two main parts, the editing command processor and the screen manager. These two parts communicate via two data structures showing screen images before and after the last editing action. The command processor takes commands from the keyboard, interprets them through the old screen image (as the user is giving the commands on the basis of what he or she sees on the screen), updates the edited file, and makes a new screen image. The screen manager computes the difference between the two screen images, and using the description of the current terminal in the termcap file, determines a sequence of characters to send to the terminal which causes the terminal to update its screen to match the new screen image. These characters are sent to the terminal, and the new screen image is moved to replace the old. At the very worst, the screen manager has to send to the terminal all the characters of the new screen image in order to have the whole screen image rebuilt. However, if the terminal has useful editing operations such as to slide lines up and down or characters to the left and right and these operations are listed in the termcap (terminal capabilities database) entry for the particular terminal in use, then the screen manager can send much less to the terminal. In the best case, the manager sends over a few characters that cause what is already there to be moved to new positions and then only the new character or characters that were, say, inserted. The performance of vi with a particular terminal can be improved considerably by careful writing of its termcap entry.

IDEAL STRUCTURE FOR vi.iv

vi.iv was to be built by modifying the code of vi in as few places as possible. Given that all characters are to be stored in time order, the editing command processor of vi should not have to be changed, apart from code necessary to implement new operations. For example, since the file contents are stored in time order and search patterns come in also in time order, there should be no change necessary to the routines that search the file for text matching patterns. It is, nevertheless, necessary to modify the screen manager to at least do some variation of the layout algorithm. However, the screen manager can take advantage of the extant before and after images constructed by the largely unchanged command processor. Specifically, each time any character on any line is changed, the screen manager must at the very worst rewrite that whole line, applying to the line the portion of the algorithm that is applied to each line. Fortunately, the screen manager can usually do better than redraw the whole line. With the proper terminal capabilities, it should be able to issue commands to slide the existing text of the line around, delete some of its characters, and insert new ones in specific places. Thus most of the difficulty in writing vi.iv should be in optimizing the screen update based on termcap information.

In the current production of vi.iv, it was decided to not worry too much about the problems of optimizing the screen updating. These could be attacked as an independent problem, say building on the work of Myers and Miller[25]. We felt that it was more important, in this first attempt, to get the semantics right. Moreover, one should not optimize until one has the software running long enough to have positively identified the code that should be optimized. As we run vi.iv, it seems that given current line, computer, terminal, and human typing speeds, the existing screen managing algorithm does an acceptable job. It may well end up that very little should be optimized. In the end, a few special tricks were put in to handle extremely common R-L document direction situations better than they are by the existing algorithm, which after all, was designed for unidirectional L-R documents. For example, in an R-L document, the cursor moves and the text grows to the right. Most existing terminals are designed to have the cursor move and the text grow only in one direction, L-R. Unless special commands exist to add text to the left of a stationary block of text, simulating the R-L growth of text can cause redrawing of a whole line after adding each character. Note that the *whole* line must be redrawn with one less blank before, the new character, and then the stationary block of text out to the right margin of the terminal screen. Fortunately, many Hebrew terminals have an append-to-the-left operator, which can be noted in their termcaps and used by the screen manager.

Of course, it was necessary to add to the command processor of vi code to deal with new commands, such as switching document direction and switching input language.

IMPLEMENTATION OF vi.iv

This section describes the steps taken to write vi.iv from the code of vi and the problems encountered. The description of the implementation and semantic problems is largely in the order that they were encountered. While it would be nice to say that the semantic issues were resolved *before* implementation began, the fact is that we did not see some of the semantic problems until in the thick of implementation.

Decomposition of vi

The first step in implementing vi.iv was to understand the implementation of vi. To this end, the detailed structure of vi was studied carefully so as to be able to identify what modules would have to be changed for each operation and for the new screen manager. It was found that the communication lines connecting the modules of vi look like spaghetti due to the high orthogonality of vi commands. For example, almost all commands need starting and ending positions to indicate the scope of the effect of the command, and any means to identify a position can be used to provide either position. Thus, nearly every routine uses nearly the same set of global variables, and each routine liberally calls other routines to do portions of its job. Despite the spaghetti, all the pieces were found and understood.

Making vi eighth-bit-clean

It had been discovered quite early that vi was taking advantage of the now invalid assumption of 7-bit characters to use the eighth bit in each character to help keep track of blanks that are due to tab expansion, and thus that are not really in the file. We needed the eighth bit cleared out to allow input of and storing of 8-bit characters. Fortunately, the characters whose otherwise zero eighth bit were used were limited to those that were on the screen. A full-screen sized table was created to keep this information separate from the characters themselves. A new eighth-bit-clean version of vi was created in which the eighth bit of the characters in the edited file were left alone. While the first author continued working, the second author tested the new vi for compatibility with the original. In several months of use, he found no difference.

Step by step construction

Producing a new eighth-bit-clean vi identical in behavior to the original suggested a step-by-step, almost prototyping method for producing vi.iv. Specifically, at each step, a new feature was added to produce a new program which was then subjected to normal daily use as a vi as well as to thorough manual testing of the new feature. The order in which the features were added is

1. ability to switch to an R-L view (of only Latin text) with R-L cursor movement
2. ability to switch to the RL language

Building vi.iv step-by-step and using it as vi after each step allowed the authors to keep their sanity by limiting the source of any incompatibilities that were found.

vi **bugs that had become features**

There was a lot of temptation to fix a number of clear bugs in vi, especially when the solution was only a few lines of code. However, the authors resisted making these changes in order not to compromise downward compatibility to vi. The point is that these bugs are so well-known by the fingers of a whole world of vi users that they are really features. The fingers of these users, the authors included, would spot the differences instantly.

Design issues

As predicted, most of the work was in getting the screen redrawing based on termcap information right. However, there were a number of other issues, most notably in the name of and in the semantics of new commands and in the semantics of existing commands when confronted with mixed directional text.

Language direction control

We originally thought to have a letter v which, as in vih, could precede each existing vi command to get its RL equivalent. Thus, a is the start of an LR append, and va is the start of an RL append. However, during implementation, we determined that it would be more orthogonal and more economical in typing if the language in which the operations work were instantly and globally settable. Thus, `^x` (Control-x) was designated to cause switching to the other language, whatever it may be. Then a means to start an append in the current language.

vi uses almost every character in sight as a command character. This leaves very few, in fact too few, unassigned characters to use as new commands. In some cases, while a chosen character had no meaning to vi *per se*, it does have a meaning to the tty driver. An example of this is `^x`. Some people use `^x` as the line kill character in place of the more common `^u`. So, the character for all such new commands is settable by vi.iv's `:set` command.

Bi-directional semantics

Deciding the bi-directional semantics of familiar vi commands at first proved a source of fierce discussions among the authors. However, in each problematic case, the final resolution fell out of a return to the basic principles:

1. the file is stored in time order,
2. the file is displayed in visual order,
3. the layout algorithm is applied by the screen manager after each change, either to the contents of the file or to the view, and
4. the screen manager in principle redraws the whole screen after each such change, but can often avoid doing that, usually redrawing only the lines that contain at least one changed or new character, and sometimes, even less than that.

Insertion

The first semantic problem was that of inserting a character of the language that is written in the current view direction into a block of text of the opposite direction. As an example, suppose the current view direction is R-L, and the cursor, denoted by an underscore, is positioned in the middle of a line containing only L-R English text. What appears at the far right of the screen is:

```
                             |            He said "" to Dan.|
```

and what is in the time-ordered file is:

```
⊣He said "" to Dan.
```

Now, an R-L append command is issued to begin inserting the word ‘שלום’ (*shalom*). After the insertion of the ‘ש’, the file contains (Label A, for future reference):

```
⊣He said "ש" to Dan.
```

and, perhaps surprisingly, the screen shows (Label B):

```
                            |           " to Dan.שHe said "|
```

The reader whose logic is complaining both about the position of the cursor and the appearance of the screen should hang on; he or she has fallen into the same mental trap into which we fell. Recall that the reading direction is from right to left, and one reads the individual LR streaks within the R-L line from right to left.

At first, we forgot about the rules for R-L reading and the first author implemented at some difficulty an algorithm that kept what appeared to be more continuous across time. This algorithm displayed (Label C):

```
                            |           He said "ש" to Dan.|
```

after input of ‘ש’. When we realized that this continuity is wrong (It is right for an L-R view!), the problem was how to implement the sudden exchange of the two halves of the LR text around the RL insertion. Interestingly, we realized that this continuity is wrong when the ditroff/ffortid bi-directional formatter used to format this paper printed the line A as B rather than C in an earlier draft’s attempted explanation of why C is correct!

After agonizing over the efficiency of doing the exchange and even doubting the correctness of the basic semantic principles mentioned in the section on ‘Bi-directional semantics’, we decided to stick to the basic semantic principles, and to fall back on the basic line-drawing algorithm. That is, the character is inserted into the file and then the screen manager takes over discovering that the line has to be redrawn and redraws it correctly with no additional action on the part of the insertion code. In this respect, the cursor is treated as an overstruck character that ends up in the same screen position as does the character to which it points in the time-ordered file.

These principles imply that an RL append means the same as vi’s append with respect to the time-ordered file (there is *no* change to that portion of the code), but that what is going into the file is RL text with the eighth bit on. The append itself is not R-L, rather what is appended is RL! This understanding makes it clear how the cursor should move during an insert (more on cursor movement later).

In the last analysis, we reverted to the surprising but correct versions and learned to accept line B as the correct display of line A under an R-L view.

Pattern matching and substitution

A pattern, entered in time order, is matched against the contents of the file, stored in time order. Of course, both the file contents, if it is in the screen, and the pattern, in the bottom, command line of the screen, are shown in visual order. The command line always has an L-R view. This behavior, while simple to explain, easy to implement, and easy to apply, can produce surprising results. Suppose a line of the file is (For the reader not familiar with Hebrew, the Hebrew letters are simply the consecutive first letters of the alphabet.) :

```
–|אבגabcdefghiדהו
```

Were it on the screen, its L-R and R-L view appearances would be:

```
|גבאabcdefghiוהד                    |
```

and

```
                                |                    והדabcdefghiגבא|
```

The time-ordered pattern

```
    /אבגabc/
```

which appears on the screen as

```
    /גבאabc/
```

matches the first six characters of the time-odered line, and the cursor ends up positioned at the first character of the line, א. In both the L-R and R-L views, the matched characters are not adjacent on the screen (Label D).

```
|גבאabcdefghiוהד                    |
 ---         ---
```

and

```
                                |                    והדabcdefghiגבא|
                                                     ---         ---
```

The matched characters are marked with underscores, and the double underscored character has the cursor as well.

The real impact of this behavior shows up when one does a substitution. The substitution follows the vi rules working on the pattern, the substitution, and the file contents *all in time order*. Under the same view direction and file contents as above, the time-ordered substitution

```
        :s/אבגabc/זחטjkl/
```

which appears as

```
        :s/גבאabc/טחזjkl/
```

causes the following file contents:

```
–|זחטjkldefghiדהו
```

The resulting screen appearances are (Label E):

```
|טחזjkldefghiוהד                |
```

and:

```
                              |                והדjkldefghiטחז|
```

The underscored characters of lines D were changed in making lines E. On the other hand, the time-ordered substitution

```
        :s/אבגabc/jklזחט/
```

which appears as

```
        :s/גבאabc/jklטחז/
```

causes the following file contents:

```
–|jklזחטdefghiדהו
```

which gives rise to the following screen appearances:

```
|jklטחזdefghiוהד                |
```

and:

```
                              |                והדdefghiטחזjkl|
```

While the pattern matching and substitution according to the time-ordered characters in the file may be easy to describe and, in fact, easy to implement, it may seem counter-intuitive especially when the matched and changed portions are not adjacent to each other on the screen. However, to pattern match and thus substitute according to what appears on the screen is fraught with a number of problems.

1. Most of the time that one searches for a pattern it is to find something that is *not* currently visible on the screen. In that case, searching according to appearance is inappropriate and searching according to time-ordered characters is really the only way that works. For example, suppose that in a previous editing session the time-ordered line (`*He said "*Hello there Uri and Dan. How are you today?*" to Uri and Daniel*`) (Label F)

```
⊣|הוא אמר "Hello there Uri and Dan. How are you today?" לאורי ודניאל
```

   were entered into the file. Under a R-L view direction, assuming a screen width sufficiently large, it happens to appear as

```
ו    לאורי ודניאל "Hello there Uri and Dan. How are you today?" והוא אמר ו
```

   on the screen, with ‘`today?`’ adjacent to ‘`" אמר`’ (`*said "*`). Now suppose that in this current session, it is desired to bring this sentence into the window. Is not the user more likely to remember that he or she typed something beginning with (`*He said "*Hello`)

```
        הוא אמר "Hello
```

   and thus the pattern should be typed

```
        /הוא אמר "Hello/
```

   than that after it was typed it looked as

```
ו    לאורי ודניאל "Hello there Uri and Dan. How are you today?" והוא אמר ו
```

   and thus the pattern should be typed something like (`*He said "*today?`) (Label G)

```
        /הוא אמר "today?/  ?
```

2. Even if a user were to remember a previous appearance enough to build a visually matching pattern, there is no guarantee that the pattern is still valid. The visual appearance depends on the location of line breaks. If an intervening edit, say by someone else, has inserted additional line breaks, then a different set of adjacencies would occur, invalidating previously valid visual patterns. For example, suppose that a line break were inserted in line F after the first period of the quoted English text. The R-L view would be

```
|                      Hello there Uri and Dan." אמר והוא|
|                      לאורי ודניאל "How are you today?|
```

Then ‘`Dan`’ would be adjacent to ‘" אמר’ (*`said "`*), and the visual pattern G would not work any more.

3. The search algorithm would have to construct the screen appearance of each line in the file before it could search in the line. This, of course, is inefficient and, moreover, requires significant changes to the basic functional part of vi. Every change to the functional part of vi increases the danger that downward compatibility from vi.iv to vi will be lost.
4. Searching by appearance contradicts the time-ordered orientation of the entire rest of vi.iv. Adopting searching by appearance would just make the command language of vi.iv inconsistent and thus more complicated.

Thus searching according to time order is adopted. Indeed, in all of the commands, some of the effects of adherence to time order seem a bit surprising. This is especially so for the deletion and change commands. However, in each case, a similar argument for adopting pure time-order semantics can be given.

Automatic wrap-around

In vi it is possible to set the `wrapmargin` variable to a number greater than its default of zero. The effect of setting `wrapmargin` to $n > 0$ is that whenever the typing of a word crosses into the last n columns of the terminal screen, that last word is moved to the next line and the space or tab before it is erased. Here a word consists of characters other than space and tab surrounded on both sides by space, tab, or newline. With this feature, in principle, the user should not ever have to type newlines, except for those that are semantically required, and still the text will be broken into lines all of which are shorter than the screen width.

The general rule is that the layout algorithm is applied to each line independently, even if the lines are created by the editor splitting longer lines. Whatever is done must follow this simple rule, at each step along the way. The automatic wrap-around is problematic if one should happen to cross the wrap-around margin while typing a word in a language whose direction is opposing that of the current view. Consider the LR language append of ‘`one two three four`’ to a line beginning with ‘הוא ספר’ under an R-L view. Suppose the line length and `wrapmargin` settings are such that during typing of the ‘`three`’, the wrapping margin is crossed. Thus, at the time the last character of ‘`three`’ is typed, the appearance of the screen is

```
                                   | one two three ספר הוא|
```

corresponding to the time-ordered input

```
    רפס אוה one two three
```

Note that as ‘`one two three`’ has been entered, the cursor has remained stationary and the text has grown to the left on the screen. Thus, to the user, it appears that the ‘one’ has crossed the wrapping margin rather than the ‘three’.

Now, as the space is typed, the wrap-around is to occur. If the implementor were not careful, the ‘one’ would get moved to the next line, leaving the ‘two three’ where they are for the screen appearance:

```
|      two three והוא ספר|
|                     one|
```

Then, entering ‘four’ would yield the screen:

```
|      two three והוא ספר|
|                one four|
```

This screen corresponds to the time-ordered file contents of

```
⊣רפס אוה two three
⊣one four
```

and is therefore clearly wrong. The file should contain:

```
⊣רפס אוה one two
⊣three four
```

and the screen should show:

```
|        one two והוא ספר|
|              three four|
```

The difficulty lay in the fact that in vi, the wrap-around is done entirely by the screen manager. The screen manager notices that the text has gone over the margin, finds the last word in the line, moves it to the next line in the screen image and then informs the editing command processor of the change to the file contents. This procedure is more efficient than the more general protocol implied by the structure of vi. In this protocol, the screen manager would notice that the text has gone over the margin and inform the command processor of this fact, the command processor would then make the change in the file and construct a new screen image, and finally the screen manager would display the new image in the normal manner. The two methods are equivalent when time order and visual order are the same. When they are not, the the more efficient, within-the-screen-manager procedure causes the anomalous behavior describe above. Handling wrap-around correctly in the presence of bi-directional text requires that either

1. the screen manager be able to undo the layout of what is in the screen image in order to find the last word entered in time and to move it, or
2. the more general full protocol be used.

The former requires a major new modification of the screen manager and addition of an un-layout procedure. The latter requires earlier reporting of the change to the command processor and that is all. The already written layout procedure takes care of the rest. Since the latter alternative is cleaner, it was adopted.

Bi-lingual characters

Another problem concerned the characters that exist in both languages. These are the digits, the punctuation symbols, and the space and tab characters. These characters exist in both the lower and upper halves of the full eight-bit character set. The two codes for any such character have exactly the same lower order seven bits, and thus differ by exactly 128. When they are typed, they will go into the file with the eighth bit setting consistent with the current language direction.

For each of these characters, the two codes are printed the same way on the screen, but they cause different behavior under the layout algorithm. The user that does not remember or does not know under which language one of these characters is entered may be surprised by the screen appearance after modifications and may not be able to predict the effects of modifications. For example, exchanging the order of inputting a space and changing language direction causes different screen appearances. Suppose the view direction is R-L. Suppose one has entered in time order an LR troff command with an RL argument. The time-ordered file contents are (`.su` *`Abstract`*):

```
⊣|.su ריצקת
```

If the space between the ‘`u`’ and the ‘ת’ is in LR, then the screen appearance of the line is:

```
                          |                        תקציר.su |
```

If, however, the space is in RL, then the screen appearance is:

```
                          |                        תקציר .su|
```

As another example, again in an R-L view, the time-ordered file contents (*`the company`* `Jones & Sons`)

```
⊣|הרבחה Jones & Sons
```

appears as

```
                          |               Jones & Sons החברה|
```

if the ‘&’ and the spaces surrounding it are in LR, but appears as

```
                          |               Sons & Jones החברה|
```

if the ‘&’ and the spaces are in RL.

In order to help the user to avoid getting lost in these circumstances, we decided to have an option under which vi.iv would highlight the characters of the language whose direction opposes that of the current view. This highlighting can be requested either from the invoking command line by adding the –MS option or from within vi.iv by uttering one of:

```
:set marksecondlang
:set marksl
```

This highlighting is done by what ever method of highlighting is offered by the current terminal as indicated by the current termcap. With an underscoring highlighting the four R-L view examples above would appear, respectively, as:

```
|                         תקציר.su_ |

|                        תקציר  .su|

|                 Jones & Sons החברה|

|                 Sons & Jones החברה|
```

This feature should help the user keep track of numerals. If one wants a numeral to appear in the normal order, it must be entered either in L-R order as LR characters or in R-L order as RL characters. The presence or lack of underscoring should remind the user which way the numerals were entered.

These bi-lingual characters cause a problem to users when constructing search patterns. It is impossible, merely by looking at a printed copy of text or by listening to someone or one's mind pronounce some text, to know for sure under which language one of these characters was entered into the edited file. Therefore, it was decided to let any of these characters match itself in either language. This decision was necessary to allow people, especially those who did not enter the file, to find specific text. Once the text is found, the highlighting of the secondary language can be used to determine in which language all the characters of the text were entered. Note that *only* pattern matching is treated this loosely; the user must properly identify the language of each inserted or replacing character. This treatment of bi-lingual characters in pattern matching is not unlike ignoring case distinctions in patterns containing Latin letters.

Because the codes of the two languages' versions of one of these bi-lingual characters differ only in the eighth bit, it is easy to implement this modification to the search algorithm. Apart from the eighth bit cleaning, this change was the only one made to the editing command processor not directly for the purpose of implementing a new feature.

Cursor movement

A particularly vexing question was what to do when the user is using arrow keys or their respective letter or control-letter equivalents and the cursor crosses a language change boundary. Should the cursor move smoothly across the screen following the visual order? Should the cursor follow the time ordering of characters even if it means that the cursor jumps to next boundary in the same direction and begins to move in the opposite direction? The insistence on adherence to time order notwithstanding, the first choice was chosen. This choice allows the arrow keys to mean the same direction as is written on the key. It seemed clear that to change the natural correspondence of the arrow keys to direction would be too counter-intuitive to the user. Two additional commands, v and V, were added for moving forward and backward in time order to allow the user to follow time order should it be necessary. These commands are useful for dispelling confusion about the order of contents of the file. One simply moves the cursor to the beginning of the line, according to the current view direction, types v, and watches where the cursor goes.

In insert mode, the cursor is always at the screen position that will receive the next character. Where this is relative to the previously entered character depends on the current language and the current view direction. After ESCAPEing from the insert mode, the cursor will move to be at the character after the last inserted character relative to the current view direction.

Retrospection on semantics

In retrospect, falling back on the basic principles and letting the screen manager take care of all seemingly violent changes to what is displayed proved to be simpler over-all to both the user and the implementor. Even though, at first sight, the discontinuities appear wrong, a brief consideration about what has transpired before the user's eye convinces the user that what has happened is consistent with the basic principles and is therefore right. The user gets used to these discontinuities and begins to expect them. Moreover, the surprising situations are highly unlikely to ever occur except in examples concocted for thorough testing of the software. Consider the example of the section on 'Insertion'. Normally, one is not typing a complete English sentence containing a single Hebrew word in the midst of a Hebrew document, with an R-L view. Such a sentence is far more likely to occur in an English document with an L-R view. In the L-R view, as mentioned above, as the 'ש' goes in, the parts of the sentence are *not* exchanged, and moreover the subsequent 'ל' goes in to the left of the 'ש'.

For the implementor, doing it right turns out to to be the easiest to implement. What could be simpler than using unchanged code in the editing command processor and letting the screen manager, which was already written to do the layout, take care of making all screen changes. The existing command processor code takes care of putting all new characters into the file in time order because for the uni-directional vi, time order and visual order are identical. Having realized earlier that doing it right is better for both the user and the implementor could have saved us much time.

TESTING AND USER EXPERIENCE

The authors recruited volunteers from the Technion community to use vi.iv both as vi and as vi.iv. In addition, with the help of a filter to convert from a time-ordered file to a screen-ordered file, some tested vi.iv as a replacement for vih. These patient and hardy volunteers were invaluable in finding bugs and allowing the authors to locate and fix them.

The vi.iv program has been in continuous use at the Faculty of Computer Science at the Technion since the Fall of 1988. The main users are graduate students, who are under a requirement to write their Master's and Doctor's theses in Hebrew, and faculty members, who are under a *publish or perish* requirement to produce papers in English for conferences and journals (hence, the Hebrew and English versions of this paper!). The nature of the work done in the faculty dictates that all of these documents contain lots of mathematical and program text. Both groups have helped to locate bugs. The groups' different natures allows finding of bugs all kinds.

Those of the student group that have talked to us express satisfaction that vi.iv is a truly *bi*-directional version of vi unlike the heretofore used vih, which, as mentioned, is really, essentially only a uni-directional Hebrew version of vi. One particularly enthusiastic student user said that he did not know he managed to survive before he started to use vi.iv.

Surprisingly, we heard no complaints about our choices of semantics for insertion, pattern matching, substitution, and wrap-around. Given the degree to which this group complains about other software problems, this lack of complaint is significant. Probably the students did not complain because they have grown used to bi-directional work through all their years of schooling in Israel. Also the really surprising cases simply do not occur in practice. All of the anomalous examples involved R-L views in which English language sentences contain embedded Hebrew language phrases. This configuration occurs only in examples concocted by the authors to help build a robust vi.iv. In real life, one has R-L views in which all sentences are in the Hebrew language and there are occasional English language phrases; this configuration does not yield any surprises with which the students were not already familiar.

The faculty group reports no significant observable difference from vi itself, the only differences being the perennial reminder of being in `LR INSERT MODE` and an occasional accidental digression to `RL INSERT MODE` as the ^x key is struck accidentally. (These reminders could be considered a major difference from the normally terse vi. However, the reminder is on even when not in insert mode; that is, it says in which language the next insertion will be, whenever that insertion will be.)

PORTING

vi.iv was implemented on 4.3bsd UNIX on a VAX™, because that was the only UNIX system and machine for which the sources were available at the Technion. A set of differences between the code of vi and vi.iv for this system and machine has been produced. It remains to be seen whether these differences are applicable to other versions of vi. As soon as the Technion obtains the license to and a copy of Sun Microsystem's vi source code, an attempt will be made to port these differences to the Sun™ vi to produce a Sun vi.iv. It is planned to use the user's ability to select fonts in vfont format[26] as screen fonts to allow use of bitmapped Arabic and Hebrew fonts with this vi.iv.

Three possibilities exist for this porting effort.

1. It will be easiest to apply the difference between the VAX vi and the VAX vi.iv to the Sun vi to obtain the Sun vi.iv.
2. It will be easiest to apply the difference between the VAX vi and the Sun vi to the VAX vi.iv to obtain the Sun vi.iv.
3. Neither of the above will be feasible and either the VAX vi.iv or the Sun vi will have to be modified from scratch to produce the Sun vi.iv.

If either of the first two options work, the portability goal will have been met.

While waiting to get the source license from Sun Microsystems, the VAXes began to disappear and the pressure for a Sun version grew. Reluctantly, the system gurus at the Technion decided to port the VAX vi.iv to both Sun3 and Sun4 systems to run on top of vthtool, the virtual Hebrew terminal created to allow vih to run in SunView™ windows.

There were two stages. The first was to get vi.iv running on the Suns with the ordinary terminals. The second was to get it to run with vthtool virtual terminals. As it turned out, stage 2 was easy compared to stage 1. However, what a disaster! So far, after about six months, it is still as buggy as can be. The system programmers, Yael Dubinsky and Haim Roman, are going through the headache that the programmers of Sun Microsystems probably suffered in when they first ported vi to their system. Most of the problems solved so far have been traced to differences between the VAX and the Sun CPUs and C compilers, especially for the Sun4. This effort is the nightmare envisioned in the section on ‘GOALS’.

After his porting experience, Roman believes that method 1 of porting would be the easiest. He offered the following reasons for believing so.

1. The hardest and least fun part of porting is trying to account for differences in the compilers and CPUs.
2. If we start with a program, vi, that *works* in the desired environment, then if there are problems, we know that they are caused by the vi.iv additions.
3. There is a hope that the Sun vi code is cleaner and better documented and commented than the VAX vi code.
4. There are people at the Technion, namely the authors, who understand the differences between vi and vi.iv quite well. People who understand the differences between the VAX vi and the Sun vi are not easily accessible to the system programmers at the Technion.

CONCLUSIONS

The experience of producing and using vi.iv has reaffirmed a number of principles discovered by others.

1. All text files should be kept in time or logical order so that applications can make semantic decisions on the basis of characters in the order that they are said.
2. Only upon printing or displaying should a text file be subjected to layout in which the text of each language is printed in its own natural direction.
3. The language or at least the direction of each character should be at least logically stored with each character. It is not terribly important whether this be done by actually storing a code in each character or by having escapes in the file whose effect lasts until the next escape.
4. Application software must not break when the language codes or escapes are encountered.
5. The current document or view direction is a function of the application and should not be stored in the file itself.

G. Allon, J. Becker, C. Buchman, Z. Becker, J. Gonczarowski, D. Knuth, and P. MacKay all report some or all of these observations in the literature cited in this paper. These observations apply equally for formatters, editors, and operating systems.

Perhaps the most interesting conclusion is that the original decision by Bill Joy to divide the vi editor into a editing command processor part and a screen manager part as shown in Figure 1 was a good one. It permitted relatively easy extension to obtain vi.iv. Therefore, Joy's decomposition of vi meets Parnas's[27] criteria for decomposing systems into modules, in that all changes were well isolated into few easily identified modules. For example, all of the code concerning layout is inside the screen manager, and only the screen manager is affected by the addition of the layout algorithm. All of the changes to the command processor part necessitated by each new feature that we added to obtain vi.iv is confined to one routine or to a group of routines accessing the same data structures (an abstract data type).

Indeed, any program, so decomposed, can be made bi-directional with essentially the same technique that we applied to make vi.iv from vi. All that is required is that the program

1. assume that all input files are in time order with the language of each character determinable from the character or other information in the file and
2. have a semantic part which is well isolated from the part that is dealing with printing or displaying output.

The modifications are mostly in the part dealing with output; the semantics part is modified only if the application needs new bi-directional features. The bi-directional formatters mentioned in the introduction have this property; other editors, e.g., emacs, have this property; and operating systems, such as UNIX[28] and MINIX[29] have this property[21]. Because this paper is about one bi-directional *editor*, the topic of editors deserves a more thorough discussion here.

The emacs[7, 30] editor is apparently also divided into an editing command processor part and a screen manager working from termcap information. It should be straightforward to develop a bi-directional emacs.scame with the full power of emacs. There might actually be two approaches to building emacs.scame. One is to modify the screen manager at the source code level as we did for vi.iv. The other possibility might be to use emacs's extensibility to change the behavior of the screen manager at the user level *without* touching the source code.

While we are not familiar with the innards of any of the various mouse-based editors that are available on most windowing systems, it is not hard to imagine constructing such an editor decomposed the way vi is. Certainly, the use of a mouse as a locator does not prevent the desired decomposition. Moreover, there is nothing in the layout algorithm that is affected by the use of a mouse as a locator. Thus, it should be easy to build bi-directional versions of these mouse-based editors using the technique described in this paper.

Moreover, since the curses[22] package of terminal-independent, screen writing primitives using termcap information was apparently derived from the screen manager of vi, it should be straightforward to develop a curses.sesruc which could be used to easily construct a bi-directional version of any application built on top of curses.

It should also be possible to develop a tri-directional version of vi (What will it be named?) that is able to write text from left to right, from right to left, and from top to bottom. It could be used to edit files with, say, Latin, Mongol, and Chinese text. The hard problem is deciding the meaning of scrolling top-to-bottom text, especially when the top-to-bottom text might be mixed with horizontal text. Should hitting the return key cause the text in the screen to move sideways? Should scrolling always be in the same direction or dependent on the direction of the text at the point of the cursor? Whatever semantics of scrolling is chosen, it should be straightforward to implement the tri-directional version of vi using a variation of the tri-directional layout algorithm of tri-roff[31].

These issues remain for future work.

ACKNOWLEDGEMENTS

The authors thank all the volunteer users of vi.iv and the people who participated in some semantics preference surveys. They include Gil Allon, Aythan Avior, Zeev Becker, Danit Ben-Ari, Giora Benedek, Orna Berry, Shulamit Cohen Schwartz, Shimon Even, Yoel Ferrara, Lior Gendel, Rachel Habusha, Micha Hofri, Amos Israeli, Alon Itai, Marc Jalfon, Eli Kantorowitz, Eli Kurim, Yoëlle Maarek, John Marberg, Moshe Molcho, Yuval Ne'eman, Bluma Roth, Adrian Segall, Zeev Shtadler, Gil Sideman, Ilan Spillinger, Ady Tal, Jacob Ukelson, Yifat Wisberg, and Yaron Wolfstahl.

The authors thank Joseph Becker and the anonymous referees for their comments, which helped to improve the paper.

Yael Dubinsky and Haim Roman had the thankless task of porting the VAX version of vi.iv to the Sun machines.

An earlier version of this paper was accepted for presentation at the 1989 International Conference on Computer Processing of Chinese and Oriental Languages sponsored by the Chinese Language Computer Society.

REFERENCES

1. Z. Wu, W. Islam, J. Jin, S. Janbolatov, and J. Song, 'A Multi-Language Characters Operating System on IBM PC/XT Microcomputer', in *Proceedings of Second International Conference on Computers and Applications*, Beijing, PRC, pp. 579–585, (June, 1987).
2. C. Buchman, D.M. Berry, and J. Gonczarowski, '*DITROFF/FFORTID*, An Adaptation of the UNIX *DITROFF* for Formatting Bi-Directional Text', *ACM Transactions on Office Information Systems*, **3** (4), 380–397 (1985).
3. D.E. Knuth and P. MacKay, 'Mixing Right-to-left Texts with Left-to-right Texts', *TUGboat*, **8** (1), 14–25 (1987).
4. B.W. Kernighan, 'A Typesetter-independent TROFF', *Computing Science Technical Report No. 97*, Bell Laboratories (March, 1982).
5. D.E. Knuth, *The TEXbook*, Addison-Wesley, Reading, MA, 1984.
6. W. Joy, 'An Introduction to Display Editing', *Electrical Engineering Computer Science*, University of California, Berkeley, CA 94720.
7. J. Gosling, *UNIX Emacs User's Manual*, Carnegie-Mellon University, 1982.
8. J.D. Becker, 'Multilingual Word Processing', *Scientific American*, **251** (1), 96–107 (1984).
9. I. David, *Vih Manual Page*, Technion, Haifa, Israel, .

10. R. Gordon and A. Herman, *Ded Manual Page,* Hebrew University, Jerusalem, Israel, .
11. *Alef-Bet Manual*.
12. *Einstein Manual*.
13. J. Weinstein, *MacInHebrew,* MIT Hillel House, Cambridge, MA, 1986.
14. *Multi-Lingual Scribe,* Gamma Productions, Santa Monica, CA, 1984.
15. *Rav Ctav Manual*.
16. *WORDMILL User's Guide,* Intersoft Software Engineering, Ltd., Jerusalem, Israel, 1984.
17. U. Habusha, vi.iv*, A Bi-Directional Version of the* vi *Full-Screen Editor (in Hebrew),* M.Sc. Thesis, Technion, Haifa, Israel, 1989.
18. U. Habusha and D.M. Berry, vi.iv *User's Manual,* Faculty of Computer Science, Technion, Haifa, Israel, 1989.
19. J.D. Becker, 'Arabic Word Processing', *Communications of the ACM*, **30** (7), 600–611 (1987).
20. J. Gonczarowski, *HNROFF/HTROFF Hebrew Formatters based on NROFF/TROFF,* Computer Science Department, Hebrew University, Jerusalem, Israel, 1980.
21. G. Allon, *Towards a Bi-Directional Operating System,* M.Sc. Thesis, Technion, Haifa, Israel, 1989.
22. K. Arnold, *Screen Updating and Cursor Movement Optimization: A Library Package,* Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720, .
23. *System V Interface Definition,* Volume I, Issue 2, AT&T Customer Information Center, Indianapolis, IN, 1986.
24. N.H.F. Beebe, 'Character Set Encoding', *TUGboat*, **11** (2), 171–175 (1990).
25. E.W. Myers and W. Miller, 'Row Replacement Algorithms for Screen Editors', *ACM Transactions on Programming Languages and Systems*, **11** (1), 33–56 (1989).
26. vfont *Manual Page,* Berkeley UNIX Manual, Section 5, University of California, Berkeley, Berkeley, CA, 1986.
27. D.L. Parnas, 'On the Criteria to be Used in Decomposing Systems into Modules', *Communications of the ACM*, **15** (2), 1053–1058 (1972).
28. D.M. Ritchie and K.L. Thompson, 'The UNIX Time-Sharing System', *Communications of ACM*, **17** (7), (1974).
29. A.S. Tanenbaum, *Operating Systems: Design and Implementation,* Prentice-Hall, Englewood-Cliffs, NJ, 1987.
30. R.M. Stallman, *GNU Emacs Manual,* Fourth Edition, Emacs Version 17, Free Software Foundation, Cambridge, MA, 1986.
31. Z. Becker and D.M. Berry, 'An Adaptation of the UNIX ditroff for Formatting Tri-Directional Text', *Chinese Computing Seminar '88*, Singapore (1988).

APPENDIX: SCREEN PHOTOGRAPHS AND DUMPS

For the benefit of the skeptic, this appendix shows photographs of various terminal screens during a run of the VAX vi.iv and the dump of a SunView screen during a run of the ported Sun3 vi.iv. All of these runs are on the same file, a file containing first some Hebrew text and then some English text. The file, called `im.ain` contains in time order (Label H):

```
אם אין אני לי, מי לי?
אם אני לעצמי, מה אני?
אם לא עכשיו, אימתי?
-פרקי אבות
If I am not for myself, then who will be?
If I am only for myself, then what am I?
If not now, then when?
-Pirke Avot
```

If the Hebrew characters are printed in time order as the lower case Latin letters encoded by their lower order seven bits, then the file appears to contain (Label I):

```
‘m ‘io ‘pi li, ni li?
‘m ‘pi lrvni, nd ‘pi?
‘m l‘ rkyie, ‘inzi?
-txwi ‘aez
If I am not for myself, then who will be?
If I am only for myself, then what am I?
If not now, then when?
-Pirke Avot
```

The L-R view of this file should be (Label J):

```
אם אין אני לי, מי לי?
אם אני לעצמי, מה אני?
אם לא עכשיו, אימתי?
-פרקי אבות
If I am not for myself, then who will be?
If I am only for myself, then what am I?
If not now, then when?
-Pirke Avot
```

The R-L view of this file should be (Label K):

```
                            אם אין אני לי, מי לי?
                            אם אני לעצמי, מה אני?
                              אם לא עכשיו, אימתי?
                                       -פרקי אבות
         If I am not for myself, then who will be?
          If I am only for myself, then what am I?
                            If not now, then when?
                                       -Pirke Avot
```

The photographs of terminal screens in Figure 2 show

1. the file sent through cat, which shows the time ordering and strips off the eighth bit of the Hebrew text (This looks like listing I.),
2. the file viewed under an L-R view in vi.iv (This looks like listing J.), and
3. the file viewed under an R-L view in vi.iv (This looks like listing K.).

The screendump in Figure 3 shows three windows, clockwise from the bottom right,

1. the file sent through cat in an ordinary SunView window,
2. the file viewed under an L-R view in vi.iv in a virtual Hebrew terminal window

(1)

(2) (3)

Figure 2. Terminal screen photographs

(This looks like listing J.), and

3. the file viewed under an R-L view in vi.iv in a virtual Hebrew terminal window (This looks like listing K.).

The run of the file through cat shows no text at all for the Hebrew lines, not even the Latin characters obtained by stripping off the eighth bit! The Sun code is already eighth-bit clean and the standard SunView window interface has assigned no glyphs to the character codes whose eighth bits are on.

Figure 3. Sun screen dump