1984

TED - a screen-based text editor

Wendy Bryant  
*University of Wollongong*

Roderick Holmes  
*University of Wollongong*

Raymond Overmass  
*University of Wollongong*

Christopher Tisseverasinghe  
*University of Wollongong*

Recommended Citation

Bryant, Wendy; Holmes, Roderick; Overmass, Raymond; and Tisseverasinghe, Christopher, TED - a screen-based text editor, Department of Computing Science, University of Wollongong, Working Paper 84-3, 1984, 43p.  
http://ro.uow.edu.au/compsciwp/51
TED - a Screen-based Text Editor

by

Wendy Bryant
Roderick Holmes
Raymond Overmass
Christopher Tisseverasinghe

Preprint No. 84.3 March 7, 1984

Submitted in partial fulfillment of the requirements of CSCI 321 Software Project at the University of Wollongong in 1983

P.O.Box 1144, WOLLONGONG, N.S.W. 2500, Australia
Tel: (042) 270 859
Telex: 29022
PART I

TECHNICAL REPORT
Technical Report for TED – a Screen-based Text editor

Wendy Bryant
Roderick Holmes
Raymond Overmass
Christopher Tisseverasinghe

TEAM D
CSCI321
Software Project
University of Wollongong

ABSTRACT

This document contains the technical report for TED, an interactive screen-based editor for use on the UNIX system. The data structures, algorithms and mechanisms used in the implementation of TED are described.
1. INTRODUCTION

The design of this interactive editor may be considered in two spaces, namely the user interface and the computer interface. The user interface defines user commands, entered at the keyboard, with which the user is able to control the editor.

In the editor both spaces have the same three dimensions. These are:

- TEXT
- SCREEN
- KEYBOARD

Each dimension is essentially orthogonal to other components, communicating loosely with other dimensions by messages or via a few global data structures.

The text dimension contains mechanisms concerned with the current line, frames and the frame line, the temporary file, the free list and the page frame. The screen dimension contains mechanisms to maintain screen, window and cursor control. The main program resides in the keyboard dimension, thus providing a mapping between user commands from user-space to the sequence of actions in computer-space. The mechanisms within this dimension are concerned mainly with input control. The editor also contains global mechanisms such as the debug trace and error recovery.

This report is based upon the three dimensions in the editor. The mechanisms within each dimension, as well as global mechanisms, are described in relation to user-space and computer-space. Most of the technical report is taken up with the computer interface since the user interface deals mainly with user commands and these are described in the Users' Guide and Tutorial for TED.[1]
EDITOR
  
TEXT
  PAGE FRAMES
  PAGE MAP
  PAGE FILE
  CURRENT LINE
  TEXT CONTROL
SCREEN
  SCREEN CONTROL
KEYBOARD
  CURSOR CONTROL
  INPUT CONTROL
GLOBAL MECHANISMS
  MODES OF OPERATION

STRUCTURE DIAGRAM
2. TEXT

Text control deals with the storage of the temporary file being edited, with its retrieval and modification, and with the replacement of the text.

The general structure is the same as the demand paging system used in a virtual memory machine. The ideas upon which this structure is based are as follows:

- the editor edits a copy of the original text file. The original text file may be empty.
- the copy of the original text file is stripped of all control characters.
- the copy of the original text is divided into blocks of a known size so that no line is divided between blocks.
- each block obtains a unique block number which is also its location on a temporary page file on secondary storage.
- a map is kept according to the first line number of each block in order to preserve the sequential order of the file.
- page frames are kept in main memory. These structures are used to retrieve and manipulate text kept on the temporary page file. Each page frame carries information detailing which page is currently in the frame, a flag to indicate if the text has been altered, a reference parameter to be used by the replacement algorithm and a pointer to the next free character in the page frame's buffer.
- all insertions and deletions take place on the current line which is a separate data structure outside of the frames so that shifting of text is minimised.
2.1. Data Structures

1. The Page Frame
   - pointer to a page in the page chain
   - dirty flag (i.e. changed or not)
   - reference parameter
   - pointer to next free character in buffer
   - text buffer

2. A Page Chain Entry
   - next entry in page chain
   - first line occurring in this entry
   - block on page file associated with this entry

3. Current Line

   Each line is considered to be a character interpretation of the line length followed by the text in that line. Each page frame buffer and each overflow buffer in the page file consists of a sequence of these lines.

4. The Page File
   This is a file consisting of the overflow buffers used by the page frames.
2.2. **Page Frames**

These structures have a number of primitive functions which work on them. These are:

- **ClearPFrame(f)** –> initialise the frame
- **Modified(f)** –> set the dirty flag to true
- **Referenced(f)** –> updates the reference parameter
- **AllocPFrame(f)** –> finds the oldest frame, writes the buffer if changed, clears the frame and returns a pointer to it
- **RoomFor(f,n)** –> determines if the frame ‘f’ has enough room for ‘n’ more characters in its buffer

2.2.1. **Algorithms**

**ClearPFrame(frame)**

1. Set the pointer to a page entry to NULL
2. Set reference parameter to its lowest value
3. Set dirty flag to false
4. Set character pointer to the beginning of the buffer

**Modified(frame)**

1. Set dirty flag to true
2. Update reference parameter to a new higher reference value

**Referenced(frame)**

1. Update reference parameter to a new higher reference value

**AllocPFrame()**

1. Set a pointer to the first frame
2. While not all frames processed do
   2.1 If frame has lowest reference parameter
      2.1.1 Update a pointer to the oldest frame
3. If the oldest frame was changed then
   3.1 Write its buffer to the page file (WritePage)
4. Return a pointer to the oldest frame

**RoomFor(frame,n)**

1. If room for ‘n’ characters between next free character pointer and end of buffer then
   1.1 Return true
2. Else return false
2.3. Page Chain And Free List

The following functions are used with the page chain and free list:

- **Morepages()** - requests more page entries to be added to the free list.
- **AllocPEntry(p,n)** - returns a pointer to a page entry (which has been inserted into the page chain such that the previous page entry is pointed to by 'p') from the free list with first line number 'n'.
- **FindPage(n)** - returns a pointer to the page entry which contains the line number 'n'.
- **FreePage(p)** - add page entry 'p' to free list.

2.3.1. Algorithms

**MorePages**

1. Request a number of page entries from operating system
2. For all the new page entries do
   2.1 Set up a unique block number associated with this entry
   2.2 Add this entry to the free list

**AllocPEntry(p,n)**

1. If free list empty then
   1.1 Get more page entries (MorePages)
2. Get the first entry off the free list
3. Reset the free list pointer to point to the remaining free page entries
4. If previous page entry 'p' is NULL then
   4.1 Set up the first map entry and a pointer to it
5. Else insert new page entry into page chain
6. Return new page entry

**FindPage(n)**

1. While more page entries in page chain do
   1.1 If 'n' < first line number of next page entry then
      1.1.1 Return a pointer to this page entry

**FindPage(p)**

1. Add page entry 'p' to free list
2. Update free list pointer
2.4. The Page File

The page file has only two primitives but it is also used in conjunction with the page frame text buffer. Functions associated with the page file are:

- **InitPage()** — creates and opens the temporary page file
- **UnlinkPage()** — releases the temporary page file and storage back to operating system
- **Getpage(f)** — reads the block of data found from the calculated offset (stored in the page entry pointed to by the f-page pointer in frame 'f') and stores it in the buffer in the frame pointed to by 'f'
- **WritePage(f)** — writes the buffer in the frame pointed to by 'f' to the page file at a calculated offset

2.4.1. Algorithms

- **InitPage()**
  1. Get a unique name for file from operating system
  2. Open file and mark file as empty

- **UnlinkPage()**
  1. Remove the temporary file associated with this edit session

- **WritePage(f)**
  1. Fill empty part of text buffer with a non-printable character
  2. Determine which overflow buffer in the page file is to be replaced by calculating an offset (block number in the page entry pointed to by frame 'f' multiplied by the block size)
  3. Move disk head to this page
  4. Write the buffer in frame 'f' into this overflow buffer

- **ReadPage(f)**
  1. Determine which overflow buffer in page file to read by calculating an offset (as in WritePage)
  2. Move disk head to this buffer
  3. Read the buffer into the page frame text buffer
  4. Examine the page frame text buffer and set up the next free character pointer
2.5. Current Line

GetLine(n) → Gets the text of the line ‘n’ into the current line data structure
PutLine(n) → Replaces the text of line ‘n’ in the buffer with the contents of the current line

2.5.1. Algorithms

GetLine(n)
1. If the current line has been changed then
   1.1 (PutLine) back into the page frame buffer
2. Find the line number ‘n’ in the page frame buffers
3. Copy the line from page frame buffer into the current line
4. Set current line variables appropriately

PutLine(n)
1. Find line ‘n’ in the page frame buffers (FindLine)
2. If not enough room for line ‘n’ then
   2.1 Split page frame buffer into two new buffers
   2.2 Set up frame that old line ‘n’ now resides in
3. Move characters in buffer so there is enough room for new line ‘n’
4. Move the current line over old line ‘n’
5. Update page frame variables appropriately
2.6. Compound Functions

There are a number of compound functions which use some or all of the data structure types. These are:

- **GetPage(p)** → returns a page frame with its buffer full from page file
- **FindPFrame** → returns frame with the page entries buffer
- **FindLine(n)** → returns pointer to line ‘n’ and its frame
- **SplitFrame(p,n)** → splits page frame into two buffers at character pointed to by ‘p’, the first line being ‘n’
- **InsertLine(n)** → inserts the contents of ‘CurrentLine’ after the line
- **LoadText** → gets all lines from the file to be edited and copies them into the page frame buffers and overflow buffer
- **LoadLine(fd)** → gets the next line from file ‘fd’ into current line
- **SaveLine(fd)** → writes a line into the file fd
- **SaveText** → writes all the lines from the page frame buffers and overflow buffers into the text file
- **InsertNewLine()** → split the current line, write the current line back and insert the new line

2.6.1. Algorithms

**GetPage(p)**
1. Allocate a page frame
2. Set the page frame’s map entry pointer to ‘p’
3. Read page from page file into page frame’s buffer
4. Return page frame

**FindPFrame(p)**
1. While not all page frames searched do
   1.1 If frame is page frame associated with page entry ‘p’ then return it
2. (GetPage) and return the frame

**FindLine(n)**
1. While not all page entries searched and line not in this page map entry do
   1.1 Get next page entry
2. If page not in page frame then
   2.1 Put page into a page frame
3. Set page frame to the FoundFrame
4. Until required line reached do
   4.1 Change line pointer to next line in sequence of lines
5. Set other variables
**SplitPageFrame(p, n)**
1. Allocate a page frame
2. Allocate a page entry with firstline number 'n' and set pointer in page frame
3. Copy characters from character position 'p' to the end of the buffer into a new buffer
4. Set up pointers to next character in both buffers
5. Set both page frames as modified
6. Return the new page frame

**InsertLine(n)**
1. Find line number 'n'
2. If no room for current line in buffer then
   2.1 Split page frame buffer
   2.2 If not room in old buffer then
      2.2.1 Set up variables to new buffer and its start
      2.2.2 Move characters so current line can be inserted
3. Move current line into buffer
4. Update next free character pointer
5. Update page entries after the buffer page entry by one
6. Update last line by one

**LoadText**
1. Check and open the text file
2. If ok then
   2.1 While more line to get do
      2.1.1 (LoadLine) into ‘CurrentLine’
      2.1.2 (InsertLine) into page frame buffer
3. Else write appropriate message
4. Close text file

**LoadLine(fd)**
1. Set up variables
2. While next character in 'fd' not newline do
   2.1 If TAB then add appropriate number of blanks to 'CurrentLine'
   2.2 Else if RETURN then set length to zero
   2.3 Else if BACKSPACE then decrement length by one
   2.4 Else if printable character then add character to 'CurrentLine'
3. Set up length of line to 'CurrentLine'
4. Return 'CurrentLine'
SaveLine(td)
1. Determine length of line
2. Determine number of blanks at beginning of line and write number of TABs and blank characters to file
3. Write remaining part of line to file
4. Write a newline character to indicate end-of-line

SaveText
1. Check and open the text file
2. If writable then
   2.1 While not all lines processed do
      2.1.1 Set up pointer to next line
      2.1.2 Write next line (SaveLine)
3. Else write message
4. Close text file

InsertNewLine0
1. Get the current line
2. Determine how far it is indented
3. Split the current line at the cursor position
4. Write all characters left of the cursor to the current line
5. Add/remove blanks to/from the characters to the right of the cursor so that it has the same number of blanks as the first part of the line
6. (InsertLine) after the current line into the text buffer
2.7. Utility Functions

There will be a number of utility functions including a set of functions for buffered input and output of files. The utility functions include:

- `inC` - maintains the buffer and passes back next character
- `outC` - maintains the buffer with new character
- `flushC` - outputs buffer to file when full
- `Copystr(src,d,n)` - copies `n` characters from a source(src) to a destination (d)
- `Getchar` - returns a character from standard input
3. SCREEN

The screen dimension contains three main mechanisms. These are text control, screen control and cursor control. We will look at each of these mechanisms in turn.

3.1. Overview

The text window is 22 lines long by 71 characters wide. The cursor can be moved around the text window while in overtype mode by using the UP, DOWN, LEFT and RIGHT keys. The user cannot move the cursor past the end of the file ('End of File' is written on the line following the last line in the file) except by using RETURN in overtype mode to create a new line to add text.

The first 8 columns of the screen contain the line's number. There are two additional lines on the screen which are not part of the text window. These are the information line and the command line and they are situated above the text window. The line numbers and the information line are displayed in inverse video for clarity. The user is never able to move the cursor into the line numbers or the information line and the command line can only be accessed when in command or insert modes.

The Information line is the top line and it contains the name of the file being edited, the current mode of operation and the column numbers which are currently in the text window.

| FILE: | Last line: | Mode: | cols: |

When the mode is changed, the variable Printmode is changed and the function WriteMode is called.

WriteMode() -> changes the value of MODE on the information line

When the cursor is moved outside the range of column numbers specified on the information line the column range is updated by the function WriteColumns.

WriteColumns -> changes the column range on the information line

The function WriteHeader writes the updated information on the information line.

The command line is the line under the information line. This line has two uses - the entering of alphanumeric commands in command mode and the insertion of text in insert mode.

Command:
This is an example of what the screen looks like in insert mode.
3.2. Text Control

AdjustScreen → Moves text window to encompass cursor position
WindowUp → Moves text window 8 lines up
WindowDown → Moves text window 8 lines down
WindowLeft → Moves text window 40 columns left
WindowRight → Moves text window 40 columns right
PutNum(n, size) → Writes line numbers on screen

The following variables are used by the text window routines:
DisplayFirst.line : first line in the text window
DisplayFirst.col : first column in the first line of text window
DisplayLast.line : last line in the text window
DisplayLast.col : last column in the last line of text window

3.2.1. Algorithms

AdjustScreen
1. If CursorText is outside the window boundaries
   1.1 Move window accordingly
2. If text has changed
   2.1 Adjust contents of screen
3. Move CursorScreen to where CursorText is placed.

WindowUp
1. Move window 8 lines up
2. Set Changes to TEXT_AND_LINES

WindowDown
1. Move window 8 lines down
2. Set Changes to TEXT_AND_LINES

WindowLeft
1. Move window 40 columns left
2. Set Changes to TEXT_ONLY

WindowRight
1. Move window 40 columns right
2. Set Changes to TEXT_ONLY

PutNum(n, size)
1. Write a number ‘n’ of length ‘size’ with padded zeroes
3.3. Screen Control
AdjustContents \rightarrow makes appropriate changes to the screen contents
RefreshText(l,d) \rightarrow writes text on screen
RefreshTextLines \rightarrow writes text and corresponding line numbers on screen
SeeLine(n) \rightarrow sees if line number 'n' exists
PutChar \rightarrow writes character from current line

The following variables are used in the screen control routines:

Changes : holds information on what type of changes for screen
CursorScreen.line : line number relative to screen
CursorScreen.col : column number relative to screen

3.3.1. Algorithms
AdjustContents
1. If Changes = ONE_CHAR
   1.1 Replace character
2. Else if Changes = ONE_LINE
   2.1 Find current line
   2.2 Move to appropriate column
   2.3 Write the line on to the screen
3. Else if Changes = INSERT_CHARS
   3.1 Find current line
   3.2 Write the line on to the screen
4. Else if Changes = TEXT_BOTTOM
   4.1 Refresh Text
5. Else if Changes = TEXT_ONLY
   5.1 Write the new column numbers
   5.2 Refresh Text
6. Else if Changes = TEXT_AND_LINES
   6.1 Refresh Text and Lines
7. Else if Changes = WHOLE_SCREEN
   7.1 Clear the screen
   7.2 Write the Header
   7.3 Refresh Text and Lines
8. Set Changes = NO_CHANGES
RefreshText(first,last)
1. For line = first to last do
   1.1 If line > last line number then
       1.1.1 Write new line number and the text
       1.2 Else write text only

RefreshTextAndLines(first,last)
1. For line = first to last do
   1.1 Write line number and text

SeeLine(n)
1. If n > Lastline
   1.1 Return (false)
2. If n is equal to the current line number
   2.1 Set pointer to the first character of the current line
   2.2 Get the length of the current line
3. Else
   3.1 Find line number ‘n’
   3.2 Set pointer to the first character of the line found
   3.3 Get the length of the line found
4. Return (true)

PutChar
1. Write a character from current line
3.4. Cursor Control
The cursor is moved around the screen by using the cursor addressing mechanism available on Volker Craig terminals.

MoveToCommand  -> moves cursor to command line
Move(line, col)  -> moves cursor to specified line and column
PutStr         -> writes a text string on a line
WriteHeader    -> writes information line
WriteLast()    -> writes last line number on information line
WriteMode()    -> writes mode on information line
WriteColn()    -> writes column range on information line
ClearMessage() -> clears command line

The following variables are used by cursor control routines:
CursorText.line : line number
CursorText.col  : column number relative to the text

3.4.1. Algorithms
MoveToCommand
1. Show user where the CursorScreen was placed
2. Move Cursor to command line

PutStr
1. Skip all leading blanks in the text line.
2. If the whole line was blanks
   2.1 Return
3. If there were enough blanks in the text line (ie.>3) to warrant moving the cursor
   3.1 Move Cursor to the first non blank character
4. Else
   4.1 Set Cursor to the beginning of the line
5. If DisplayLast.col > current text line's length
   5.1 Set end of line to current line length - 1
6. Else
   6.1 Set end of line to DisplayLast.col
7. Skip blanks from end of line until the first non blank character
8. Write the string
9. Set CursorScreen.col to end of line.
Move(line,col)
1. Move Cursor to specified line and column
2. Set CursorScreen to line and column.

WriteHeader
1. Write file name being edited
2. Write which mode the user is in
3. Write column numbers

WriteLast()
1. Move to line 0.column 25
2. Print "Last Line"
3. Write last line number

WriteMode()
1. Move to line 0.column 47
2. Print "Mode"
3. Write current mode from PrintMode

WriteColn()
1. Move to line 0.column 65
2. Print "col:"
3. Write Displayfirst.col
4. If DisplayLast.col > MAXLENGTH then
   4.1 Write MAXLENGTH
5. Write DisplayLast.col + 1

ClearMessage()
1. Move to command line
2. Clear to end of line
4. KEYBOARD

In user-space the *keyboard* dimension is concerned with user commands which control the editor. In computer-space it is concerned with receiving and interpreting input, then executing appropriate procedures according to the input.

4.1. Input Control

Keyboard inputs drive the edit process. The terminal is set to raw mode and all input is interpreted by the editor. The program sets up parity bits by masking every input character. Input for the program is entered via printable characters typed at the keyboard in overtype mode, or via the HOME key and BREAK key for command and insert modes respectively.

The input for command and insert modes is typed on the command line. It is interpreted by a few basic functions. The input is read character by character using the function `getchar`. Printable characters only are accepted and input is terminated by the RETURN key.

For insertions, the line entered is inserted into the text when the RETURN key is pressed.

After a command has been issued from the command line, the entered line is broken into the command character (see Commands) and its corresponding arguments. The command character is then used to call the appropriate command. The arguments are dependent on the command and so are validated by the command. If the command character is invalid an error message is produced on the command line (see Error Handling).

The main functions used to interpret input are:

- `ReadCmd` -> accepts user input on command line for insert and command modes
- `CmdArgs` -> breaks the input buffer into separate arguments

4.1.1. Algorithms

`ReadCmd`

1. While valid command character entered do
   1.1 If special command key (BS, TAB, etc)
      1.1.1 Adjust command buffer accordingly
      1.1.2 Write edited line command
   1.2 Else if printable character
      1.2.1 Store in command buffer
      1.2.2 Write on command line
2. Add end of character string character
3. Erase command line
CmdArgs

1. If no more characters in command buffer
   1.1 If command does not use previous arguments
       1.1.1 Initialise argument to spaces
   2. Else
      2.1 Initialise arguments to spaces
      2.2 For each argument do
         2.2.1 While next cmd buffer character not space and not command do
            2.2.1.1 Store character in argument
         2.2.2 Ignore all intervening spaces
4.2. Overtype Mode

The simplest form of editing is to overtype characters displayed on the screen. The user types the character that is desired over the character currently in the file (like a typewriter), or in the case of an empty line the user types over the (seemingly) blank spaces. This is how overtype mode is used for text entry.

Overtyping involves the simple substitution of the character entered for the character in the current cursor position. A copy of the current line is changed and the actual change occurs in the copy of the text file when the cursor is moved and the text buffers changed.

The RETURN key works differently to printable character keys because when RETURN is typed it does not overtype the character at the current cursor position. The RETURN key inserts an Invisible newline marker immediately to the left of the cursor position. Automatic Indentation is provided in TED. This means that whenever RETURN is pressed, the cursor is placed directly underneath the first character on the previous line. Any lines of text below the cursor are moved downwards to make room for the new line.

If the cursor is in the middle of a line when the RETURN key is typed the line will split immediately before the cursor position. The character at the cursor position and everything to the right of it will be moved down to the next line on the screen, again with automatic indentation.

\[ \text{DoPrint}(c) \rightarrow \text{replaces character at current cursor position with the character 'c'} \]

**Deletion**

When in overtype mode it is possible to delete text without entering command mode. This is done by using control keys. Deletions of a character, a word and to the end of the line are possible using the specified keys (RUBOUT,CTRL-w,CLEAR respectively).

\[ \text{DeleteChar} \rightarrow \text{deletes character at current cursor position} \]
\[ \text{DeleteWord} \rightarrow \text{deletes words of alphanumeric characters to the right of the cursor position} \]
\[ \text{DeleteToEoln} \rightarrow \text{deletes from cursor position to the end of line} \]

4.2.1. Algorithms

\[ \text{DoPrint}(c) \]
1. If cursor beyond last character in line
   1.1 Extend line with spaces up to the cursor position
2. Replace character
3. Move cursor to next character on line
4. Set flag that current line has been changed
DeleteChar
1. Copy the line from position after cursor position to end of line, to the cursor position
2. Decrement length of current line
3. Set flag to re-show line

DeleteWord
1. Initialise 'end-of-word' to cursor position
2. While next character in line is a space and 'end-of-word' < line length
   2.1 Increment 'end-of-word'
3. While next character is alphanumeric
   3.1 Increment 'end-of-word'
4. DeleteChar from cursor position to 'end-of-word'
5. Set flag to re-show line

DeleteToEoln
1. Set length of line to current cursor position
2. Set flag to re-show line
4.3. Insert Mode

Insert mode is entered by pressing the BREAK key. This moves the cursor to the start of the command line. The cursor position in the text is marked in inverse video so that the user can clearly see where the text will be inserted. The text to be inserted is typed on the command line and when the RETURN key is pressed, the text on the command line is inserted in between the marked cursor position and the character immediately to the left of the cursor.

This method of insertion entails copying the characters following the marked cursor position on a line, further along the line to make room for the additional text. In this way, the need for text on a line to be moved each time a letter is inserted is eliminated.

- **Insert**  → driver to insert text
- **InsertChars**  → inserts text from the cursor position

4.3.1. Algorithms

**Insert**

1. Move to command line, mark previous cursor position
2. Read command
3. Move to marked cursor position
4. Insert the text (InsertChars)

**InsertChars**

1. Extend line to cursor position if past last character in line
2. Copy characters to position further along line (CopyStr)
3. Copy inserted characters into position (CopyStr)
4. Move cursor to after inserted characters
5. Set flag that current line has been changed
4.4. Command Mode

To enter command mode the HOME key is pressed. This marks the cursor position in inverse video then moves the cursor to the beginning of the command line. The command is typed on the command line and is executed when the RETURN key is pressed. If the command is valid, it is executed and the editor returns to overtype mode with the cursor at the marked position or at its new position as determined by the command. If an invalid command is entered a message appears on the command line in place of the invalid command and the editor returns to overtype mode.

All changes that are made to text in the file are displayed in the text window. When global substitution commands are executed the last change made is displayed.

Commands are available to:

- Go to a specified line number
- Find a string
- Make single substitutions
- Make global substitutions
- Delete line(s)
- Copy line(s)
- Move line(s)
- Save parts of the file under any filename
- Read in another UNIX file
- Move forward a specified no. of text windows
- Move backwards a specified no. of text windows
- Move right a specified no. of text windows
- Move left a specified no. of text windows

Command -> driver to execute commands
GoToLine(n) -> puts cursor in desired position
DoFind -> searches for string
DoChange -> makes text substitution
DoGlobalChange -> uses DoChange to make global substitutions
DoDelete -> deletes line n if it exists
DoCopy -> duplicates specified lines
DoSave -> updates file with text in work area
DoRead -> appends a specified file after a given line number
DoMove -> moves specified lines
DoUpScreen(n) -> moves text window ‘n’ text windows backwards in the text
DoDownScreen(n) -> moves text window ‘n’ text windows forward
DoRightScreen(n) -> moves text window ‘n’ text window widths right
DoLeftScreen(n) -> moves text window ‘n’ text window widths left
4.4.1. Algorithms

Command
1. Move to command line
2. Read command
3. Break up arguments
4. If command valid then
   4.1 Call appropriate command
5. Else
   5.1 Write error message

GoToLine(n)
1. Convert argument to numeric
2. If argument non-numeric then
   2.1 Write error message
3. If 'n' > last line number then
   3.1 Write error message
4. Else if n > 0 then
   4.1 CursorText.line is set to 'n'
5. Else
   5.1 Write error message
6. Set column number

DoFind
1. If no argument entered then
   1.1 Write error message
2. Else
   2.1 Search (using R.G.Dromey's Quicksearch [2]) from current position until
      found or complete cycle
   2.2 If not found then
      2.2.1 Write message

DoChange
1. If not 2 arguments entered then
   1.1 Write error message
2. Else
   2.1 Perform search from current position for first argument
   2.2 If found then
      2.2.1 Copy 2nd argument for 1st
   2.3 Else
      2.3.1 Write error message
DoGlobalChange
1. While string1 found do
   1.1 DoChange

DoDelete
1. Convert arguments to numeric
2. (GetLine) to be deleted
3. Copy remaining lines in buffer over the present line
4. Update frame pointer
5. If frame empty
   5.1 Remove page entry from page map
   5.2 Release page to page pool
6. Update page map to indicate removal of line

DoCopy
1. Convert arguments to numeric values
2. (GetLine) to be copied
3. Insert line obtained from GetLine after the specified line number

DoSave
1. If filename specified open and create it
2. for lines n1 to n2 do
   2.1 (GetLine)
   2.2 Use (SaveLine) to save the line in the file
3. Close file

DoRead
1. If file exists
   1.1 Open file
   1.2 While more lines in file
      1.2.1 Load in file name
      1.2.2 Insert line after the specified line number (or the current line)
   1.3 Close line

DoMove
1. (DoCopy)
2. For each line copied do
   2.1 (DoDelete)
**DoUpScreen(n)**
1. Calculate how many lines to move back using argument ‘n’
2. Subtract the calculated number of lines from DisplayFirst.line and set cursor to line in the middle of the screen
3. If the line number where cursor will be < 1 then
   3.1 Set line number where cursor will be to 1
4. (GoToLine) where cursor will be

**DoDownScreen(n)**
1. Calculate how many lines to move forward using argument ‘n’
2. Add the calculated number of lines to DisplayLast.line and set cursor to line in the middle of the screen
3. If the line number where cursor will be > last line number then
   3.1 Set line number where cursor will be to last line number
4. (GoToLine) where cursor will be

**DoRightScreen(n)**
1. Calculate how many columns to move right using argument ‘n’
2. Add the calculated number of columns to CursorText.col
3. If the column number where cursor will be > MAXLENGTH then
   3.1 Set CursorText.col to MAXLENGTH

**DoLeftScreen(n)**
1. Calculate how many columns to move left using argument ‘n’
2. Subtract the calculated number of columns from CursorText.col
3. If the column number where cursor will be < 1 then
   3.1 Set CursorText.col to 1
5. GLOBAL MECHANISMS

5.1. Trace Mechanism

The program trace is global. It consists of a procedure called 'Trace' as well as a statement at the beginning of every procedure which has the form:

\[
\text{if (TraceOn) Trace("procedure name", level)}
\]

where 'TraceOn' is a global variable that is set to zero for no trace and set to one when a trace is desired. The procedure 'Trace' writes out to a trace file the procedure name passed to it as an argument. 'Level' is suitably defined so that a different combination of global variables can be printed out if desired.

5.2. Error Handling Mechanism

The error handling mechanism in the editor is concerned with erroneous input by the user. Invalid control characters are ignored by the editor as are invalid commands in command mode, however these are accompanied by error messages. These error messages are printed on the command line and the editor returns to overtype mode with the cursor positioned at the previous cursor position.

When a user goes to leave the editor (CTRL-d) they are asked whether they wish to save the changes made. If a 'n' is typed in answer to the question the changes are not saved and if a 'y' is entered, the file is saved. However, as an added precaution, the user is able to change the character entered in answer to the question before the RETURN key is pressed.
6. REFERENCES


TED(1)

NAME
ted - a screen based text editor

SYNOPSIS
ted [filename]

DESCRIPTION
ted makes the terminal screen act as a "window" into the file filename. The area of the file around the cursor is the section which is displayed in the text window at any time.

The cursor can be moved around the file using the cursor movement keys, the TAB key (move to next tabstop), the ESC key (move to previous tabstop) and by using commands for longer movements.

There are three modes of operation in tedi. These are overtype mode, insert mode and command mode. The default mode is overtype. Text may be entered when in overtype mode or insert mode, while alphanumeric commands are entered while in command mode.

In overtype mode, the terminal acts like a typewriter. Any printable character which the user types at the keyboard is written at the cursor position. Therefore, if a character is already present at the cursor position, it is "overtyped".

The RETURN key inserts a newline immediately to the left of the cursor position. Thus, typing RETURN within an existing line splits the line, while typing it at the end of a line effectively creates a new line. Some commands are available only in overtype mode. Some of these are: CTRL-d to leave the editor (with optional save), RUBOUT to delete the character which the cursor currently flashes over and CLEAR to delete to the end of the current line.

Insert mode is used for the insertion of printable character strings. To put the editor into insert mode the user types BREAK.

Command mode is entered by pressing the HOME key. The user is then able to type in any of the commands which are available in command mode. (These are listed in the summary of commands)

ted works on a temporary copy of the file. The original file is not updated unless the save command (w) is issued in command mode, or the user saves it before terminating the editing session with a CTRL-d. This is done by typing in "y" when tedi asks if the file is to be saved.

LIMITATIONS
Any ASCII control characters are stripped out on loading and tab characters are expanded to the required number of spaces. When the file is saved, trailing spaces are removed from each line and leading spaces converted to tab characters wherever possible.

Lines are limited to 255 characters.

Currently, the hat character (^) cannot be used in arguments for search or substitute commands, as it has the special meaning of the blank character in these circumstances.
DIAGNOSTICS

Command not recognized for an unrecognizable command.
Usage [ followed by form of correct usage ] when a command is used incorrectly or when ted is invoked without a filename.
String not found when a string argument for a find or substitute command is not found
Line does not exist when an illegal line number is specified
If a CTRL-d is done without a w having occurred since the last buffer change, the user is asked if they wish to save the file or not before leaving the editor.

FILES
/pub/321/groupD/ted
/pub/321/groupD/READ_ME

SEE ALSO

AUTHORS

SUMMARY OF COMMANDS

Commands Available in Command Mode.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>l n</td>
<td>go to line n.</td>
</tr>
<tr>
<td>f [string]</td>
<td>find next occurrence of string.</td>
</tr>
<tr>
<td>s [s1] [s2]</td>
<td>replace next occurrence of s1 with s2.</td>
</tr>
<tr>
<td>g [s1] [s2]</td>
<td>replace all occurrences of s1 with s2.</td>
</tr>
<tr>
<td>d n1 [n2]</td>
<td>delete specified line or line range.</td>
</tr>
<tr>
<td>c n1 n2 [n3]</td>
<td>copy line (or line range) after n2 (or n3).</td>
</tr>
<tr>
<td>m n1 n2 [n3]</td>
<td>move line (or line range) after n2 (or n3).</td>
</tr>
<tr>
<td>w { f [n1 n2] }</td>
<td>save all lines (or line range) in file.</td>
</tr>
<tr>
<td>r f [n1]</td>
<td>read in UNIX file f after line n.</td>
</tr>
<tr>
<td>n [N]</td>
<td>move text window N windows forward.</td>
</tr>
<tr>
<td>p [N]</td>
<td>move text window N windows backward.</td>
</tr>
<tr>
<td>&gt; [N]</td>
<td>move text window N windows right.</td>
</tr>
<tr>
<td>&lt; [N]</td>
<td>move text window N windows left.</td>
</tr>
</tbody>
</table>
Commands Available in Overtype Mode.

CTRL-d       exit editor with optional save.
CTRL-v       re-display the screen.
RUBOUT       delete character at cursor position.
CTRL-w       delete word to the right.
CLEAR        delete to the end of the line.
PART 2

USER GUIDE
Users' Guide and Tutorial for TED – a Screen-based Text Editor

Wendy Bryant
Roderick Holmes
Raymond Overmass
Christopher Tisseverasinghe
TEAM D
CSCI321
Software Project
University of Wollongong

ABSTRACT

This document is a guide to using the screen-based interactive text editor called TED. It describes how to enter text and issue commands when using the editor. Uses of commands are explained as well as illustrated with examples.
1. Introduction

*ted* is an interactive full-screen text editor for use on UNIX text files.

The editor consists of three modes. The default mode is overtype mode, where each character typed at the keyboard replaces the character at the current cursor position. Insert mode is available for inserting additional text. Command mode is used for issuing various alphanumeric commands while control-shifted alphabetic keys and special keys are used in overtype mode for commands.

At the top of the screen is the Information line. This line contains the name and length of the file being edited, the current mode of operation, and the column range within the text window. These items are automatically updated whenever necessary.

Below the information line is the command line. This is used when in command and insert modes, and cannot be accessed at any other times.

Line numbers are displayed on the left side of the screen and the end of the file is shown by those words appearing immediately after the last line in the file.

2. Using the Editor

To begin editing a file type the UNIX command

```
ted filename
```

where *filename* is the name of your file. If the specified file already exists then *ted* will use that file. If the file does not exist, a new file is created.

When you want to leave the editor you type CTRL-d (press the d key while holding down the CTRL key). If you have not saved the file (using the *save* command) before typing CTRL-d a message will appear on the command line asking you if you wish to save the file before leaving the editor. If you want to save the changes that you have made to the file then answer "y" to the question and the file will be saved before the editing session is terminated. If you do not want to save the file then you answer "n" to the question and you will leave the editor without having made any alterations to the original file.

3. The Cursor

The cursor position is defined as being the position where the cursor is on the screen. The screen consists of a window into the text file displaying the 22 line by 71 column section of the file in which the cursor currently resides.

The cursor can be moved around the file by using the cursor movement keys (UP, LINEFEED, LEFT and RIGHT) which move the cursor one character or line in their respective directions. The TAB key (move to next tabstop) and ESC key (move to previous tabstop) are used for easier movement of the cursor. For larger cursor movements there are commands available in command mode to move the cursor to a different section of the file.

4. Overtype Mode

Overtype mode is the default mode of operation for *ted*. In this mode the editor works like a typewriter - each time a character is typed in, it replaces the character at the current cursor position. Overtype mode is used for main text entry as well as for altering mis-typed characters and inserting new lines of text.

Automatic indentation is a feature of *ted*. This involves positioning the cursor directly under the first character of the preceding line whenever RETURN is pressed.

Typing the RETURN key at the end of a line inserts an invisible newline marker at the cursor position and causes the cursor to move to the indented start of the newly created blank line. Any lines of text below the cursor are moved downwards to make room for the new line. As far as the user is concerned, this new line is a line of 255 blank spaces ready to be overtyped if desired. Thus when the user is creating a new file, they hit the RETURN key to obtain lines for entering text on.

Typing RETURN in the middle of an existing line causes the line to split at the cursor position. Everything to the right of the cursor position (including the character at the
cursor position) will be moved to the start of the next line on the screen.

Several commands are available in overtype mode. These are:

- **CTRL-d**: to leave the editor (see Using the Editor)
- **RUBOUT**: to delete the character at the current cursor position
- **CTRL-w**: to delete one word to the right of current cursor position
- **CLEAR**: to delete from the cursor position to the end of the line
- **CTRL-v**: to re-display the screen contents

5. **Insert Mode**

Insert mode is entered by hitting the **BREAK** key, whereupon the cursor moves to the command line and the previous cursor position is marked in inverse video. The text to be inserted is then typed on the command line. When the **RETURN** key is hit the text on the command line is inserted into the file in between the marked cursor position and the character immediately on its left. The editor is automatically returned to overtype mode with the cursor situated on the character after the text that was inserted.

Only 71 characters at a time can be inserted by using insert mode but this does not matter, as any large insertions (i.e. a line or more) should be done in overtype mode.

Insert mode is designed especially for inserting additional letters, words or phrases which are required in the middle of a line.

6. **Command Mode**

Although it is possible to issue a few commands whilst in overtype mode (deleting words, characters etc.), all other commands are issued while in command mode.

To change to command mode, the **HOME** key is pressed. This moves the cursor to the start of the command line as well as marking the previous cursor position on the screen in inverse video. An alphanumeric command is typed on the command line. Each command consists of one letter plus a number of arguments separated by blank spaces. Since the blank space is used as the command delimiter, when a blank space is required within an argument the caret character (\^) is used to signify a blank. The command is interpreted when the **RETURN** key is pressed.

If the command is invalid an error message appears on the command line and the editor returns to overtype mode with the cursor positioned at the marked cursor position.

If the command is accepted, the appropriate action is taken and the editor returns to overtype mode with the cursor positioned at the previously marked position or at the new position as determined by the command.

7. **Inserting Text**

Although **ted** has an insert mode, this mode is not used for most text insertion. Insert mode is used for inserting additional characters or strings into existing lines. (See Insert Mode)

Overtype is the main mode for text insertion. It is used for original text entry and for adding lines. This is possible because the **RETURN** key effectively creates new lines in overtype mode by inserting an invisible **newline** character.

8. **Deleting Text**

Whole lines of text can be deleted from the file by using the delete command (**d**) when in command mode. A single line or a range of lines may be deleted. When lines are deleted, all lines after the deleted lines move up to fill in the gap left by the deleted lines, thereby changing their line numbers.

As already mentioned (see Overtype Mode), deletions are also possible when in overtype mode. Control-shifted keys and special keys are used to delete characters, words and to the end of the line.
9. Moving the Text Window

The text window can be moved in several ways. The cursor can be moved out of the
text window (up, down, right or left) by using the UP, LINEFEED, RIGHT and LEFT
keys, or by issuing a command that changes text in a different part of the file. When
this happens, the text window is updated so that the portion of the file in which the
cursor resides is displayed.

Commands are available in command mode for long text movements. By specifying
numerical arguments with the commands n, p, >, and <, the text window can be
moved the specified number of window widths forward, backwards, right and left
respectively. For example:

```
n 4
```
moves the text window four window lengths forward.

10. Movement of Text

Commands are available in command mode for the easy movement of text within the
file. There is a copy command (c) which duplicates given lines after a certain line
number.

The move command (m) is similar to the copy command except that the lines of text
are actually moved to the new position, not duplicated.

In both these commands the last argument is the line number after which the lines are
written. For example:

```
c 16 25
```
copies line 16 after line 25, while

```
c 16 25 30
```
copies line 16 to 25 inclusive after line 30.

11. Searching

The find command (f) can be entered with or without a string argument. If an argument
is given, the cursor and the text window are moved to the first occurrence of the string
after the marked cursor position. The find command searches the whole file for the
string and if the string is not found, a message is written on the command line.

If the find command is issued without an argument the previous find command is
repeated from the marked cursor position. Typing in

```
f abcd
```
will find the next (i.e. from the cursor position) occurrence of abcd in the file. If the
command

```
f
```
is then entered the cursor will go to the next occurrence of abcd. If the find command
has not been used in the editing session when it is issued without an argument, an
error message appears on the command line.

12. Substitution

There are 2 substitute commands available in tedi. Single substitution (s) substitutes
the next occurrence of the given first string for the second string. For example:

```
s abcd xyz
```
will replace the first occurrence of abcd with the string xyz. Global substitution (g)
substitutes all occurrences of the first string for the second string.

A variation of the substitution command is available when no arguments are entered.
In this case, the command is executed using the arguments of the previously entered
substitute command. If no substitute command has been previously entered in the
current editing session, a message appears on the command line. A message also
appears if the string which is to be replaced is not present in the file.

13. Finding A Line
There is a command available (I) for moving the cursor and text window to a specified line. If the line number does not exist, an error message is printed on the command line and the cursor does not move.

This command is convenient for moving the text window to the first or last section of the file. Entering

```
1 1
```

will move the window to the first 22 lines in the file. If \( n \) is the last line number specified on the information line, then the command

```
1 n
```

will move the text window to display the last lines of the file.

14. Reading In Another UNIX File
TED allows the user to read in a UNIX text file into the file currently being edited. The named file (if readable to the user) is inserted into the file after the current cursor line or after the specified line number if this argument is given. For example:

```
r testfile 15
```

will put the contents of `testfile` after line 15 in the file being edited.

15. Saving the File
As already mentioned (see Using the Editor), the user has the option of saving the text file after issuing a `CTRL-d` to leave the editor. If the user wishes to save the file without leaving the editor, possibly for safeguard purposes, they may issue a command in command mode to do this. By typing in

```
w
```

the whole file is saved.

If the user types in

```
w testfile
```

the contents of the file being edited are saved in the file `testfile`.

If a line number range is specified after the filename, only those line numbers are saved in the given file. For example:

```
w testfile 10 50
```

would save lines 10 to 50 of the current file inclusive in the file `testfile`.

This command is useful in the dissecting of files.