Getting Started With ed

Version 1.12

1. Introduction

"ed" is the original UNIX editor, and as a consequence can be regarded as the "standard" editor. It is a line-oriented text editor used to create, manipulate, modify and display text files. Since ed reads textual commands from standard input, it can be used in scripts to automate editing tasks, as well as allowing interactive editing.

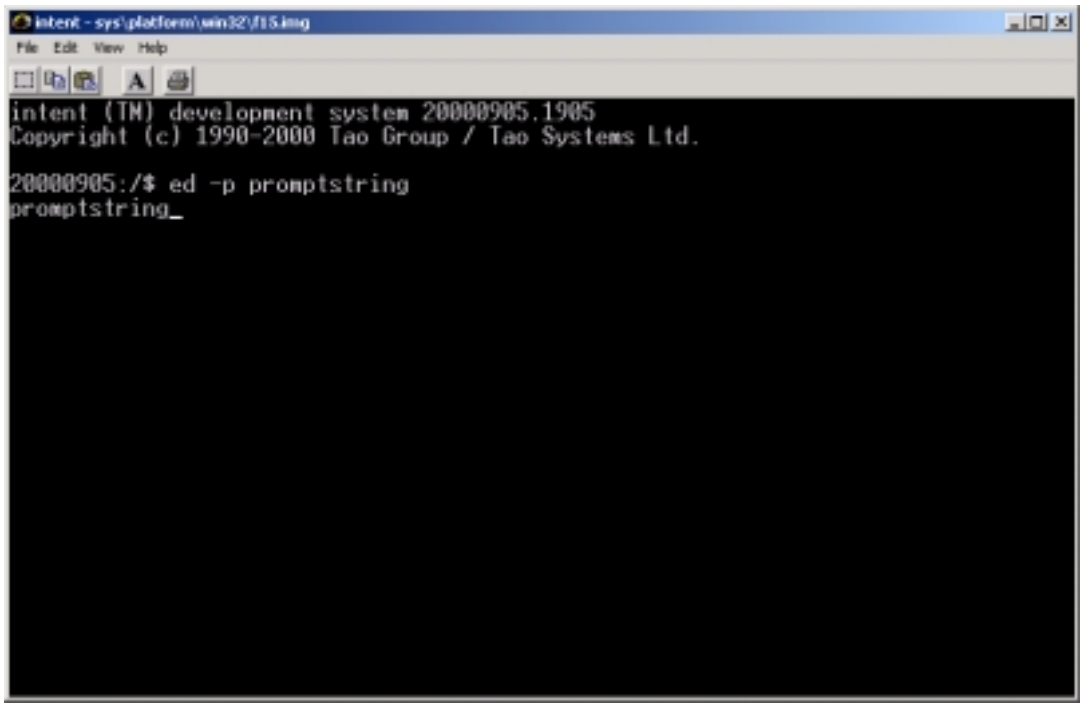
ed maintains a buffer, in which a temporary copy of a file can be edited. Changes made to the buffer contents do not automatically affect the contents of the file. In order to transfer such changes to the file, the buffer must be saved, and its contents explicitly written to the file.

1.1. Invoking ed

The ed editor can be invoked from the shell by typing the following command:

```
ed [-p <string>] [-s] [<filename>]
```

The option "-p" should be included if the user wishes to specify a prompt to appear when the editor is ready to receive a command. The <*string*> typed after "-p" should be the text which the user wishes to appear on the screen as the prompt.



In the example above, ed has been invoked using the command line:

```
ed -p promptstring
```

Thus, in the next line, the string "`promptstring`" is used as the command prompt.

The "`-s`" option causes the editor to be invoked in "quiet mode." In this mode, ed will not print out byte counts after executing "e", "`E`", "`r`" and "`w`" commands. Quiet mode also precludes the display of the "`!`" prompt after a ! command.

The "`-s`" option should be added if ed is being used by a script.

If a filename is specified at the end of this command, then the editor will automatically load the contents of the relevant file into a buffer so that it is ready to be edited by the user. If no filename is specified, when the editor is invoked an empty buffer associated with no file is made available to the user.

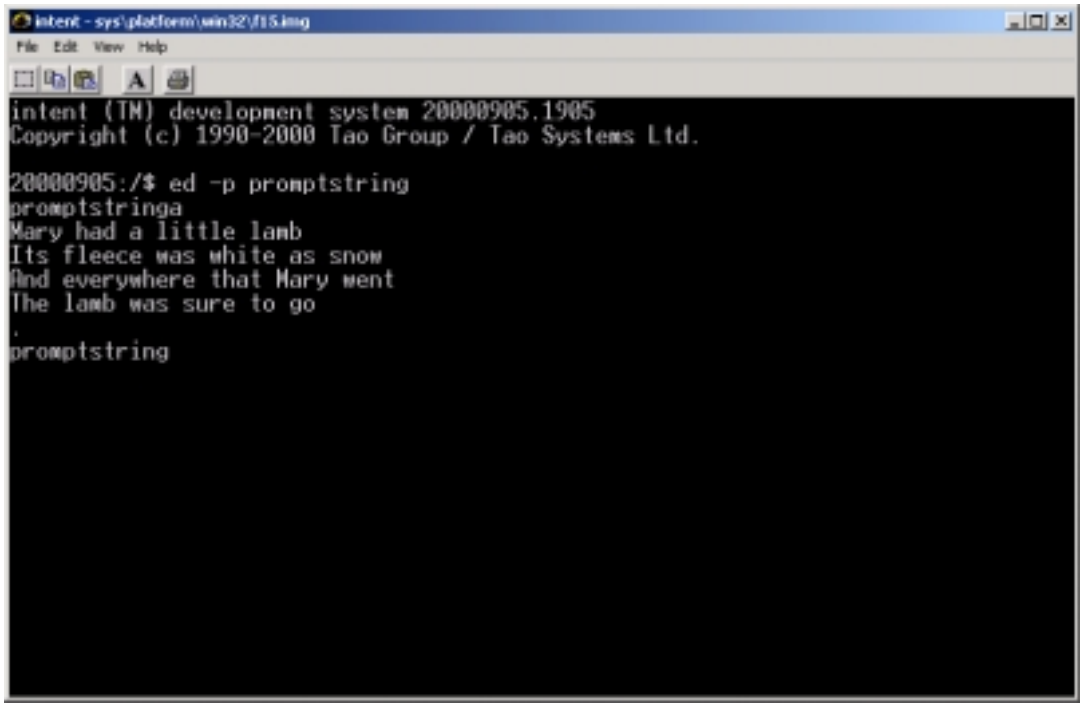
When first invoked, the editor is in "command mode." The editor makes use of the following two modes.

- Input mode:
  In this mode, input characters are treated as additions to the text.
- Command mode:
  In this mode, characters input by the user are interpreted as commands.

2. The Input Mode

When first invoked, the editor is in command mode. The editor enters input mode in response to commands such as "a", "i" or "c", which allow the user to append, insert and change text in the buffer, respectively. When the editor is in this mode, a prompt is never provided, and no commands are available. Instead, all that the user types is entered into the buffer.

In order to leave input mode and return to command mode, the user should enter a single period (".") on a line by itself.



In the example above, the command "a" has been used to enter input mode, some text has been typed into the buffer, and an isolated period used to return to command mode, as follows:

```
a
Mary had a little lamb
Its fleece was white as snow
And everywhere that Mary went
The lamb was sure to go
.
```

The final period returns the editor to command mode, and thus the specified prompt is displayed:

```
promptstring
```

3. The Command Mode

All ed commands follow the same form:

<address/range><command><arguments>['l', 'n' or 'p']

An address or range of addresses may be placed before the command, depending upon the requirements of the particular command. Different commands may take a maximum of zero, one or two addresses. The different address formats may be found in the section on "Addresses" later in this document.

The command itself is comprised of a single letter. The most useful ed commands may be found listed in the section on "Commands."

A different collection of arguments is associated with each command. Some of these arguments are optional, some mandatory.

After the arguments, it is not unusual to append "l", "n" or "p". This ensures that the "l", "n" or "p" command is executed after the completion of the main command.

4. Addresses

In ed, each address refers to a line in the buffer. There is, conceptually, a 'line zero,' which does not correspond to any line in the buffer, and which is not valid for all commands. At all times, one line is considered to be the 'current line.' The current line will often be the last line accessed by a command. In ed, a line consists of text up to and including a newline.

Commands in ed may be prefixed by zero, one or two addresses. These allow the user to reference single lines, or ranges of lines, in the buffer. All commands have default sets of addresses, which they will use if no addresses are specified.

Each address is constructed from the following components.

Component	Description
.	Represents the current line in the buffer. If there are no lines in the buffer, this is set to line 0. In some special cases, the current line may be set to line 0 even when there are lines in the buffer. Many commands, such as the "e" command, set the current line.
$	This is the last line in the buffer. If there are no lines, it is set to line 0.
n	Where n is an integer, this refers to the *n*th line in the buffer.
/<regexp>/	Refers to the first line subsequent to the current line which has a match with the specified regular expression. The text contained between the two forward slashes is the regular expression. The search for the specified string starts at the line following the current line, and proceeds forward through the text. If it reaches the end of the buffer without finding a match, it wraps around to the first line, and searches until it has reached the current line. If no line matches, an error is produced. (Regular Expressions are described in more detail in a section later in this document.)
?<regexp>?	Refers to the first line matching the specified regular expression, searching backwards. The search for the string starts on the line preceding the current. If the

	start of the buffer is reached, the search will continue backwards from the last line, until the current line is reached. If no line matches, an error is produced. It is possible to omit the trailing question mark, if it occurs directly before the end of a line.
'<*letter*>	This indicates the mark referred to by the specified lowercase letter. Marks are defined by using the "k" command, as described later in this document.
<*address*> +*n* or <*address*> -*n*	If an address is followed by a plus sign, and then the decimal number *n*, this specifies the line *n* lines after the named address. If the address is followed by a minus sign, than the number *n*, the component refers to the line *n* lines before the specified address. As a default, the address is taken to be the current address. The number *n* defaults to 1. In the "<address> +n" form, the "+" is optional. Trailing "+" and "-" characters are cumulative. Therefore, "--" would refer to the current line minus 2.

4.1. Address Ranges

Ranges of addresses may be expressed in the following ways.

Form	Description
<*address1*>, <*address2*>	Specifies a range of addresses between and including the two specified addresses.
<*address1*>; <*address2*>	This is the same above, except that the current line is reset to the value of the first address, so that the *address2* is calculated relative to *address1*.
, or ;	If the comma appears alone, and not as a divider for two addresses, then it acts as a shorthand for "1,$". Similarly, a lone semi-colon is shorthand for ".,$".

5. Regular Expressions

The regular expressions used by ed are POSIX Basic Regular Expressions (regexps). These are pattern matching mechanisms, and are commonly used by comparing them to a string to see if that string matches the pattern, or by searching within a string for a substring that matches.

Within ed, context search requests, the substitute command and the global command use a "regular expression." For example, when the user instructs the editor to seek out occurrences of a certain string within the text, a regular expression will be used.

A specific character is always used to delimit a regular expression. This will usually be "/". Thus, a regular expression might be input as

```
/april/
```

where "april" was the string that needed to be matched against the text. Where a regular expression can legitimately appear at the end of a line, it is not necessary to type in the closing delimiter. It will be implicitly added by the editor, which will append a "p" after the delimiter.

If the user types in an empty regular expression, such as

```
//
```

the editor replaces this with the last regular expression encountered.

Newlines cannot appear within regular expressions. In order to use a backslash as anything other than a regular expression delimiter, it is necessary to use the appropriate 'escape.'

Some other characters acquire a special meaning when included within a regular expression. They are listed below, along with their special character meaning, and the format of the escape required if they are to be included as normal characters.

Char	Escape	Purpose
*	*	The element preceding the * may be repeated 0, 1 or more times, and still match the pattern. Thus, the regular expression "smooo*th" will match with the strings "smooth", "smoooth", "smooooth", etc.
.	\.	This can be matched to any character. Thus the regexp "p.ck" may be matched to "pack", "pbck", "pcck", "pdck", "peck", etc.
[ ]	\[\]	These enclose a 'character set.' Like ".", a character set does not match with a single, literal character. Instead it provides a range of characters with which the string can be compared. Character sets take the following three forms. [aeiou] specifies that the characters "a", "e", "i", "o" and "u" are the members of the set. [^aeiou] specifies that all characters **except** these five are members of the set. [[:<*class-name*>:]] specifies that all characters of a predefined class are members.
\(… \)		A section of a regexp that delimited in this way is a subexpression. This can be used to group regexp constructs so that other operators can be applied to the whole group.
\+	Omit "\"	"+" is similar to "*" except that it requires the preceding element to be matched at least once. Thus the regexp "smooo+th" would not match the string "smooth".
\?	Omit "\"	This indicates that the preceding element is optional. It may be matched once, or not at all.
\{*m*,*n*\}	Omit "\"	An expression of this sort, where m and n are integers and *n*>*m*, indicates that the preceding element must match at least m times, and no more that *n* times.
|	Omit "\"	This is used to divide several regular expressions that are listed as alternative matches for a particular string.
n	Omit "\"	This is a backreference. It matches a repetition of text which matched the nth subexpression. The subexpressions are counted forward from the start of the regexp

6. Commands

Each command is comprised of a single letter, and may accept a maximum of zero, one or two addresses, depending upon the command. It is illegal to exceed the maximum number of addresses appropriate for a particular command. Each command has some default addresses which it can use if insufficient addresses are specified. If a single address is inserted where a range is required, the editor will interpret it as a range in which the starting line and the ending line are the same.

Where a command takes a file argument, the file may be expressed as a pathname, or as a !command-line. Where the ! form is used, ed reads the output of the command as input, or writes the output to be used as the input of the command. Filename arguments may be preceded by whitespace, and extend to the end of the line. Suffixes cannot be appended to commands that may use filename arguments.

Within this section, the most commonly used ed commands are listed, along with their default addresses, and the number of addresses each command requires.

6.1. Adding, Deleting and Copying Text

Command	Description
`.a`	This allows the user to type in text that will be inserted into the buffer. The text is inserted after the address line specified before the command. Valid addresses range from 0 to $. If address zero is specified, the text is inserted at the start of the buffer. If no such address is specified, the text is inserted after the current line. The editor reads text from the terminal into the buffer until a line containing only "." is entered, or until the end of input. At this point it is once again ready to accept commands. After the execution of this command, the current line indicator is set to the last line inserted.
`.i`	Allows the user to insert text into the buffer in much the same fashion as command "a". Unlike "a", however, this command inserts the text before the specified address, rather than after it. After the command has executed, the current line is set to the last inserted line. If no lines were inserted, the specified address becomes the current line.
`.,.c`	This command operates upon a range of addresses. All the lines within this range are deleted from the buffer, and the user is permitted to enter text to be inserted in its place, in the same fashion as for command "a". After the command has executed, the current line is set to the last line to be inserted. If no text was inserted, the current line is that following the last line to be deleted. If the deleted lines ran to the end of the buffer, the current line is set to "$".
`.,.d`	Like "c", this command operates upon a range of addresses, deleting all the lines included within it from the buffer. The current line is set to the first line after the deleted range. If the deleted range ran to the end of the buffer, the current line is set to "$".
`.,.t` <*address*>	Copies the specified range of lines, and inserts this copy after the given address. If no address is given, <*address*> defaults to the current line. If <*address*> is 0, then the copy is placed at the start of the buffer. If <*address*> lies within the specified range of lines, the command is invalid. The last line copies becomes the new current line.

6.2. Search and Replace

`.,.s/` <*regexp*>`/` <*new*>`/` `[flags]`	Seeks out a specified regular expression within the stated range, and replaces it with a new string. The editor searches each line within the specified range for strings matching <*regexp*>. The first matching string discovered will normally be replaced by the string <*new*>. A newline character can be included in the <*new*> string if preceded by a backslash. Thus a substitution of this sort is useful for splitting a line into two. If the specified <*new*> string consists only of a percent sign (%) then the editor will use the <*new*> from the previous "s" command. If an ampersand (&) appears in <*new*>, the character is replaced by the text matching <*regexp*>. A literal ampersand may be used, if preceded by a backslash (see section on regular expressions). "\1", "\2", etc are replaced by the corresponding matching subexpression. Although in the example to the left, the forward slash has been used to divide the different sections of the command, any character may be used as a delimiter, providing that it is used consistently throughout the command. The trailing delimiter may be omitted. In these cases a "p" suffix will be implicitly added. There are several optional flags which this command may use, alone or in combination. The "g" flag globally substitutes all non-overlapping occurrences of the regular

	expression, rather than just the first. If an integer *n* is used as a flag for this command, only the *n*th occurrence of the regular expression is replaced by the *new* string. The commands "l," "n" or "p" can be used as suffixes. These cause the last line in which a substitution was made to be displayed on the screen in the style appropriate to the command.

6.3. The Remembered Filename

The 'remembered filename' is usually that of the last file to be read into a buffer. Many commands that require a filename parameter use the 'remembered filename' as a default. With the use of the following commands, the 'remembered filename' can be printed to the screen, or altered to a different filename.

`f` <*filename*>	If a filename follows this command, the 'remembered filename' is set to this filename. Whether a filename is specified or not, this command prints the 'remembered filename' to the screen. The current line remains unchanged.

6.4. Loading File Contents into a Buffer

`e[]`	Replaces the content of the current buffer with contents of a specified file. Where no file is specified, the editor will load the contents of the 'remembered file' into the buffer. If a file is specified, then its contents are read into the buffer, and it becomes the new 'remembered file.' Unless the editor is in 'quiet mode,' a count of the bytes in the file is printed to the screen. It may be that the buffer's existing contents have been altered since the last time they were written to file. In such circumstances, the editor will notify the user of this before discarding the old text in the buffer. The command is not immediately executed, and the user is given the opportunity to abandon it. If, however, the user repeats the command, the warning is not repeated, and the command executes.
`E[]`	This performs much the same operation as the "e" command. The only difference is that if the "E" command is used, the editor will not warn the user of changes since the last save, and will instead execute the command immediately.
`($)r [`<*filename*>`]`	Inserts the contents of the specified file into the text occupying the buffer. The contents of the specified file are placed after the specified line in the buffer. If no filename is given, the 'remembered filename' is used. If the specified address is 0, the inserted text is placed at the start of the buffer. If the last byte read is not a newline, then a newline is implicitly created at this point. If no 'remembered filename' exists, any filename that has been specified in this command, and does not begin with "`!`", will become the new 'remembered filename.' The last line inserted from the file becomes the new current line.
`($)r !command`	If "`r`" is followed by a filename that starts with "`!`", the rest of the line is interpreted as a shell command. This command is run, and its output read into the buffer instead of the file contents.

6.5. Saving Text to File

`1,$w [<filename>]`	Writes the specified range of lines to the named file. If no file is specified, the 'remembered filename' is used. If no 'remembered filename' exists, the specified filename becomes the 'remembered filename.' The location of the current line does not change. The editor will display the number of bytes saved to file.
`1,$w !<command>`	When the "w" command is followed by a filename that starts with "!", it is interpreted as a shell command. This command is run, and the specified range of lines are written to its standard input. The 'remembered filename' and current line are unchanged. The number of bytes written is displayed at standard output.

In the example below, ed has been invoked, and text typed into the buffer using the "a" command. The text is the buffer has then been saved by typing:

```
1,$w Mary
```

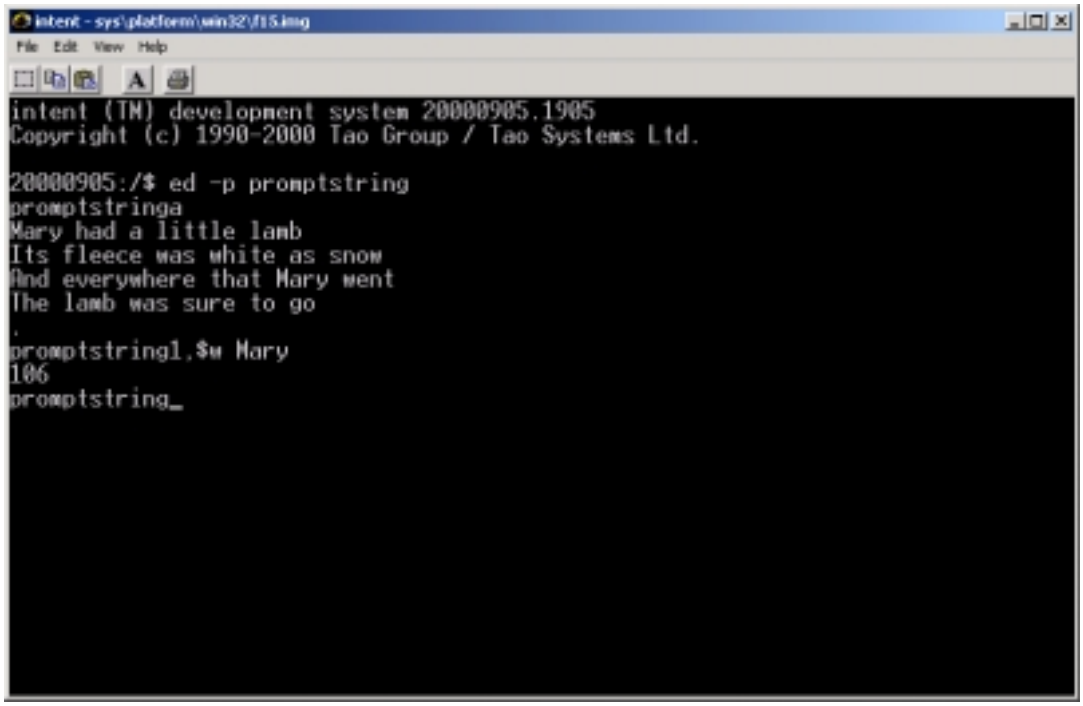
The "w" command writes the text in the buffer into a file. The address range specified "1,$", and the command therefore saves the entire contents of the buffer. The filename under which it is saved is "Mary".

After saving the file, the editor has printed out the number of bytes saved:

```
106
```

The editor then returns to command mode, and the customary prompt appears on the next line:

```
promptstring
```



In the figure below, ed has been re-invoked. Since no filename has been specified with the "ed" shell command, the editor has opened an empty buffer. The user has entered text into this buffer using the "a" command, as follows:

```
a
Hey diddle diddle
```

```
.
```

Subsequently, the user has attempted to load the contents of the file "`Mary`" into the buffer by typing:

```
e Mary
```

Since the buffer has been modified since ed was invoked, the editor warns the user of this with the general error message:

```
?
```

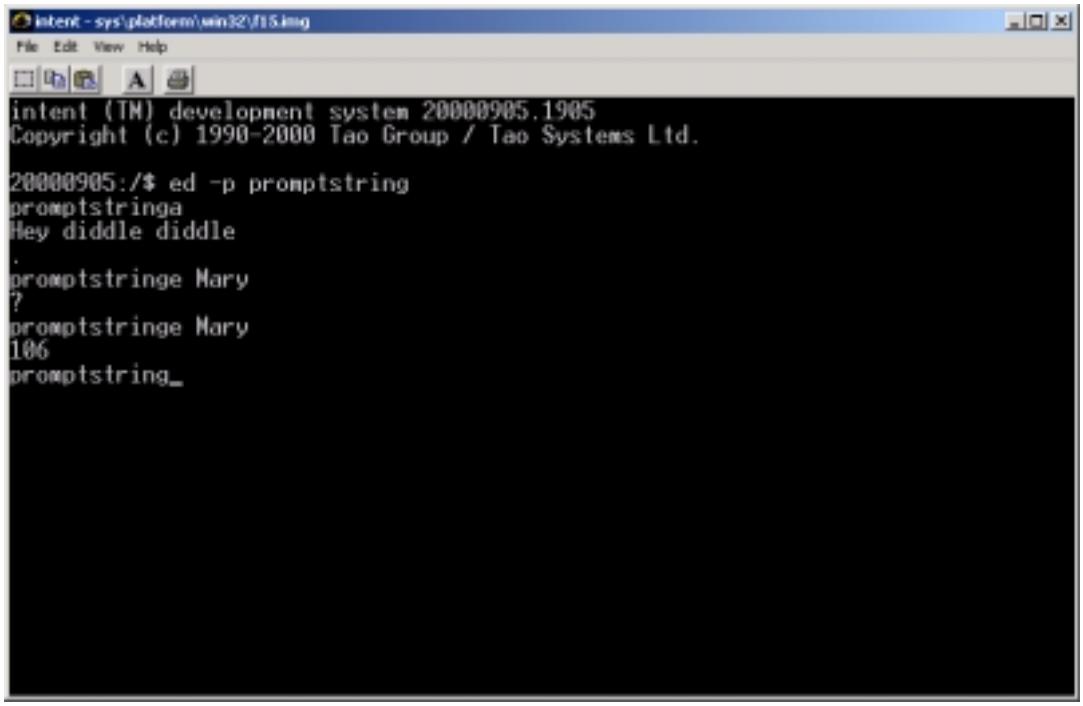
(It would be possible to acquire a more detailed message from the editor by typing in the command "h". More details of this can be found in the later section on "Getting Help." In this case, typing "h" would have caused the message "`warning buffer modified`" to be displayed on the screen.)

The user has here decided to persist in attempting to load the contents of the file into the buffer. The command "`e Mary`" is entered a second time. This time the editor loads the contents of the file into the buffer, then displays on the screen the number of bytes loaded.

```
106
```

The text of the loaded file is not displayed at this point. Instead, the editor returns to the command mode prompt.

```
promptstring
```



6.6. Display Contents of Buffer

It should be noted that ed does not automatically display changes to the buffer on the screen. In order to view a range of lines in the buffer, the following commands should be used. These commands can be used alone, or they may be placed as suffixes at the end of another command, so that its effects can be viewed. "`l`", "p" and "n" may be used as suffixes for any commands except "e", "`E`", "`f`", "`q`", "`Q`", "`r`", "`w`" and "`!`".

In the following example, text has been entered into the buffer using the "a" command, as shown previously. The "d" command has then been used to delete the third and fourth lines.

```
3,4d
```

The succeeding command has been used to print out the entire contents of the buffer in 'signposted' fashion, hence the use of indentation and line numbers.

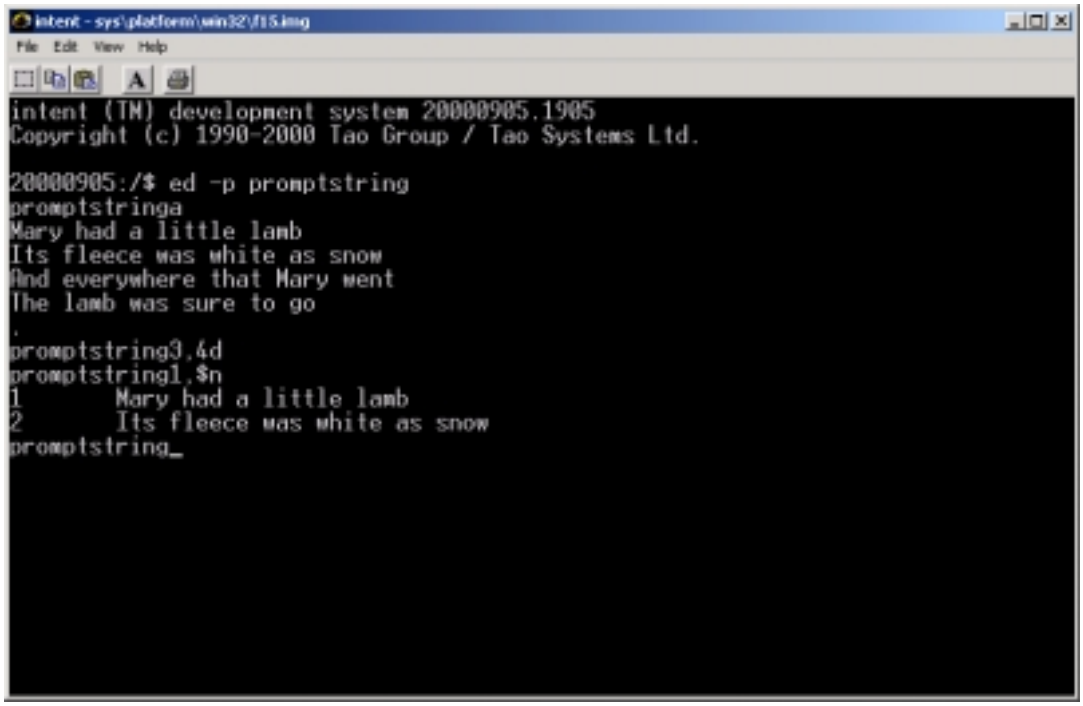
```
1,$n
```

The editor displays the following,

```
1     Mary had a little lamb
2     Its fleece was white as snow
```

before returning to the command mode prompt.

```
promptstring
```



`.,.p`	Prints the addressed range of lines to standard output. The last line displayed becomes the new current line.
`.,.n`	Prints the addressed range of lines to standard output like "p", but in a 'signposted' fashion. All lines are indented by a tab, and line numbers are listed down the left margin. The last line displayed becomes the new current line.
`.,.l`	Prints the addressed range of lines to standard output in visually unambiguous form. The last line displayed becomes the new current line. Nonprintable control characters and backslashes are represented in the C backslash escape form. Thus, long lines are split with a backslash newline sequence , and each line is terminated by a "$". This style of output is the same as that generated by the "`l`" command in sed.
.+1	If a single address is given without an explicit command, the addressed line is displayed as if the "p" command had been used.

6.7. Adding/Removing the Prompt

The editor allows the user to choose whether a prompt is displayed. If the prompt display mode is activated, the prompt string will appear whenever ed is awaiting a command. It will not be displayed when the editor is awaiting the input of text to be inserted in the buffer.

P	This is a toggle command that turns prompt display mode on or off. As a default, this mode is off. When ed is invoked, the prompt display mode can be turned on by using the "`-p` <*string*>" option with the "`ed`" command. This also allows the user to specify the string that will be used as a prompt. The "P" command can subsequently be used to disable this prompt, or to re-enable it. If no string has been specified through "`-p`", no prompt will initially be displayed. If a "P" command is subsequently issued, the editor will display "`*`" as a prompt.

6.8. Manipulating Buffer Lines

Since ed is a line-oriented system, and most commands make use of the line-based addressing system, it is useful for the user to be able to adjust the format of the lines, to 'mark' certain lines with labels, or to find out the line numbers associated with certain 'mark' labels.

`.,.+1j`	Joins the addressed range of lines to become one line. All newline characters except the last are removed from the range. The current line is set to the new, joined line.
`.k`<*letter*>	In a long document, it may become difficult to keep track of line numbers, particularly if the latter are continually changed by edits performed upon the buffer. As a consequence, it is often useful to 'mark' a line. The "k" command 'marks' the specified line with the mark that corresponds to the name <*letter*>. The letter used to name a mark should be lower case. Subsequently, the user can refer to the line in question using the construct '<*letter*>. <*letter*> remains attached to relevant line regardless of edits performed upon the buffer. Since lower case letters are used as names, it is possible to mark as many as 26 lines in a buffer.
`$=`	Displays the line number of the specified line to standard output. This may be useful for finding the line number of the current line, or of a 'marked' line. Address zero is valid. This command does not change the current line.

6.9. Undoing Commands

U	Undoes the last command to modify the buffer. For the purposes of "u", the commands which create such modifications are "a", "c", "d", "`i`", "`j`", "`r`", "s", "`t`", and "u" itself. These commands count as modifications, even if they have not actually changed the buffer contents. Since "u" is considered to modify the buffer, a second consecutive use of the command will return the buffer to its state before the first. The current line returns to the position it held before the "undone" modification.

6.10. Help messages

h	Displays a help message at standard output. This explains the last error to occur.
H	This is a toggle command which turns a help message mode on or off. As a default, the editor will simply print "?" to the screen if an error occurs. If the help message mode is turned on, ed will display a brief help message explaining each error after the "?" notification. If an error has already occurred when the mode is turned on, ed will provide a help message as if an "h" command had been entered.

6.11. Exiting ed

q	Causes the editor to exit. If unsaved changes have been made to the buffer, ed will notify the user of this, and the command will not immediately execute. If after such a warning, however, the command is repeated, no other warning is given, and the editor will exit.
Q	Causes the editor to exit. Unlike the "q" command, "Q" does not cause the editor to warn the user of unsaved changes, but instead executes immediately.

6.12. Other Commands

!<*command*>	Runs the command <*command*> as if the latter had been typed into the user's chosen command interpreter. "! !" may be typed to reissue the previous <*command*>.