(Very) Small Editors

By Jerry Peek

Linux systems use text pervasively and provide an almost-infinite number of tools to manipulate it. This month, let's look at three lesser-known text handling tools: the line editors *ex* (which is usually part of the *vi* editor) and *ed*, and the stream editor, *sed*.

Most of our examples use regular expressions (or "regex"). If you aren't familiar with regex, see the article "Hitting the Motherlode," available online at http://www.linux-mag.com/2002-07/regex_01.html. (Depending on the version you have, *ed*, *ex*, and *sed* may not handle regular expressions as sophisticated as the ones handled by Perl, but they'll all understand at least the metacharacters `^ $ & * \ . [-]`.)

Scripted editing and more with *ed* and *ex*

Before *emacs* and *vi* came along, *ed* was *the* Unix editor. Designed in the days when a Unix "display" was a printing teletype, *ed* reads commands from its standard input — usually your keyboard or a file. It only displays the text you're editing if you use its `p` ("print") command. The original version output only a `?` after an error, leaving you to figure out what went wrong. The *ex* editor, which comes with many versions of *vi*, is an extended *ed* with more commands and friendlier output.

Since we have powerful visual editors these days, why use an ancient beast like *ed* or *ex*? Here are three reasons:

- If your terminal or system is corrupted, or if you're trying to edit over a clogged network link, using a line editor can mean the difference between getting your edits done and not. This can be important when, say, you need to edit a system configuration file but you can't run a screen-oriented editor. If you're a system administrator, you should know how to use a line editor!

- You can write editing scripts — either in a file or "on-the-fly" — to edit files automatically.

- When you're using *vi*, you can switch to *ex* mode and do things you can't do (easily) from standard *vi* mode.

If you haven't used a line editor, please see a good Linux or Unix book for details. We'll introduce them, then quickly get into specific examples.

ed and *ex* commands have basically two parts: an *address* (or a range of addresses) and an *operation*. However, only one of those is required. An address tells where to perform the operation; the default, in most cases, is the current line. An address by itself — without an operation — simply changes the current line and displays the line you've moved to. Most operations make a change to the file.

The address can be for a single line or a range of lines. You can give explicit line numbers, where `1` is the first line, and `$` stands for the last line number, whatever that is. You can also use regular expressions that match one or more lines: just put the regex between forward slash (`/`) characters.

Let's see an example. The command…

```
1,/^#/d
```

… reads, "Delete all lines between line 1 and the next, subsequent line starting with a `#`." Here, the operation, `d`, deletes the lines; `1` refers to the first line and `^#` refers to the next line that begins (the `^` refers to the start of the line) with a `#`. When decoding commands like this, train your eyes to break commands into the address range and the operation.

Here's another example of addressing. You'll often want to match some line *before* another line. For instance, you might want to delete all lines from the first line until the line *preceding* the comment character. Here's how:

```
1,/^#/-1d
```

If you compare this to the previous example, you'll see that we've changed the range-ending line from `/^#/` to `/^#/-1`. In *ed* and *ex*, you can make a *relative* line address with a `+` (to increase the line number) or `-` (to decrease it). So, the previous command reads, "Delete all lines between line 1 and the line before the next line starting with `#`, inclusive."

Let's finish our whirlwind syntax tour with one of the most common operations: `s`, for *substitute*. The `s` command has basically two parts: a *pattern* and a *replacement*, separated by forward slash (`/`) characters. The pattern is a regular expression; the replacement is literal text with a few twists.

Here's an example: we want to add a comment character (`#`) and a space to the start of every line in the same range (`1,/^#/-1`) we used previously. (That is, we want to "comment out" those lines.) The operation to use is `s/^/# /`, which reads, "At the start of the line (`^`), insert `#` and a space." Here's the whole command, including the address range:

```
1,/^#/-1s/^/# /
```

We've jumped right in to one of the most cryptic exam-

ples — not to scare you away — but to show you the expressive power of line editors. With just a few keystrokes, you can do something complex. Once you've learned the syntax well, you can even type commands over a slow data line with no "echo" to show what you've done.

If you want to perform that same command on all of the files whose names end with *.cfg*, a simple Bourne shell loop can invoke *ed* on each file. Use *echo* and a pipe to write editing commands to *ed*'s standard input (which is where *ed* gets its commands). Here's the script:

```
for file in *.cfg
do
  echo '1,/^#/-1s/^/# /
  w' | ed "$file"
done
```

echo can send more than one line of commands to *ed* by surrounding multiple lines with quotes. The `w` command (the second line that *echo* outputs) tells *ed* to write the edited file. There are lots of other *ed* operations, but let's jump ahead and take a brief look at *ex*.

Introducing the *ex* Editor

The *ex* editor, an extended *ed*, is built-in to most versions of *vi*. There are two ways to run *ex* commands from within *vi*:

- Type a colon (`:`), followed by an *ex* command. After you press `RETURN`, the command runs and places you back in *vi* mode. (Some *ex* commands you might have used already include `:%s` to do a global substitution, `:w` to write the file, and `:q` to quit.)

- Type `Q` (uppercase "Q") to enter *ex* mode. (If your version of *vi* supports this, you should see a colon (`:`) prompt.) Type *ed* or *ex* commands. To go back to *vi* mode, type `vi`.

For instance, if you're editing a file with *vi* and want to "comment out" all of the lines from the top of the file to just before the first comment, you could type the same *ed* command we saw earlier, preceded by a colon:

```
:1,/^#/-1s/^/# /
```

That command adds a hash sign (`#`) to every line, whether there's any text on it or not. Because you'll be back in *vi* mode immediately after, you can see the effect of your edit. (You can press `u` to undo the command, or use other *vi* commands to make other changes).

You can do more with the *ex* (and *ed*) "global" operation, `g`. It steps through each line in a range, performing another operation on that line. Let's see the command, then talk about it:

```
1,/^#/-1g/./s/^/# /
```

(From *vi*, type a colon (`:`) before that command.) `1,/^#/-1` is the address range we've used before. The `g` is followed by a regular expression, `/./`, which matches any line with at least one character. The rest of the command, `s/^/# /`, is executed on all matching lines; as we saw before, it prepends a comment character and a space.

So, to sum up the previous command, on all lines from line 1 to the line just before the first comment character (`1,/^#/-1`), on all non-empty lines (`g/./`), add a comment character and a space to the start of the line (`/^/# /`).

The *sed* Stream Editor

Most text editors are designed to edit files. The *sed* editor is different: it edits a *stream* of text from its standard input, and emits the edited version of the text to its standard output. So *sed* doesn't edit a file; it edits a "stream" (hence its name) of text "on the fly."

sed commands look like *ed* commands, with a couple of important differences:

> The *sed* editor is different: it edits *stdin* and emits changes to *stdout*

- In *ed*, the default address is the current line. (That is, if you don't give an address, *ed* operates only on the current line.) In *sed*, the default address is *every line*. So, for instance, the *sed* command `s/^/# /` would add a comment character to the start of every line, while the command `1s/^/# /` would add it only to line 1.

- Relative addressing, like "the line before the next line with a comment character," doesn't work in *sed*. That's because *sed* reads the text stream line-by-line. (You can work around this with *sed* loops and multiline patterns.)

Listing One (pg. 36) has a familiar example: "commenting out" all lines from line 1 to the first line with a comment character, inclusive. *sed* can accept commands on its command line, but it's best to use the `-e` option.

To make this more obvious, we'll violate the old rule that "Using *cat* with just one filename means you're doing something wrong," use *cat*, and pipe a file to *sed*'s standard input. You'll often pipe *sed*'s output somewhere else — to another program or the printer, for instance — but we'll simply display it on the screen.

LISTING ONE: A simple file, and a simple *sed* edit

```
$ cat afile
This is line 1.

This is line 3.
# Line 4 starts with a comment.
Line 5 doesn't.
$ cat afile | sed -e '1,/^#/s/^/# /'
# This is line 1.
#
# This is line 3.
# # Line 4 starts with a comment.
Line 5 doesn't.
```

In *Listing One*, *cat* displays the original, unedited file. Next, *cat* is piped to *sed*, which adds a comment character to the (#) beginning of several lines. You can see that *sed* added an extra comment character to the start of line 4 before it stopped editing. Let's fix that.

You can put editing commands in a file. Our file is called is *sedscr*, which includes a test-and-branch command before the `s` command. *Listing Two* shows the script and the result.

sed's grouping braces let you apply a series of commands to all lines in a range of addresses. Our group of commands starts with the familiar range we've seen: `1,/^#/`, which means, "All lines from line 1 through the first line starting with a # character." On each matching line, *sed* attempts the braced commands from first to last:

- The first command, `/^#/b`, has the address `/^#/`. This matches comments (lines starting with #). On those lines, *sed* executes `b`, which skips the rest of the editing script. So, this first command doesn't take effect on lines 1 through 3, but on line 4, the command makes *sed* branch — and skip the next command, which would have added another # to the start of line 4.
- The command `s/^/# /` should be familiar by now. It operates on every line of input matching `1,/^#/` — except line 4 (due to the `/^#/b` command) and line 5 (which is outside the range `1,/^#/`).

LISTING TWO: A *sed* script with nested patterns

```
$ cat sedscr
1,/^#/ {
   /^#/b
   s/^/# /
}
$ cat afile | sed -f sedscr
# This is line 1.
#
# This is line 3.
# Line 4 starts with a comment.
Line 5 doesn't.
```

Why use this unique editor instead of, say, Tcl or Perl? *sed* is small and fast, which is useful for busy machines like mail servers that can use *sed* to edit email messages on-the-fly from within *procmail*. Let's see a more complete example.

sed can collect multiple lines, editing them as they're collected or later. There's also a special *hold space* where you can store lines temporarily. Let's use these two *sed* features and make an editing script that joins continued lines (lines starting with spaces or tabs) in a mail message header. A header could look like this before editing...

```
From: Jerry Peek <jpeek@xyz.pdq>
To: joe@abc.pdq,
    aecho@bcd.pdq
Subject: A test message
```

... and this afterward:

```
From: Jerry Peek <jpeek@xyz.pdq>
To: joe@abc.pdq, aecho@bcd.pdq
Subject: A test message
```

Listing Three shows a script to join the lines. It appends input lines to the hold space until a non-continuation line is found; then the hold space is output and the new line is held.

All of the commands are inside an address range `1,/^$/`, so they apply only to the message header (which always ends with an empty line, matched by `/^$/`). The group of commands between the inner curly-braces are only applied to continuation lines. (Although you cannot see them, the

See Power Tools, *pg. 70*

LISTING THREE: *sed* script using *hold space*

```
1,/^$/ {
  # Join continuation lines:
  /^[   ]/ {
    x
    G
    s/\n[       ]*/ /
    h
    b
  }
  # This is a non-continuation line.
  # Hold it and output previous line
  # (except line 1, when nothing's held):
  x
  1!p
}
```

Power Tools, *from pg. 36*

braces in `/^[ ]/` have a single space and a single TAB character inside). Let's see how this works on the `To:` field.

1. When we reach the commands matched by `/^[ ]/`, the *pattern space* (which has the current input line) contains `aecho@bcd.pdq,` and the hold space has the previous input line `To: joe@abc.pdq,`. The `x` "exchange" command swaps these two. Now the pattern space has the previous line `To: joe@abc.pdq,`.

2. The "get" command adds a newline and the line from the hold space onto the pattern space. Now the pattern space has both lines of the `To:` field.

3. The `s` command replaces the newline (`\n`) and all spaces and tabs at the start of the next line (after the newline) with a single space. This joins the lines.

4. The `h` or "hold" command puts the joined line into the hold space.

5. The `b` command skips the rest of the script, since those commands do not apply to a continuation line.

The rest of the script is outside the inner curly braces, so it's only read for non-continuation lines. The `x` command swaps the current line with the previous, held line(s). The `1!p` command uses the `!` or "not" operator. It says, "On all lines *except* line 1, print." (When we're reading line 1, which is never be a continuation line, there's nothing to print.)

This compact language takes some time to get used to. The *sed* FAQ, at http://sed.sourceforge.net/sedfaq.html, has much more information.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.

POWER TIP: SUSPENDING *SSH*

The *ssh* program, and its insecure cousin *rsh,* open a login session to a remote system. You can't access your local system from that terminal while *ssh* is running, but you can use job control to suspend your *ssh* session.

Type the *ssh command string,* which is RETURN followed by a tilde (~) character; then type the usual job-control suspend character CTRL-Z. To resume the session later, use the job-control "foreground" command, *fg.*

The *ssh man* page lists more of these handy commands.

Advertisers' Index

The Advertisers' Index lists each company's Web address and advertisement page. To advertise in Linux Magazine, please contact adsales@linux-mag.com for a media kit containing an editorial schedule, rate card, and ad close dates.

Linux Magazine (ISSN 1536-4674) is published monthly by InfoStrada LLC at 234 Escuela Avenue #64 Mountain View, CA 94040. The U.S. subscription rate is $29.95 for 12 issues. Canadian Post Corporation Agreement No. 40048133. Send returns to WDS, Station A. PO Box 54, Windsor ON N9A 6J5. In Canada and Mexico, a one-year subscription is $59.95 US. In all other countries, the annual rate is $89.95 US. Non-US subscriptions must be pre-paid in US funds drawn on a US bank. Periodical Postage Paid at Mountain View, CA and additional mailing offices.

POSTMASTER: Send address changes to Linux Magazine, P.O. Box 55731, Boulder, CO 80323-5731.

Article submissions and letters should be e-mailed to editors@linux-mag.com. Linux Magazine reserves the right to edit all submissions and assumes no responsibility for unsolicited material. Subscription requests should be e-mailed to linuxmag@neodata.com or visit our Web site at www.linuxmagazine.com.