nroff Text

Processor Tutorial

Table of Contents

1. Introduction

nroff is the text formatting program provided by the COHERENT™ system. Working with **nroff** is easy. You provide both the text you want processed and commands to control the processing; the command lines are interspersed among the lines of text.

This tutorial describes how to work with **nroff**. It assumes you are familiar with the basic features of the COHERENT system. In particular, you should know what a *command* is, what a *file* is, and how to create and edit a file. If you are not familiar with these concepts, you should read the *Introduction to the COHERENT System* before you read this tutorial. Other relevant COHERENT manuals include the *ed Interactive Editor Tutorial* (which provides more detailed information on the COHERENT text editor ed) and the *COHERENT Command Manual* (which gives concise descriptions of COHERENT commands).

The input you give to **nroff** may be a file you have written or you may have **nroff** accept input directly from your terminal. This choice is made when you initially call it. In either case, **nroff** normally prints its output on your terminal. If you simply type

```
nroff
```

then **nroff** accepts input from your terminal and prints its output there. If you create a file named **script.r** which you want **nroff** to process, type the command line

```
nroff script.r
```

nroff processes **script.r** and prints its output on your terminal. The suffix **.r** is often used to indicate that a file contains **nroff** input. You may save the output by redirecting it to another file target:

```
nroff script.r >target
```

If your COHERENT installation provides a line printer, you can print copies of the output on it; you might use a *pipe* to funnel the output of **nroff**'s activity to the line printer:

```
nroff script.r | lpr
```

As you will discover in working through this tutorial, it is possible for you to control all significant aspects of the output's appearance. An unexpected consequence of this, though, is that since you have ultimate control over almost everything, **nroff** does very few things without specifically being commanded to do them. It does not automatically leave margins at the top and bottom of pages; it does not automatically number pages; it does not automatically format paragraphs. You must use predefined or create your own sets, of **nroff** commands, called *macros*, to produce these features if you want them. In a sense, when you work with the basic **nroff** command, the script you write for **nroff** to process is a program that tells **nroff** what to do with your text.

An **nroff** macro package called **ms** provides predefined ways of formatting paragraphs, producing header and footer areas (the areas at the top and bottom of pages, respectively), and so on. Using the macro package is easy. The command

```
        nroff -ms
```

accepts input from your terminal and prints output there;

```
        nroff -ms script.r
```

processes the file **script.r** and prints the output on your terminal;

```
        nroff -ms script.r >target
```

redirects the output to another file; and

```
        nroff -ms script.r | lpr
```

prints the output produced from processing **script.r** on the line printer.

Using **nroff** with the **ms** macros is easier than using the **nroff** itself, since many output format design decisions have already been made. The mechanics of creating an acceptable input script for **nroff** – **ms** are no different than they are for the basic program. Working with the macro package is a good way to gain confidence in working with **nroff** commands.

The only way to learn about **nroff** is to use it. You should try all the examples in this tutorial, as well as altering them and examining the resulting output. You should also create your own examples. Don't hesitate to experiment; you can often learn more from analyzing why something unexpected happens than you can from simply copying an example that works as expected.

Section 1 of this tutorial describes using **nroff** with the **ms** macro package. It should be sufficient for the needs of many users. Sections 2 through 6 give more detail about how **nroff** actually works with the input text to produce its output. Section 7 describes command line options available when calling **nroff**. A final Summary gives a brief overview of **nroff** commands.

2. The ms Macro Package

nroff is the text formatting program for the COHERENT system. Its input consists of text lines with interspersed command lines to control the processing. Its single most outstanding feature is its flexibility: the user has the ability to control line length, page offset, page length, paragraph format, beginning and end of page format, and so on. When you create your input for **nroff**, you are really writing a program telling it what to do with your text.

Fortunately, another feature of **nroff** makes it easier to learn to prepare input for it. A sequence of basic commands can be given a new command name; the sequence is called a *macro*. Whenever you want the sequence performed, you merely insert a reference to the macro. For example, you might group together the commands to format a paragraph under the name **PP**. Rather than retype the same sequence of commands each time you want to begin a paragraph, all you need to do is to insert the command line **.PP** before the start of a paragraph.

If you use **nroff** without the **ms** macros, you must devise your own ways to implement paragraph formatting and numbering pages. **nroff** does *not* do such things for you automatically. However, if you use it in the form **nroff --ms**, then **nroff** automatically includes the manuscript of macros described in this section with your text. These macros take care of setting line length and page length, numbering pages, formatting paragraphs, and so on. You do not need to know which basic commands are used in the macros; you only need to know the names of the macros and what they do, so that you may use them appropriately.

The only disadvantage of using the **ms** macro package is the very fact that it makes many formatting decisions automatically for you; you give up much of **nroff**'s flexibility. But this is a small price to pay for the convenience of the **ms** commands. to use **nroff** in its basic form. Also, learning to use the **ms** package first is a good way to become accustomed to preparing input for **nroff**, so the features of the basic program will not seem so alien if you eventually choose to work with them.

Section 5 of this tutorial describes the internal operation of macros in detail. However, you do *not* need to understand this in order to use the **ms** macro package. In general, it is *not* advisable to try to alter the macros in an existing package such as the **ms** macros. If you are sufficiently well acquainted with **nroff**, it is probably better to write your own macro package than to tamper with an existing one.

Text and Commands

nroff input includes both *text* and *commands*. The commands control the processing of the text. **nroff** distinguishes between text and commands by looking at the first non-space character of each input line. If the character is a period '.' or an apostrophe ''', the line is a command; otherwise, it is text.

To become accustomed to using **nroff**, enter some text into a file. Create a file **script.r** containing the following text, or containing your own text if you prefer:

```
   London.  Michaelmas Term lately over,
     and the Lord Chancellor sitting in
    Lincoln's Inn Hall.  Implacable November weather.
      As much mud in the streets, as if the waters
      had but newly retired from the face of the
      earth, and it would not be wonderful to meet
     a Megalosaurus, forty feet long or so, waddling
       like an elephantine lizard up Holborn Hill.
```

This file contains no commands; every line is a text line. Process the file with the command

```
nroff script.r
```

If you are working at a CRT terminal, pipe the output into **scat** so that it will not rush past you:

```
nroff script.r | scat
```

The result of processing the above text with **nroff** will look like this:

```
London.   Michaelmas Term lately over, and the
Lord Chancellor sitting in Lincoln's Inn Hall.
Implacable November weather.   As much mud in the
streets, as if the waters had but newly retired
from the face of the earth, and it would not be
wonderful to meet a Megalosaurus, forty feet long
or so, waddling like an elephantine lizard up
Holborn Hill.
```

When you try this example, the spacing will be a little different; spacing for examples in this tutorial is adjusted to indent the output within the rest of the tutorial text. You should notice several things about the output. **nroff** automatically adjusts the spacing between words to keep a strict right margin, even though the input text contains a ragged right margin. Each output line contains 65 characters and each output page contains 66 lines. This is called *justification*.

Now try processing **script.r** again, this time with the **ms** macro package. Type:

```
nroff -ms script.r
```

or

```
nroff -ms script.r | scat
```

By examining the output, you will see that **nroff** again adjusts the spacing to keep a strict right margin. **nroff** indents each output line with 10 leading spaces, followed by 65 characters. Each output page contains 66 lines, but **nroff – ms** leaves blank lines at the top of the page and puts the page number in a blank space at the bottom of the page.

The output of **nroff** is really just a sequence of characters. It is useful, though, to think of the output as being printed at ten characters per inch (Pica or 10 pitch spacing) and six lines per inch. Many output devices correspond to this spacing. With these assumptions, each page of output from **nroff – ms** fits on an 8 1/2 by 11 inch page, with an inch of blank space at the top, at the bottom, and on each side. Section 5 of this tutorial discusses units of measurement in more detail.

As the example demonstrates, **nroff** adjusts spacing between words to keep a strict right margin. When you type in the text, don't worry about the right margin. You should, however, keep a strict left margin. When **nroff** encounters a line of text that begins with blank spaces, it prints out the line it is currently processing and then assembles another line beginning with the blank spaces. This is called breaking the line.

In addition, you should not hyphenate words, with the start of a word on one line and the remainder on the following line. If you do, **nroff** treats the parts of the word as two separate "words" (the first ending with the hyphen character), rather than joining them as you intend.

If you have a line of text beginning with a period or an apostrophe, **nroff** normally interprets it as a command line. To prevent this, place the symbols '\&' at the beginning of the line before the period or apostrophe.

In the remainder of this tutorial, you will learn how to use commands in input text to change the appearance of the output. You can make the output lines longer or shorter or change the page format, by using the appropriate commands.

The name of each basic **nroff** command consists of two lower-case letters. Some commands include additional information on the command line, separated from the command name by a space. For example, sp is the command to leave vertical space between output lines. The command line

```
        .sp
```

leaves a single space, while

```
        .sp2
```

leaves two spaces. The information following the command name on the command line is called an *argument* or a *field*.

The name of each macro defined in the ms macro package usually consists of one or two upper-case letters. For example, **PP** is the name of the macro that begins a new paragraph. The remainder of this section describes some ms macros in detail.

Paragraphs

Every time you want to begin a new paragraph, use the *paragraph* command **PP**; that is, place the command line **.PP** in the text. For example,

```
        .PP
        It is a truth universally acknowledged,
        that a single man in possession of a good fortune,
        must be in want of a wife.
        .PP
        However little known the feelings or views of such
        a man may be on first entering a neighborhood, the
        truth is so well fixed in the minds of the surrounding
        families, that he is considered as the rightful
        property of some one or the other of their daughters.
```

When you process this text with **nroff** – **ms**, the result will look like the following:

```
             It is a truth universally acknowledged, that
        a  single man  in possession  of a  good fortune,
        must be in want of a wife.

             However little  known the feelings  or views
        of such  a man may be on  first entering a neigh-
        borhood, the truth  is so well fixed in the minds
        of  the  surrounding families,  that  he is  con-
        sidered as  the rightful property of  some one or
        the other of their daughters.
```

As the output shows, the **PP** command inserts a blank line before beginning a new paragraph and indents the first line of the new paragraph by five spaces.

The ms macro package provides another paragraph format in addition to **PP**. The **IP** command creates an *indented paragraph*. **nroff** indents only the first line of each paragraph created by a **PP** command, but it indents every line in an indented paragraph. The command line

```
        .IP
```

skips a line and then begins every line in the following paragraph with an indent of five spaces. For example,

```
.IP
This is an indented paragraph.
All the lines are indented by
the same amount.
.PP
This is a normal paragraph.
nroff indents the first line
but does not indent the following lines.
```

gives the output

```
          This  is  an  indented paragraph.   All  the
          lines are indented by the same amount.

          This is  a normal paragraph.   nroff indents
     the first line  but does not indent the following
     lines.
```

Several important variants to the basic **.IP** command line are available. You can add two additional arguments to the command line; each argument should be preceded by a space. **nroff** interprets the first argument after the **IP** as a *tag* to the paragraph. It interprets the second argument as the amount of indentation you want. For example,

```
.IP A. 8
This is the first line of text.
nroff indents the following lines by the same
amount as the first.
The indent is eight spaces.
The paragraph includes a tag in the indent.
```

produces

```
A.      This  is the first  line of  text.  nroff
        indents the  following lines by  the same
        amount as the first.  The indent is eight
        spaces.  The paragraph  includes a tag in
        the indent.
```

You should make sure the indent leaves enough spaces for the tag. If the tag contains blank spaces, enclose it in double quotes:

```
.IP "King Lear:" 16
Is man no more than this?
Consider him well.
Thou owest the worm no silk,
the beast no hide,
the sheep no wool,
the cat no perfume...
Unaccomodated man is no more
but such a poor, bare, forked
animal as thou art.
```

produces

```
King Lear:      Is  man no more  than this?  Con-
                sider him  well.  Thou  owest the
                worm no silk,  the beast no hide,
                the  sheep no  wool,  the cat  no
                perfume...  Unaccomodated  man is
                no  more but  such a  poor, bare,
                forked animal as thou art.
```

As this example shows, this form of the **IP** command might be useful to format the script for a play.

If you do not want a tag but merely wish to set the indent to something other than the automatic five spaces, then use a pair of double quotes with nothing between them for the first field:

```
.IP "" 8
```

If you forget the quotes, you will not get what you expect; **nroff** interprets 8 as a tag and uses the normal indent of five spaces.

Once you set the amount of indentation, the new indent stays in effect until you change it again. For example, if you format a paragraph with

```
.IP "" 8
```

and follow it with another paragraph beginning with **.IP**, **nroff** also indents the second paragraph by eight spaces. The indent remains in effect until you explicitly change it, for example by beginning a paragraph with **.IP "" 6** (which resets the indent to six spaces).

Normally, **nroff** measures the paragraph indentation from the left margin. Another variation of **IP** makes it possible to measure the indentation of a new indented paragraph from the left-hand edge of a previous indented paragraph, thus producing a *relative indent*. To do this, enclose the new paragraph between the commands **RS** and **RE** (for relative indent start and relative indent end). For example,

```
.IP
I began to nod drowsily over the dim page;
my eye wandered from manuscript to print.
I saw a red ornamented title --
.RS
.IP
Seventy Times Seven, and the First of the
Seventy-First.  A Pious Discourse delivered
by the Reverend Jabes Branderham, in the
Chapel of Gimmerden Sough.
.RE
.IP
And while I was, half consciously, worrying
my brain to guess what Jabes Branderham would
make of his subject, I sank back in bed, and
fell asleep.
```

produces

```
     I began  to nod drowsily over  the dim page;
     my eye wandered from manuscript to print.  I
     saw a red ornamented title --

          Seventy Times Seven,  and the  First of
          the  Seventy-First.  A  Pious Discourse
          delivered by  the Reverend  Jabes Bran-
          derham,  in  the  Chapel  of  Gimmerden
          Sough.

     And while I  was, half consciously, worrying
     my  brain  to  guess what  Jabes  Branderham
     would make  of his  subject, I sank  back in
     bed, and fell asleep.
```

You can include any number of indented paragraphs between **RS** and **RE**; also, you may specify tags and different indents just as for ordinary indented paragraphs. You may even nest **RS** and **RE** pairs inside each other to produce multiple relative indents. Just remember that an **RS** must always be balanced by an **RE**. The following Hollywood scenario uses relative indents for levels of nested flashbacks.

```
.IP
In England during World War II, a captain tells the
story of his Free French bomber squadron.
.RS
.IP
In the early days of the war, a French ship picks up
five men adrift in a small boat.  One tells of their
life on Devil's Island.
.RS
.IP
A convict tells others of his past.
.RS
.IP
Publication of anti-Nazi material leads to arrest on
false charges.
.RE
.IP
The convicts escape to help France in the war.
.RE
.IP
When France surrenders, the crew overpowers pro-Vichy
officers and heads for England instead of Marseilles.
.RE
.IP
The captain concludes his story as the bombers return
from a mission.
```

This produces the following output.

```
In  England during World  War II,  a captain
tells  the story of  his Free  French bomber
squadron.

     In the early  days of the war, a French
     ship  picks  up five  men  adrift  in  a
     small boat.  One tells of their life on
     Devil's Island.

          A  convict  tells  others  of  his
          past.

               Publication    of   anti-Nazi
               material  leads to  arrest on
               false charges.

          The convicts escape to help France
          in the war.

     When France  surrenders, the crew over-
     powers pro-Vichy officers and heads for
     England instead of Marseilles.

The captain concludes his story as the bomb-
ers return from a mission.
```

If you build up successive layers of relative indentation with several **RS** commands, each **RE** peels away the current layer of indentation and places you in the previous one. To return to an even earlier level, your input must include the appropriate number of **RE** commands before you begin another paragraph.

A third type of paragraph is the *quoted* paragraph; this sets off a quotation from the surrounding text. It produces a paragraph which is indented both on the right side and on the left side. To produce such a paragraph, precede it with the **QS** command and follow it with the **QE** command. To break the quote into different sections, insert a blank line in the text before each line that you want to begin a new section. For example, try the following example.

```
Form of Tender of Rescue from Strange Young
Gentleman to Strange Young Lady at a Fire.
.QS
Although through the fiat of a cruel fate, I have
been debarred the gracious privilege of your
acquaintance, permit me, Miss [here insert name,
if known], the inestimable honor of offering you
the aid of a true and loyal arm against the fiery
doom which now o'ershadows you with its crimson
wing.  [This form to be memorized, and practiced
in private.]
.QE
Should she accept, the young gentleman should offer
his arm - bowing, and observing "Permit me" -
and so escort her to the fire escape and deposit
her in it.
```

This produces the output:

```
Form   of   Tender   of   Rescue from  Strange  Young
Gentleman to Strange Young Lady at a Fire.

     Although  through the  fiat of  a cruel
     fate, I have been debarred the gracious
     privilege of  your acquaintance, permit
     me, Miss [here  insert name, if known],
     the inestimable  honor of  offering you
     the aid of a true and loyal arm against
     the  fiery doom  which now  o'ershadows
     you with its  crimson wing.  [This form
     to  be   memorized,  and  practiced  in
     private.]

Should she accept, the young gentleman should of-
fer his arm - bowing, and observing "Permit me" -
and so escort  her to the fire escape and deposit
her in it.
```

Section Headings

The *section heading* command **SH** prints a heading or title. For example:

```
.SH
Section Headings
```

The heading may be more than one line long; consequently, you should follow a section heading by a **PP** or **IP** command. **nroff** leaves a blank line before the heading and prints the heading flush with the left margin in **boldface** type, as described below in the section on Fonts.

The *numbered heading* command **NH** produces consecutively numbered section headings. For example:

```
.NH
Guess What's Coming to Dinner?
.NH
Guess Why I Won't be There?
```

produces

```
1.  Guess What's Coming to Dinner?

2.  Guess Why I Won't be There?
```

if these are the first two **NH** commands in the input.

You can produce numbering for subsection headings by entering a number from 2 to 5 on the **NH** command line. The number indicates the level of section headings; **NH 2** numbers subsection headings, **NH 3** numbers subsubsection headings, and so on. For example:

```
.NH
Guess What's Coming to Dinner?
.NH 2
Guess What it Looks Like?
.NH 3
Teeth Like That Might Frighten the Children!
.NH 2
What Does it Eat?
.NH
Guess Why I Won't be There?
```

produces

```
1.  Guess What's Coming to Dinner?

1.1 Guess What it Looks Like?

1.1.1  Teeth Like That Might Frighten the Children!

1.2 What Does it Eat?

2.  Guess Why I Won't be There?
```

The number on the **NH** command line is *not* the number that appears in front of the heading; instead, the number controls how many "parts" the number that appears contains. For example, **NH 3** produces a three-part number like 2.5.3, while **NH 4** produces a four-part number like 7.4.5.2.

You can reset the entire numbering scheme by using the command **NH 0**; for example,

```
.NH 0
Nancy and Ron's Favorite Recipes
```

produces

```
1.  Nancy and Ron's Favorite Recipes
```

with numbering starting at 1.

Title Page

If you want your output to begin with a title page, begin the input with the following.

```
.TL
Title of document (may be more than one line)
.AU
Name(s) of author(s) (may be more than one line)
.AI
Institution(s) of author(s)
.AB
Abstract (line length 5.5 inches)
.AE
```

The **TL** command indicates the *title*, the **AU** command indicates the *author*, the **AI** command indicates the *author's institution*, and the **AB** command precedes the *abstract*. The **AE** command, for abstract end, marks the end of the abstract. If you do not want some of these headings to appear, simply omit the relevant command lines. The actual text of the document should begin immediately following the **AE** command line. You must begin the text with a command such as **PP** or **SH**.

The following is a typical example of a title page:

```
.TL
Doctor Smith Meets the Green Lady
.AU
P. R. Soba
.AI
The Psychedelic Haberdashers' Club
.AB
Driven frantic by the metallic sheen and
high-pitched whine of the Green Lady's siren song,
Doctor Smith prepares to abandon the safety of the
spaceship and risk all on an interstellar fling.
.AE
.PP
(Text of the first paragraph begins here...)
```

If you try this example, you see in the output that the text begins on the same page as the title information. You may or may not want this to happen. If you do not, one solution is to insert two additional commands between the AE command and the first **PP** (or whatever your first formatting command might be):

```
        .PP
        .bp
```

The **PP** followed by the *begin page* command **bp** puts a dummy paragraph on the title page and then begins a new page. The reason bp alone will not work lies in the construction of the title page macros; the dummy paragraph forces **nroff** to print title page information before beginning a new page.

Headers and Footers

The *header* macro controls the format of the top of each page. It automatically skips an inch of space. The *footer* macro controls the format of the bottom of each page. It reserves a one inch vertical block of space at the bottom, blank except for the page number (roughly in the center of the block).

It is easy to print a title in either or both of the header and footer areas in the following way. Each title is a three-part title. **nroff** prints the first part on the left-hand side of the page, the second part in the middle, and the third part on the right-hand side of the page. The parts of the header title are:

```
        LT: left-hand part
        CT: center part
        RT: right-hand part
```

The parts of the footer title are:

```
        LF: left-hand part
        CF: center part
        RF: right-hand part
```

These parts are called *strings*. Section 4 of this tutorial describes strings in detail. Normally, each of these strings is empty, except for **CF**, which gives the current page number. Therefore, the header macro prints nothing, while the footer macro prints the page number in the center of the block of space at the bottom of each page. To define one of these strings, do the following:

```
        .ds LT "nroff Text Processor Tutorial"
```

After you define **LT** in this fashion, **nroff** prints **nroff Text Processor Tutorial** at the top of each page on the left-hand side. If you want the date to appear on the right-hand side at the top, use:

```
        .ds RT "9/10/82"
```

You can use the same procedure to define the strings in the footer title. If you want something other than the page number to appear in the position allocated to **CF**, use the ds command to redefine **CF**. If you do not want anything to appear there, use

```
        .ds CF ""
```

Wherever you want the current page number to appear in the header or footer, use the symbol '%'. For example, if you want the page number to appear in the upper right-hand corner of each page, use

```
        .ds RT "Page %"
```

The resulting numberings look like 'Page 7', 'Page 10', and so on.

Fonts

Most **nroff** output consists of normal characters, called Roman. In addition, **nroff** lets you emphasize words with **boldface** and *italic* characters. Each of the three type styles—Roman, boldface and italic—is called a *font*, in keeping with typesetting terminology.

nroff represents each boldface and italic character by a special three-character output sequence. It represents a boldface character c by the character 'c', followed by the backspace character **<ctrl-H>**, followed by another 'c'. It represents an italic character *c* by the underbar character '_', followed by the backspace character **<ctrl-H>**, followed by 'c'.

Because of these special representations, the appearance of **nroff** boldface and italic fonts depends on the device on which you see the output. On a CRT terminal, the **<ctrl-H>** backspaces the cursor, and the third character of each sequence replaces the first; therefore, boldface and italic characters look the same as Roman characters. On a hard-copy terminal, boldface characters are

Your text should include each footnote at the point in your text where the reference to it occurs; **nroff** sees to it that the footnote appears at the bottom of the page. For example:

```
Text preceding the appearance of the footnote...
The President, Mr. Wally Wiggin, is a real dufus.*
.FS
*dufus: cretin, mezzo cullello, etc.
.FE
Normal text resumes again here...
```

The definition of *dufus* appears at the bottom of the page.

Displays and Keeps

A *display* is a portion of text that you wish to appear in the output exactly as it appears in the input, such as a graph or a table. If you do not give **nroff** instructions to the contrary, it will alter the spacings between elements in your display, thus destroying its appearance. Therefore, display commands are available to tell **nroff** not to alter spacings between elements. Also, **nroff** makes certain not to split your display between two pages.

Enclose the text of your display between the *display start* command **DS** and the *display end* command **DE**.

```
.DS
The text of the display goes here,
exactly as you want it to
appear in the output.
.DE
```

There are three variants to the basic **DS** command.

.DS C The *display start centered* command centers each line of your display. Since **nroff** centers each line individually, both right and left margins are ragged.

.DS B The *display start block-centered* command considers the entire display at once and centers it. You can think of this as simply shifting the display to the right or to the left by an appropriate amount.

.DS I The *display start indented* command indents the entire display by 1/2 inch.

If your display is longer than one page, do not use **DS** or any of its variants. Instead, begin the display with one of the following.

.CD The *centered display* command centers each line of the display.

.BD The *block-centered display* command considers the entire display as a block and centers it.

.LD The *left display* command performs no indenting or centering, but simply begins each line at the left margin.

.ID The *indented display* command indents each line by 1/2 inch.

If you begin the display with one of these three commands, do not end the display with **DE**; simply use **IP** or **SH** or whatever command is needed at that point. You can also end the display with **PP** and then continue.

You must be conscious of one important fact when you use display commands. The length of normal output lines is 6.5 inches. If the display contains lines longer than this, **nroff** simply prints them as they are, since that is what it does in a display. Consequently, long lines may extend into the right margin. If a line is too long to fit on one output line, **nroff** extends it as far as possible to the right and then continues it on another line. The visual effect can be quite unpleasant. The only restriction on what you can safely put in a display, then, is that lines should be shorter than 6.5 inches. If you are using an indented display, lines should be shorter than 6 inches.

The idea behind the *keep* is similar to that of the display: you put text between the *keep start* command **KS** and the *keep end* command **KE** when you do not want it split across a page boundary. If you put a block of text between these commands that turns out to be longer than a page in length, it begins on a new page but necessarily spills over at some point onto another page.

The major difference between the keep and the display is that normal processing occurs in the keep: **nroff** adjusts spacings between words, performs hyphenation, and so on, just as it normally does.

Other Commands

Several basic **nroff** commands are safe to use in conjunction with the **ms** macro package. The command **sp** *N* skips *N* lines on the output page; for example, **sp 4** skips 4 lines. The *begin page* command **bp** does just that: when **nroff** encounters the command, it stops printing on the current page (no matter how far down the page it is) and begins a new page.

Another command is the *break* command **br**. To understand what this command does, you need to understand a bit about how **nroff** assembles output lines. Imagine that **nroff** has a row of slots 6.5 inches long; **nroff** fills each slot with words and blank spaces. **nroff** takes a word at a time from the input and tries to add it into the row of slots. When it grabs a word that does not fit, it either rejects it entirely (for the moment) or hyphenates it and takes part of it. Then it adjusts spacing between words to completely fill out the row of slots. Finally, **nroff** prints the line.

If you interpose the command **br** at some point in your input text, **nroff** prints whatever words it has collected at that point, whether or not they form a complete line. The break command is actually incorporated into many of the **ms** macros, such as **PP** and **IP**; this explains why the last line of a paragraph is often shorter than a normal line of output.

The remaining sections of this tutorial provide more information about these basic commands and about other **nroff** commands.

3. Basic Commands

Your most elementary concern about the appearance of your processed text is the position of the text on the output page. This involves control of line length, left and right margins, page offset (how far from the left edge of the page each line begins), page length, and so on. Control of these formatting functions is quite easy with the appropriate **nroff** commands.

The *line length* command **ll** controls the line length, while the *page offset* command **po** controls the page offset. If you are writing an **nroff** script, you should include these commands before the beginning of your text, so **nroff** puts them into effect immediately. For example, you could demand a line length of 3 inches and a page offset of 2 inches:

```
.ll 3i          \" set line length
.po 2i          \" set page offset
He bounded up the stairs two at a time.
At the top of the first flight he
grabbed the railing to catapult
himself around the corner of the landing.
He nearly hurtled right into her room.
The door was open; she was standing there totally...
```

nroff ignores comments following the symbols '\"' in the input; comments are for the benefit of someone who reads your **nroff** script. Judiciously placed comments can make a complicated script much easier to understand.

You may have noticed the specification of the line length and page offset in inches in this example. As noted in Section 1 of this tutorial, **nroff** output is actually just a sequence of characters, but it is often convenient to think of the output as being "printed" at ten characters per inch (Pica or 10 pitch spacing) and six lines per inch. The specifications in inches make this assumption. The line length specification

```
ll 3i
```

produces output lines containing 30 characters each; on many output devices, the output lines are indeed 3 inches long. Section 5 below discusses **nroff** units of measurement in detail.

Breaks

It is important for you to understand how **nroff** constructs finished lines of output. Suppose you tell **nroff** you want output lines 5 inches long. **nroff** takes a word at a time from the input and attempts to add it to the buffer.

What happens when the line is almost full, with room for part of the next word but not for the whole word? Unless you specify otherwise, **nroff** tries to maintain strict right and left margins. It adjusts the number of blank spaces between words and may hyphenate the next word, so the sequence of characters and blank spaces in its buffer is exactly as long as you want.

You know now that **nroff** normally gathers words in its buffer and adjusts spacing until the buffer is full. The *break* command **br** tells **nroff** to print whatever is in the buffer, even if it is not a complete line. When **nroff** encounters the break command, it might still be looking for words to fill the buffer. The break forces **nroff** to print the buffer without adjusting the spacing between words, so the end of the line probably will not be flush with the right hand margin. You should experiment by inserting a **br** or two in the text of a sample script to see what happens.

The idea of a break might seem strange at first, but you are no doubt familiar with a simple example: the end of a paragraph. You do not want the start of a new paragraph to be on the same line as the end of the previous paragraph. You want to print the end of the previous paragraph whether or not it fills a complete line. And you want to begin the new paragraph on a new line. As you will learn later, some **nroff** commands cause breaks automatically; you should be aware of this when you use them.

Fill and Adjust Modes

Two basic terms describe how **nroff** processes your input to create its output: *filling* and *adjusting or justifying*. Unless you specifically tell it not to, **nroff** operates in fill mode and adjust mode.

The *fill* command **fi** tells **nroff** to use fill mode; the *no fill* command **nf** tells it to use no-fill mode. Similarly, the *adjust* command **ad** tells **nroff** to use adjust mode, while the *no adjust* command **na** tells it to use no-adjust mode.

As mentioned above, **nroff** is initially in both fill mode and adjust mode, so it is not necessary to begin your script with **fi** and **ad** if you want filling and adjusting. These commands are necessary to restore the modes after you change them with the **nf** and **na** commands.

If you use **nf** to turn off fill mode, **nroff** no longer tries to fill lines to a fixed line length. It prints each line of input text exactly as received. However, if you use a large page offset, a sufficiently long line of text could reach the right-hand edge of the page when **nroff** prints it. If the input line cannot fit on one line, **nroff** continues printing it on the next line with no page offset.

In adjust mode, **nroff** adjusts the spacing between words to fill lines of text, as described above. When **nroff** is in no-fill mode, it is automatically in no-adjust mode: with no fixed line length, there is no need to adjust spacing. Moral: you can fill without adjusting but you cannot adjust without filling.

If you request filling but no adjusting, **nroff** fills its output line with input but does not adjust spacing between words; it does not try to keep an even right margin. Since each output line must fit into the buffer, it is either shorter than the line length you specify or exactly as long.

The **ad** command includes several options. If you use the command **ad** without an argument, **nroff** keeps strict left and right margins. **nroff** uses this mode by default. **ad l** keeps the left margin only; **ad r** keeps the right margin only; **ad b** or **ad n** keeps both margins. Finally, **ad c** centers output lines while keeping their lengths less than or equal to the specified length.

You should remember that **nroff** ignores adjust requests if you are in no-fill mode. If **nroff** is in fill mode and you request any variety of adjustment, it adjusts accordingly until you give either a no-fill or a no-adjust command. If you give a no-fill command, only a fill command restores adjustment; no plea for a different kind of adjustment works in no-fill mode. If you give a no-adjust command, only a request for some type of adjustment restores adjust mode.

As an example, enter the following script and process it with **nroff**. Edit the text if you wish, but keep the same sequence of commands and try to understand how each command affects the output.

```
A typical night in the bag house.  A Doors album
was playing so loudly that the speaker covers
were trying to leap away to safety.
The dying sun, weakened by its flight through the
five foot weeds outside, spent itself in etching
dim jungle scenes on the living room walls.
.sp
.na             \"no adjust
Steve tapped his foot as he stirred the huge
vat of chili that seethed on the stove.  Whole
onions occasionally bobbed to the surface and,
like bizarre bolus-shaped whales, gave off jets
of steam before sinking back down into the brine.
.sp
.ad r           \"right-adjust
Steve dumped the last half of the can of
cayenne pepper into the pot and sat down
to finish reading his Heavy Metal.
A vague desire was forming
in his mind... but for what?
Then he knew.  He went to the freezer,
shoved aside the bags of scraps and fat
from last Sunday's pork roast, and pulled
out the transuranium strawberry cake.
He got the hack saw and sawed off a slice.
.sp
.nf             \"no-fill
Ratfink sauntered into the kitchen and began
rubbing his left cheek against Steve's leg.
        "Hello cat."
Steve leaned back against the wall
and nearly glued himself to the piece of french
toast that Paul had stuck there.
.sp
.fi             \"fill
The sickening yowls of a cat in fear for its
life rent the air.  The monster Fluff scuttled
across the bag room floor as fast as his
considerable flab and hundreds of scattered grocery
bags would permit.  Paul followed close behind,
```

```
armed with his favorite weapon: a squirt gun.
Only panic made Fluff's desperate bid
for a window sill successful.
He cowered there and emitted feeble little bleats
as blast after blast of cold water pelted his fur.
```

When you process this input with **nroff**, your output should look like this:

```
A typical night  in the bag house.  A Doors album
was  playing so  loudly that  the  speaker covers
were trying  to leap  away to safety.   The dying
sun, weakened by its flight through the five foot
weeds outside, spent itself in etching dim jungle
scenes on the living room walls.

Steve tapped his foot as he stirred the huge vat
of chili that seethed on the stove.  Whole onions
occasionally bobbed to the surface and, like
bizarre bolus-shaped whales, gave off jets of
steam before sinking back down into the brine.

 Steve dumped the last half of the can of cayenne
       pepper into the pot and sat down to finish
reading his Heavy Metal.  A vague desire was for-
ming in his mind... but for what?  Then he knew.
 He went to the freezer, shoved aside the bags of
scraps and fat from last Sunday's pork roast, and
 pulled out the transuranium strawberry cake.  He
got the hack saw and sawed off a slice.

Ratfink sauntered into the kitchen and began
rubbing his left cheek against Steve's leg.
          "Hello cat."
Steve leaned back against the wall
and nearly glued himself to the piece of french
toast that Paul had stuck there.

The sickening yowls of a cat in fear for its life
 rent the air.  The monster Fluff scuttled across
   the bag room floor as fast as his considerable
flab and hundreds of scattered grocery bags would
  permit.  Paul followed close behind, armed with
   his favorite weapon: a squirt gun.  Only panic
made Fluff's desperate bid for a window sill suc-
     cessful.  He cowered there and emitted feeble
 little bleats as blast after blast of cold water
pelted his fur.
```

Since the beginning of the input text contains no fill or adjust specification, by default **nroff** fills and adjusts the first paragraph. After the **na** command, it fills but does not adjust the second paragraph. After the **ad r** command, it fills and right adjusts the third paragraph. After the **nf** command, it neither fills nor adjusts the fourth paragraph. Finally, after the **fi** command, it fills the fifth paragraph and uses the **ad r** adjust option which was in effect previously.

Sometimes **nroff** is not able to adjust in adjust mode. For example, suppose you specify a one-inch line length. A seven-letter or eight-letter word takes up the greater portion of a line. Your text could include a long word followed by a word which cannot fit on a line with the long word. Rather than printing a line longer than you specified, **nroff** begins a new line with the second word. The right margin is uneven, as though adjustment were not taking place.

Paragraphs

What happens if you copy text from several pages of a book into a file without adding any formatting commands and then process the file with **nroff**? There is no page offset; in the absence of any specification, **nroff** assumes a page offset of zero, so processed lines begin at the left margin. The processed lines are 6.5 inches long; this is the default value **nroff** assumes.

More interesting things happen with paragraphs. Suppose you skip one line between paragraphs and begin each paragraph with an indent of five spaces. A blank line in the input text causes a break (as discussed earlier) and then causes **nroff** to print a blank line. The last line of each paragraph is not adjusted and probably not flush with the right hand margin. There is a blank line before the next paragraph.

Initial blank spaces in a line of input also cause a break. In this example, the breaks caused by initial blank spaces at the beginning of each paragraph really do nothing, since the preceding blank line forces out the last line of the preceding paragraph. **nroff** always considers initial blank spaces in a line significant and preserves them in the output.

Copy the following example and then run it through **nroff**:

```
     Here is a little text so you can see
whether nroff will ignore the initial
indentation
          in this very very long sentence.
Here is a little bit more text.

     And here is something to mimic
the beginning of a new paragraph.
```

The output looks like this:

```
     Here is a little text so you can see whether
nroff will ignore the initial indentation
          in this very  very long sentence.  Here is
a little bit more text.

     And here is something to mimic the beginning
of a new paragraph.
```

Instead of leaving a blank line in the text, you could use the *space* command **sp 1**, which causes a break and inserts one blank line in the output. Similarly, **sp 5** causes a break and inserts 5 blank lines in the output. Edit the example and replace the blank line by the command line

```
.sp 1
```

to see that it has the same effect. You can also use the command sp; **nroff** assumes you want one space if you omit the argument.

Most **nroff** input consists of many paragraphs containing text. You probably want each paragraph to have the same format in the output. Rather than formatting each paragraph explicitly as in this example, you can use the *macro* facility of **nroff** to define a sequence of commands to format a paragraph. Macros are the subject of the next section of this tutorial.

Centering

The *center* command ce centers a line or several lines of text. For example, you can center a two-line heading as follows:

```
.ce 2
Heading Printed
In Center of Page
```

If you use the ce command with no argument, **nroff** assumes a default argument of 1 and centers the next line of input.

Tabs

If your **nroff** input includes tables of information, you may find it convenient to use tabs to separate items in a line of the table. **nroff** recognizes the **<tab>** character **<ctrl-I>** and expands tabs into spaces. If you use tabs to format a table, remember to use no-fill mode; otherwise, **nroff** tries to fill and adjust your output lines.

By default, **nroff** uses tab stops set eight characters apart on its output line, at positions 8, 16, 24, and so on. You can use the *tab* command **ta** to change the positions of the tab stops. For example,

```
.ta 10 20 30 40 50 60
```

sets tab stops ten characters apart rather than eight.

You can use the *tab character* command tc to change the character **nroff** prints between its current position and the next tab stop. For example,

```
.ta 9 19 29 39
.tc *
.nf
<tab>1<tab>2<tab>3<tab>4
```

produces the output

```
*********1*********2*********3*********4
```

Pages

The *begin page* command **bp** causes a break and forces **nroff** to the next output page. By default, **nroff** assumes a page length of 11 inches (66 lines). You can change the page length with the *page length* command **pl**. For example,

```
.pl 21
```

specifies a two-inch page length.

What about the format of a page? Does **nroff** automatically keep top and bottom page margins, number pages, or do anything similar? The answer, sadly, is no. **nroff** just keeps track of the current output page number and the current line number on the current output page.

Offhand, this does not seem to do much good, since you do not know beforehand the effects of filling and adjustment on your input. You might wonder whether you could have **nroff** execute a set of commands whenever it reaches a certain position on the page. This would solve the problem of producing top and bottom margins and would not require you to know where to place the commands. In fact you can, by using *traps*. The next section of this tutorial describes traps and how to use them to format a page.

4. Macros

To become familiar with the idea of a *macro*, consider the problem of formatting paragraphs. **nroff** preserves blank lines and initial indents, so one way you could force **nroff** to break your text into paragraphs would be to format your input yourself: put a blank line after the last line of each paragraph and then indent the first line of the next paragraph.

Another way to achieve the same effect would be to put the three commands

```
        .br             \" break
        .sp             \" skip a line
        .ti 5           \" indent next line 5 spaces
```

between the end of each paragraph and the start of the next paragraph. You should recognize the first two commands: **br** causes a break, so **nroff** prints the last line of the previous paragraph even though it might not be a complete line; sp skips a line before the next paragraph begins. The third command is the *temporary indent* command **ti**; the number indicates how many spaces to indent the next output line. Since this command indents the first line of the paragraph, you do not need to indent the line in your file. For example:

```
.ll 31          \" line length
.po 31          \" page offset
.ti 5           \" indent next line
John and Lizzie Wilson moved to Stuart,
Florida from Boston five years ago.
.br             \" break
.sp             \" skip a line
.ti 5           \" indent next line
They now live in a trailer surrounded
by a cute little white picket fence.
Lizzie spends her time tending the tiny
garden she has planted behind the trailer.
A plastic pink flamingo stalks
the gravel patch between the two
halves of the garden.
.br
.sp
.ti 5
John sits in the sun, his eyes shielded
by the visor of his Red Sox cap,
playing with his fishing lures.
```

Suppose your file is very long, with hundreds of paragraphs. Every time you want to begin a paragraph, you need to include the same set of commands in the text. It would save considerable agony if you could create a *name* for this set of commands. Then you could simply put the name in your text whenever you want **nroff** to perform the commands, rather than repeating the commands again and again.

As you probably have guessed by now, you can do just that; the set of commands is called a *macro*. Here is the John and Lizzie story again, this time with a paragraph macro called **PP** that takes care of formatting each paragraph.

```
.ll 31          \" line length
.po 31          \" page offset
.de PP          \" paragraph macro
.br
.sp
.ti 5
..              \" end of macro definition
.PP
John and Lizzie Wilson moved to Stuart,
Florida from Boston five years ago.
.PP
They now live in a trailer surrounded
by a cute little white picket fence.
Lizzie spends her time tending the tiny
garden she has planted behind the trailer.
A plastic pink flamingo stalks
the gravel patch between the two halves
of the garden.
.PP
John sits in the sun, his eyes shielded
by the visor of his Red Sox cap,
playing with his fishing lures.
```

Before you can use a macro, you must define it. The definition associates the *macro name* with the *definition* you supply. The *define* command **de** defines a macro; the name which follows the **de** specifies the macro name. The macro name may be either one or two characters long. The above example defines the macro **PP**:

```
.de PP          \" paragraph macro
```

Each **nroff** command you have seen previously consists of a single command line. The **de** command itself is also a single command line, but you must follow it with other lines which contain the definition of the macro. The definition ends with a line containing two periods, "..". The command lines between the **de** command and the two periods are sometimes called the *body* of the macro. You cannot nest one macro definition inside another.

You use a macro by calling it like any **nroff** command: you precede its name with a period or an apostrophe on a command line. When you call a macro, it has precisely the same effect as placing the body of the macro at that point in the text. It is much easier to include the one-line invocation of the macro each time you need it rather than to repeat the set of commands each time. Each time you want **nroff** to make a new paragraph, you simply place the command line

```
.PP
```

between the line that ends a paragraph and the line that begins the next paragraph.

One of the most important things to remember about macros is that you define them yourself using basic **nroff** commands. A macro may contain whatever basic commands you care to put in it; the commands you use depend completely upon what you want the macro to do.

Macros may contain text as well as commands. For example, if you have a block of text repeated many times throughout your script which you want to format differently from the rest of the text, you could create a macro that is essentially a mini-script containing the text and the commands to format it. Whenever you want the text to appear in the output, you merely call the macro in your input. The point again is the generality of macros; you create them to suit your needs.

Traps

Now consider the problem of formatting the beginning and ending of each page of output. You could define what are traditionally called *header* and *footer* macros containing the commands you want performed at the top and bottom of each page. But you cannot possibly know where to call these macros in the input text, since you cannot know the vertical position of a given line on the output page before processing it with **nroff**. You can solve this problem by the use of *traps*.

nroff keeps track of its vertical position on each output page. You can set a *trap* for **nroff** to execute a specified macro at a given vertical position on every page. When a line of output reaches or extends past that position on the page, **nroff** automatically executes the commands in the macro before any more processing takes place.

The *when* command **wh** sets a trap for a macro, specifying the name of the macro and the vertical position of the trap. For example, you probably want **nroff** to call your header macro **hd** (whatever commands it might contain) at the very top of each page. The command

```
.wh 0 hd        \" set header trap
```

sets a trap for the macro **hd** at vertical position 0 (the very top of the page) of every output page. To set a trap for your footer macro **fo** (whatever commands it might contain) one inch from the bottom of each page, use the command

```
.wh -1i fo      \" set footer trap
```

The negative number tells **nroff** to measure distance from the bottom of the page rather than from the top; the **i** is an abbreviation for inches. **nroff** recognizes various units of measurement, as described in more detail in Section 5 below. It is safe to always include the abbreviation for the unit, as in this example; if you leave the unit indicator off of a measurement in a command, **nroff** might not measure in the units you expect. For example, if you write **wh -1 fo**, **nroff** interprets the unit of measurement to be the height of one line of print, just as with the **sp** command. The abbreviation for this unit of vertical measurement is **v**.

Suppose you want to design the output page by defining the header and footer macros. A simple header macro just skips an inch of space at the top of each page; a simple footer macro forces printing to stop an inch from the bottom of each page and prints the page number. **nroff** does not print page numbers automatically, but it does automatically keep track of what output page it is on. It stores the page number internally in a *register* you can access with the symbol '%'. Section 5 below gives more information about registers and how to use them.

A simple footer macro that prints the page number is:

```
.de fo          \" footer macro
'sp 4v          \" skip four lines (no break)
.tl ''- % -''   \" print page number
'bp             \" new page
..
```

There are several points of interest raised by this macro.

First, notice that some commands are preceded with an apostrophe rather than with a period. This supresses the break these commands normally cause. This is desirable, because **nroff** takes a word at a time from the input text and places it in the output line until a word does not fit. It then either hyphenates the word or leaves it out of the line entirely; in either case, it adjusts the spacings between words in the line and prints the line. **nroff** still has part or all of the last word left to begin filling the buffer. If the output line triggers a trap for a macro, **nroff** executes the commands in the macro before it accepts any more input text. It is still holding the portion of the word that did not fit into the output line. If any of the commands in the macro cause a break, **nroff** prints the next word on a new output line.

You might run into problems, then, if you naively define your header macro as follows:

```
.de hd          \" header macro
.sp 1i          \" skip an inch (break)
..
```

You want this to leave a blank space of one inch at the top of each page. But the sp command causes a break, so if a word were left over from producing the last line on the preceding page, **nroff** would print it at the very top of the next page. The visual effect would be quite unpleasant.

But if you use 'sp instead of .sp in the macro, **nroff** suppresses the break and does not print the partial word until after it performs the macro commands. Likewise for the footer macro; you do not want anything unplanned to be printed in the blank space at the bottom of the page. You should always be conscious of these considerations when you use commands that cause breaks.

Second, new item in the example is the *title* command **tl**, which prints a three-part title. A three-part title contains a left part (aligned to the left margin of the page), a center part (centered), and a right part (aligned to the right margin). The command name **tl** is followed by a field containing four single quote characters. **nroff** prints the characters you supply between the first two quotes as the left part of the title line, what you supply between the second and third quotes as the center part, and what you supply between

the third and fourth quotes as the right part of the three-part title. If you do not want **nroff** to print anything in one of these positions, simply put nothing between the appropriate pair of quotes. In the above example, the **tl** command tells **nroff** to print nothing on the ends of the title line and the page number in the center. If you want the quote character to appear in a part of the title, precede it with the backslash character '\'.

The length of the title line is independent of the length of normal output lines, so you must set it with the *length of title* command **lt** unless you want **nroff** to use the default title length of 6.5 inches. For example, to set the length of the title to five inches, use the command

```
.lt 5i
```

In light of all you now know, you should give the John and Lizzie story the treatment it truly deserves.

```
.ll 3i          \" set line length
.po 2i
.pl 3i
.wh 0 hd        \" header trap
.wh -1i fo      \" footer trap
.de hd          \" header
'sp 1i
..
.de fo          \" footer
'sp 2
.tl ''- % -''
'bp
..
.de PP          \" paragraph macro
.sp 1
.ti 5
..
.PP
John and Lizzie Wilson moved to Stuart,
Florida from Boston five years ago.
.PP
They now live in a trailer surrounded
by a cute little white picket fence.
Lizzie spends her time tending the tiny
garden she has planted behind the trailer.
A plastic pink flamingo stalks
the gravel patch between the two halves
of the garden.
.PP
John sits in the sun, his eyes shielded
by the visor of his Red Sox cap,
playing with his fishing lures.
```

As a point of technique, you should always set header and footer traps early in your input script. This is because **nroff** moves past vertical position 0 on the first page of output as soon as it encounters either the first command that causes a break or the first portion of ordinary text to process. If the header trap is not set, **nroff** will not print the header on the first page.

Macro Arguments

Suppose you want to format a cake recipe with **nroff**. The first part of the recipe lists the ingredients:

```
six tons of flour
five pounds of chocolate
four ounces of gravel
seven gallons of buttermilk
one pound of baking soda
```

Each of these lines has the same format: amount, unit of measurement, and ingredient. You can create a macro (call it **re** for recipe) that captures the format of these lines and contains three "slots"; the slots are for the amount, unit of measurement, and ingredient. Each time you use the macro, you indicate what you want to go into each slot and **nroff** substitutes it for you.

```
.de re
\\$1 \\$2 of \\$3
..
.re six tons flour
.re five pounds chocolate
.re four ounces gravel
.re seven gallons buttermilk
.re one pound "baking soda"
```

You defined macros in previous examples, but this example is the first time you have written a macro which takes *arguments*. When you call a macro which takes arguments, you give the arguments on the same command line as the macro name. A macro may have up to nine arguments following it; they are denoted by **\$1**, **\$2**, ..., **\$9**. The first field following the macro name on the line invoking the macro is called **\$1**, the second is called **\$2**, and so on.

If you want to use a string of characters which includes blank spaces as an argument, you must enclose the string inside double quotes, as with the words "baking soda" in the example above. If you forget to include the double quotes, **nroff** distributes the portions of the string separated by blanks to different arguments.

Do not try to use arguments in a macro called by a trap. Macros called by traps do not accept arguments. This should seem reasonable: how can you specify the arguments if you do not control when the macro is called?

If you examined the above example carefully, you probably noticed that the definition of re includes double backslashes rather than single to identify each macro argument:

```
\\$1 \\$2 of \\$3
```

Within the definition of the macro, you should use **\\$1** and so on rather than **\$1** wherever you want **nroff** to substitute the argument you provide when you call the macro.

The reason you should not use **\$1** in the definition of a macro is confusing at first, but worth taking the time to understand in detail. **nroff** processes the definition of a macro when the macro is defined. During the processing, it expands embedded macro calls and strings; the next section of this tutorial describes strings. Subsequently, **nroff** refers to the *processed* definition each time you call the macro. You want the argument names **\$1**, **\$2**, and so on to be found in the macro body *after* the initial reading of the macro definition rather than *before* the initial reading. You do not want **nroff** to substitute for arguments at the time of definition of the macro (when there is nothing to substitute for them) but rather to substitute for them each time you call the macro.

This is what the double backslash accomplishes. When **nroff** reads **\\$1** in the macro definition, it translates it into **\$1** rather than attempting to substitute for it. When you call the macro, **nroff** finds **\$1** in the processed macro body and substitutes the argument from the command line.

If you were to use **\$1** somewhere in the definition of a macro, **nroff** would try to substitute the value of argument 1 for it. But you supply the values for arguments when you *call* the macro, not when you *define* it. The point of having arguments is to allow you to substitute different things for the arguments each time you call the macro. Since **nroff** cannot find a value for the argument at the time of definition, it substitutes nothing for it.

If the time of definition and of a call to a macro are confusing you, think of the paragraph macro in the John and Lizzie story. Towards the very beginning of the script, you define the paragraph macro with the command

```
.de PP
```

followed by the macro body. Later, whenever you want a paragraph in the text, you use the command **PP**; each is a call to the macro.

As a final example, consider a simple paragraph macro using an argument. The output format might seem silly, but the example illustrates the point about double backslashes. The paragraphs produced by this macro look like this:

```
          first paragraph:

          text of paragraph...
     more text of paragraph...

          second paragraph:

          text of paragraph...
     more text of paragraph...

          third paragraph:

          text of paragraph...
     more text of paragraph...
```

You supply the number (such as **first** or **second**) which identifies each paragraph as an argument to the macro. You can define the macro as follows.

```
.de NP
.sp
     \\$1 paragraph:
.sp
.ti 0.5i
..
```

After **nroff** processes this definition, it has the following processed definition for **NP**:

```
.sp
     \$1 paragraph:
.sp
.ti 0.5i
```

nroff uses this sequence of commands each time you call the macro. Notice the argument **\$1** in the processed definition, waiting to be filled by the argument you provide on the **NP** command line. For example,

```
        .NP first
```

produces the tag, as desired.

```
            first paragraph:
```

Now change the definition of **NP** to the following:

```
        .de NP
        .sp
            \$1 paragraph:
        .sp
        .ti 0.5i
        ..
```

When it processes the definition, **nroff** does not find any argument corresponding to **\$1**, so it simply replaces **\$1** with the empty string (the string which contains no characters). Therefore, **nroff** remembers the following commands as the processed definition of **NP**:

```
        .sp
                paragraph:
        .sp
        .ti 0.5i
```

Then **NP** tags every paragraph as follows:

```
             paragraph:

            text of paragraph...
       more text of paragraph...
             paragraph:

            text of paragraph...
       more text of paragraph...
             paragraph:

            text of paragraph...
       more text of paragraph...
```

which is not the output you want.

5. Strings

Suppose you are writing a script for **nroff** and, to relieve the tedium, decide to occasionally punctuate the text with a rousing cry of "FOOD! FOOD! FOOD!". If you plan to include this interjection more than a few times in your script, you can take advantage of another labor-saving device similar to a macro, called a *string*. You can use a string name as an abbreviation for a long string of characters you use frequently. Just like a macro, a string is a *name* which **nroff** associates with a *definition* you supply. Wherever you put the name in your text, **nroff** prints the definition. Whereas macros refer to sets of commands you define, strings refer to strings of characters you define.

You define a string with the *define string* command **ds**:

```
.ds FD "FOOD! FOOD! FOOD!"
```

The first field after the **ds** gives the name of the string, in this case **FD**. Like a macro name, a string name may be either one or two characters. The second field after the **ds** gives the definition of the string, in this case

```
"FOOD! FOOD! FOOD!."
```

As in this example, you should enclose the definition in quote marks if it contains spaces.

You should be careful whenever you define a macro or a string. If you already have a macro or a string named **X** and you define a new macro or string named **X**, **nroff** forgets the previous meaning of **X**.

Once you have defined a string, you can insert a reference to it anywhere in your text. The string itself appears in the output text wherever a reference to it appears in the input text. You refer to the string **FD** in the following fashion:

```
\*(FD
```

Use the left parenthesis '(' only when the name of the string is two characters long. If the string name is only a single character, such as S, refer to it as follows:

```
\*S
```

As an example, try the following **nroff** script:

```
.ll 3i
.po 2i
.ds FD "FOOD! FOOD! FOOD!"
.ds H "HALLELUJAH!"
There is \*(FD a dead frog on my terminal .
He enjoys \*(FD very much (\*H \*H).
```

nroff adjusts the spacings between words in a string but does not hyphenate any word in a string. If you use a very short line length, say two inches, and define a string which includes a three-inch long word, that word would not be hyphenated but would extend past the right-hand margin.

You cannot include a newline character in a string. However, you can spread the definition of a string out over more than one line with the aid of "concealed" newlines (preceded by the backslash character '\'). **nroff** ignores each concealed newline. For example, add the following string to the previous example:

```
.ds pr "PRAISE \
THE LORD!"
```

In fact, **nroff** ignores concealed newlines anywhere in its input.

Strings Within Strings

It is possible to define a string that has embedded within it a reference to another string. Whenever you refer to the bigger string in your text, **nroff** substitutes the definition of the smaller string for any reference to the smaller string. When you embed strings, though, you should use two backslashes to refer to the embedded string. The reasons for this are similar to the reasons for using two backslashes when you refer to an argument within a macro: **nroff** processes the string definition when it occurs in the input text and then processes it again each time you refer to the string. If you define a string similar to:

```
.ds S "This string \*x has embedded \*y strings"
```

the danger is that at the time of definition of S you might not have defined **x** or **y**. If this is the case, **nroff** simply ignores the refer-

ences in S to **x** and **y**; more precisely, it replaces each reference by the empty string (the string which contains no characters).

If you define S as follows:

```
.ds S "This string \\*x has embedded \\*y strings"
```

then **nroff** does not try to substitute for x and y when it first reads the definition of S. Instead, it simply translates *x into *x and *y into *y. When you put the reference *S in your text, **nroff** fetches the definition of S, sees the references to strings x and y, and substitutes the definitions for the references.

To help understand this better, try the following three scripts. The first script contains proper references to embedded strings (using double backslashes); it works as expected.

```
.ds S "strings \\*x, strings \\*y, strings \\*z"
.ds x "here"
.ds y "there"
.ds z "everywhere"
\*S
```

The second script contains embedded references using only single backslashes. Since the embedded strings are defined after the larger string, they are not available when **nroff** defines the larger string, and so the references are ignored.

```
.ds S "strings \*x, strings \*y, strings \*z"
.ds x "here"
.ds y "there"
.ds z "everywhere"
\*S
```

The third script again contains embedded references using single backslashes. This time the embedded strings are defined before the larger string and are available when the larger string is defined.

```
.ds x "here"
.ds y "there"
.ds z "everywhere"
.ds S "strings \*x, strings \*y, strings \*z"
\*S
```

To avoid unnecessary worry, you should always play it safe and use double backslashes to refer to embedded strings.

6. Number Registers

You learned in previous sections that **nroff** keeps track of output page numbers while printing its output. You made use of this fact when you created a footer macro that printed page numbers. **nroff** also keeps track of other housekeeping information, such as the current line length, page offset, page length, and vertical position of the last output line. It keeps this information in storage locations called ***number registers***.

You can use the name of a number register to refer to the number stored in it. When you place a reference to a number register in your text, **nroff** substitutes whatever number is currently in the register.

Number register names are either one or two characters long, just like macro and string names. You can have a number register with the same name as a string or a macro without confusing **nroff**; you might recall that you may not have a macro and a string with the same name. However, *you* might get confused; **nroff** scripts are usually easier to understand if you keep all macro names, string names and register names distinct.

A difference between number registers, macros and strings is that **nroff** itself does not define any macros or strings (although the ms macro package does), but it does automatically define and update quite a few number registers. You can use these predefined number registers in much the same way you use registers you define yourself, except that you cannot change their values.

To define a number register, you need to specify the ***register name*** and the ***initial value*** for the register. The ***number register*** command **nr** looks like this:

```
.nr X 5
```

Here **X** is the name of the register and **5** is the initial value to store in it. To refer to number register **X** in your text, use **\nX**; if the name is two characters long, say **xy**, use **\n(xy**. This system of reference is exactly like that for referencing strings, except for the use of the letter **n** instead of the asterisk *. When **nroff** sees a reference to number register **X**, it automatically substitutes the value stored in **X**. As you will see shortly, **nroff** can do arithmetic, and learning to use number registers is an important part of learning to take advantage of **nroff**'s arithmetic abilities.

A reference to a number register may occur anywhere a number would normally occur. For example, if you set register **X** to 5 as above, you can set the line length to five inches as follows:

```
.ll \nXi
```

This command is essentially the same as

```
.ll 5i
```

if the current value of register **X** is 5.

A familiar problem (with a familiar solution) arises when you refer to a number register inside a macro or a string definition. If you use just one backslash, **nroff** substitutes the value in the register for the reference when it processes the definition of the macro or string. If you have not yet defined the number register in your script, **nroff** automatically substitutes **0** for the reference. If you want the reference to survive the initial processing of the definition so substitution takes place when you call the macro or string, use a double backslash, such as **\\nX** or **\\n(xy**.

Deferring evaluation of a register reference by using a double backslash is particularly important if you change the value of the register throughout your script and want the current value to appear in the macro or string each time you call it. If you use a single backslash to refer to the number register, substitution takes place at the time of *definition* of the macro or string. The processed macro or string body includes a fixed number instead of a reference to a number register, and each time you call it **nroff** uses the same number. The fact that you keep changing the value in the number register is irrelevant to **nroff**, since it no longer sees the reference to the register in the processed macro or string body.

Try the following examples. Make sure you understand why **nroff** prints what it does in each case. The first example defines a string with a register reference preceded by a single backslash.

```
.ds S "Here is a number \nx"
.nr x 55
\*S
\nx
```

The second example is similar, but the register is defined before the string.

```
.nr y 56
.ds T "Here is a number \ny"
\*T
\ny
```

The third example uses a double backslash for the register reference.

```
.ds U "Here is a number \\nz"
.nr z 57
\*U
.nr z 58
\*U
```

The final example uses a single backslash again.

```
.nr w 59
.ds V "Here is a number \nw"
\*V
.nr w 60
\*V
```

The last example illustrates the danger of using a single backslash to refer to a number register within a string definition. You define w before you define **V**, so the value for w is available when **nroff** reads the definition of **V**. **nroff** substitutes the value when it reads the definition; the reference to w is no longer there. You then change the value of w, but as you see in the next call of **V**, the change does not affect the number that appears in **V**. In contrast to this, notice in the third example that the double backslash in the definition of **U** causes the reference to **z** to survive the definition of **U**. Whenever you change the value of **z** and then call **U**, **nroff** substitutes the new value of **z** for the reference to **z** in **U**.

You can also use the **nr** command to increase or decrease the value in a number register. For example, suppose you initially store the value 5 in **X**:

```
.nr X 5
```

You can change the value of **X** to 9 by adding 4, as follows:

```
.nr X +4
```

You can then change the value of **X** to 7 by subtracting 2:

```
.nr X -2
```

A plus or minus sign before a number on the **nr** command line tells **nroff** to add or subtract the given amount from the value in the register.

Since a negative number is always preceded by a minus sign while a positive number usually is not preceded by a plus sign, you can use **nr** to set a register to a positive value in a way that cannot be imitated for negative values. For example, suppose you again start out with a value of 5 in **X**:

```
.nr X 5
```

If you immediately follow this with

```
.nr X 7
```

then **nroff** stores 7 in **X**. The second **nr** command does not increment the value of **X** by 7 to produce 12; instead, it wipes out the previous value of 5 and replaces it by the value 7. The command line to increment **X** by 7 is

```
.nr X +7
```

If you again start with a value of 5 in **X** and want to change the value to −4, you cannot use the following command line:

```
.nr X -4
```

nroff interprets this as a command to decrease the current value of **X** by 4, which is not what you intend. This command places the value 1 in **X**, since 5 − 4 = 1. If **X** initially has a value of 5 and you want to change the value to −4, you could use the command

```
.nr X -9
```

You can also increment or decrement the value of a number register without using the **nr** command. If number register **x** currently has the value 10, the reference **\n+x** increments the value in **x** by 1 to 11 and substitutes the new value for the reference. The value in **x** becomes 11; **nroff** replaces the next reference **\nx** by 11, while another reference **\n+x** increments the value in **x** to 12 and replaces the reference by 12. Similarly, if number register **xy** currently has the value 15, the reference **\n+(xy** increments the value in **xy** to 16 and replaces the reference by 16.

There is a similar way to decrement the value in a register. The reference **\n−x** decreases the current value in **x** by 1 and substitutes the new value for the reference. Likewise, the reference **\n−(xy** decreases the current value in **xy** by 1 and substitutes the new value for the reference.

You can change the size of the increment or decrement by means of another option to the **nr** command. If you define **x** with

```
.nr x 1 5
```

then **nroff** sets the value of **x** to 1 and sets the increment value for **x** to 5. The next reference **\n+x** increments the value in **x** from 1 to 6 (the '+' now causes **nroff** to add 5 to the current value of **x** rather than adding 1) and substitutes 6 for the reference. In the same manner, **\n−x** subtracts 5 from the current value of **x** and substitutes the new value for the reference. This is convenient if you plan to repeatedly increment or decrement **x** by the same fixed amount. If you wish to change the size of the increment, simply redefine **x** with another **nr** command, specifying the new initial value as well as the new increment value. If you define a number register but do not specify an increment value, **nroff** assumes the increment value to be 1.

The following example of a macro illustrates a typical use of a number register and incrementing.

```
.nr b 1
.ds x "Here's Bachelorette No. \\nb"
.de B
.br
\\*x \\$1!!!!!!
.nr b \\n+b
..
.B "Polly Underground"
.B "Lascivia Servant"
.B "Lips La Roux"
```

The output produced by this example is:

```
Here's Bachelorette No. 1 Polly Underground!!!!!!
Here's Bachelorette No. 2 Lascivia Servant!!!!!!
Here's Bachelorette No. 3 Lips La Roux!!!!!!
```

A reference to a number register may appear any place a number can normally appear. For example:

```
.nr x \ny \nz
```

sets register **x** to the value of register **y** and sets the increment for **x** to the value of register **z**.

As mentioned before, **nroff** can evaluate arithmetic expressions. It understands and evaluates properly formed arithmetic expressions involving numbers, references to number registers, the arithmetic operators '+', '−', '*', '/', '%', and parentheses. The first four operators represent addition, subtraction, multiplication and division. The '%' is the "modulus" or "remainder" operator: the value of 7%3 is 1, which is the remainder when 7 is divided by 3.

One word of caution: **nroff** evaluates expressions from left to right without any preference for performing some operations before others. For example,

```
.nr x 5+4*3/9
```

stores 3 in **x**. **nroff** does not perform the multiplication and division before the addition, as you might expect.

Another important fact is that number registers hold only integers. If you write

```
.nr x 3.6
```

nroff truncates the value 3.6 and stores 3 in **x**. Also, an assignment such as

```
.nr x 3.9*3.9
```

stores 9 in **x**; **nroff** truncates each factor before it performs the multiplication. The assignment

```
.nr x 0.4*8
```

stores 0 in **x** rather than 3; truncation occurs before **nroff** performs the multiplication rather than after.

A final word of caution: when you use numbers with commands other than **nr**, the results may *not* be what you expect. **nroff** understands several different units of measurement and converts between units automatically. The next section explains units and conversion in detail.

Units of Measurement

As mentioned above, **nroff** maintains many number registers during processing. For example, it stores the current page length in the register **.l** (notice that the period '.' is actually part of the name of this register). If you set the line length to 5 inches with the command

```
.ll 5i
```

nroff stores the length in register **.l** automatically. If you print the value in register **.l** with a reference such as **\n(.l**, you find the value is 600. What does this mean?

Many **nroff** commands require specification of lengths or measurements as arguments. You are already familiar with many of these commands: for example, **ll**, **po**, **pl**, **lt**, and so on. **nroff** accepts various units of measurement, but for purposes of calculation, it converts each into a basic unit called a *machine unit*, abbreviated **u**. The length of a machine unit is 1/120 of an inch. Since one inch is 120 machine units, the length of a 5 inch line is 5*120=600 machine units.

The conversion table for units of measurement is as follows:

inch:	1i = 120u
vertical line space:	1v = 20u
centimeter:	1c = 47u
em:	1m = 12u
en:	1n = 12u
pica:	1P = 20u
point:	1p = 1u

Most of these are traditional typesetting terms.

As noted briefly in Section 1 of this tutorial, **nroff** output actually consists of a sequence of characters. It is useful, though, to think of the output as being "printed" at ten characters per inch (Pica or 10 pitch spacing) and six lines per inch. Many output devices correspond to this spacing. With these assumptions, 5i corresponds to 5 inches of printed output.

For each command, **nroff** assumes a default unit if you do not specify a unit on the number you supply with the command. For example, the default unit for **ll** and **po** is **m**, while the default unit for **sp** is **v**. If you type

```
.ll 5
```

nroff interprets it not as 5 inches or 5 centimeters but as 5**m**, which it converts to 5*12=60 machine units (60u).

nroff always assumes a unit specification as part of each number and automatically converts each number and its unit specification into machine units. If you append an explicit unit specification to the number, **nroff** uses it; if you do not, **nroff** uses the default unit for the command.

For example, suppose you write the following commands:

```
.nr x 2i
.ll \nx
```

What line length results? The first command stores the number 2*120=240 in register **x**. The second command is therefore the same as

```
.ll 240
```

But the default unit for **ll** is **m**. Since 1**m** is 12**u**, **nroff** sets the line length to 12*240=2880 machine units. If you intended a line

length of 2 inches to result from the above commands, you will be unpleasantly surprised, since 2i=240u. Instead, you should write:

```
.nr x 2i
.ll \nxu
```

By including the **u** in the **ll** command, you prevent multiplication by the scale factor of 12 as in the first example.

The point bears repeating. **nroff** converts every number given to a command as an argument into machine units; if you do not want the number to change, you must append a **u** to it, unless the default unit for the specific command is **u** already. Otherwise, **nroff** multiplies by a conversion factor.

Incidentally, the default unit for the number register command **nr** is **u**; this explains why numbers without unit specifications do not change in value when you assign them to registers. Since the default unit for the **nr** command is **u**, **nroff** does not multiply the numbers by any conversion factor.

You should think of the unit specification as a part of a number. Since **nroff** accepts so many different units of measurement, a number without a unit specification is ambiguous. What does '5' mean? 5 inches? Centimeters? Ems? **nroff** must know what unit of measurement you are using.

Thinking of the unit specification as a part of a number helps explain potentially mystifying behavior like the following. As mentioned before, number registers store only integers and **nroff** truncates each number in an arithmetic expression to an integer before evaluating the expression. Therefore, the following stores 0 in register **x**:

```
.nr x 0.4*9
```

But now try the following:

```
.nr x 0.4i
\nx
```

This does not store 0 in **x** like the previous command; it stores 0.4*120=48 in **x**. The 0.4 is not truncated to 0 here! Truncation occurs *after* conversion to machine units, so **nroff** truncates 0.4u in the first example. But the number in the second example is given in inches **i** instead of machine units **u**. **nroff** converts it to **u** before truncating to get an integer.

As another example, the following stores **1** in **x**:

```
.nr x 0.01i
```

nroff converts 0.01 inches to 0.01*120 = 1.2u and then truncates 1.2 to 1.

The following command illustrates that **nroff** understands *each* number in an arithmetic expression to have an attached unit specification, whether you supply one or not.

```
.ll 2*8
```

Recall that **nroff** stores the current line length in the register **.l**; if you type

```
\n(.l
```

you find 2304 in a register **.l**. **nroff** interprets the **2** as **2m** and the **8** as **8m**, since the default unit for **ll** is **m**. Then it converts each to machine units and multiplies to give the result (2*12)*(8*12) = 2304.

Consider one final example illustrating the unusual consequences of seemingly innocent assignments. Suppose you set the page offset as follows:

```
.po 8/3
```

nroff stores the current page offset in register **.o**; to see what number it stores there, type

```
\n(.o
```

You see that the page offset is 2. Since the default unit for po is **m**, the calculation is: (8*12)/(3*12) = 8/3, which **nroff** truncates to 2. **2u** is only 1/60 of an inch. This is not a physically reasonable value for most typewriter-like devices, so a page offset of 0 characters results. On the other hand,

```
.po 8/3u
```

produces a page offset of approximately 1/4 of an inch.

Conditional Input

Now that you understand number registers, you can use them in conjunction with powerful *conditional commands* to create more elaborate **nroff** scripts. Consider again the problem of creating header and footer macros. In the section on macros, you constructed macros which skipped space at the top of the page and printed the page number at the bottom of each page.

Suppose you are formatting a paper that has a title. You want to print the page number on page 1 at the bottom of the page and to print the rest of the page numbers at the top of the page. Both the header and the footer need some kind of conditional mechanism in order to perform differently on the first page than on subsequent pages.

On page 1, the header should skip to where the title will be printed; on other pages, the header should print the page number. On page 1, the footer should print the page number; on other pages, the footer should leave a block of blank space at the bottom of the page.

To execute commands conditionally, use the *if/else* commands **ie** and **el**.

```
.de hd          \" define header
.ie \\n%=1 .A
.el .B
..
.de A           \" first header option
.sp |1.0i
..
.de B           \" second option
'sp 2v
.tl ''- % -''
'sp |1.0i
..
.de fo          \" define footer
.ie \\n%=1 .C
.el .D
..
.de C           \" first footer option
'sp |-4v
.tl ''- % -''
'bp
..
.de D           \" second option
'bp
..
```

As you can see, the **ie** and **el** commands always occur as a pair. The **ie** command line consists of three parts: first the **ie**, then a *condition* which **nroff** tests, followed by a *command* for **nroff** to perform if the condition is true. In the example, you wanted the commands in various macros performed; you could have written the commands out rather than putting them in another macro, as described below. If the condition on the **ie** command line is not true, **nroff** performs the command on the **el** line instead of the command on the **ie** line.

Each conditional in the example invokes a macro on the command line. Actually, the conditional can specify input text rather than the command after the condition. If you want to execute several commands or include several text lines conditionally, enclose the lines with the special sequences '\{' and '\}'.

You should notice one other new element in the construction of these macros. Some of the sp commands have a vertical bar immediately in front of the measurement; for example,

```
.sp |1.0i
```

Normally, when **nroff** processes a command like **sp 4**, it moves down four vertical spaces on the output page. The number 4 is in a sense a *relative* measurement, relative to where **nroff** happens to be on the output page when it receives the request. The vertical bar tells **nroff** that the following measurement is an *absolute* measurement either from the top of the page (if positive) or from the bottom of the page (if negative). Therefore,

```
.sp |1.0i
```

tells **nroff** to move to one inch from the top of the page;

```
.sp |(-4v)
```

tells it to move to four vertical spaces from the bottom of the page.

The closely related *if* command **if** has a command line formed exactly like **ie**. Unlike **ie**, which must always be used with **el**, **if** commands may be used singly. If the condition on the **if** command line is true, **nroff** performs the command following the condition; if the condition is false, it ignores the command.

This section ends with two rather substantial examples incorporating most of what you have done so far. To illustrate the use of conditionals, the first example begins each even paragraph of output with the phrase **Even Paragraph:** and begins each odd paragraph with the phrase **Odd Paragraph:**.

```
.wh 0 hd         \" set traps; basic formatting
.wh |3i fo
.nr eo 1
.po 2i
.pl 4i
.lt 4i
.de hd           \" header
'sp |(1i-1v)
.ti ''\\*(ws''
'sp |1.5i
..
.de fo           \" footer
'sp |(3i+3v)
.tl ''- % -''
'bp
..
.ds ws "A Wondrous Story"
.de PP           \" paragraph macro
.ie \\n(eo=0 .EP
.el .OP
..
.de EP           \" even paragraph
.br
.nr eo 1
.sp 1v
.ll 4i
.lt 4i
\\*e
.sp 1v
..
.ds e "Even Paragraph:"
.de OP           \" odd paragraph
.br
.nr eo 0
.sp 1v
.ll 3i
.lt 3i
\\*o
..
.ds o "Odd Paragraph:"
```

```
.PP
text...
.PP
text...
.PP
text...
.PP
text...
```

This example uses the "even/odd" register eo to determine whether you are beginning an even or an odd paragraph. To distinguish between even and odd paragraphs, it uses a line length of 4 inches for even paragraphs and a line length of 3 inches for odd paragraphs. It changes the title length with each paragraph, so nroff centers the page number with respect to whichever kind of paragraph happens to occur at the bottom of a page.

The final example illustrates a loop constructed with the if/else commands. The first paragraph is 6 inches long with a page offset of 0; each succeeding paragraph is 1 inch shorter with a page offset 1 inch larger. The line length of the sixth paragraph is 1 inch; the next paragraph renews the cycle with a 6-inch line length.

```
        .nr PO 0 1
        .de PP
        .ie \\n(PO=6 .A
        .el .B
        ..
        .de A
        .br
        .nr PO 0
        .nr LL 6-\\n(PO
        .ll \\n(LLi
        .po \\n(POi
        .nr PO \\n+(PO
        .sp
        ..
        .de B
        .br
        .nr LL 6-\\n(PO
        .ll \\n(LLi
        .po \\n(POi
        .nr PO \\n+(PO
        .sp
        ..
```

You should try this example with at least seven paragraphs of input to see that the "loop" really works as advertised.

7. Environments

By now you should be familiar with the basic **nroff** commands. The commands control the *environment* in which **nroff** processes your input text. The basic features of the environment include line length, fill and adjust modes, indentation, and so on.

nroff actually provides you with three independent environments, labelled 0, 1, and 2. In each of the three, you may set parameters like line length, filling, adjustment, and indentation as you wish. You can call a different environment with the **ev** command; the parameters you select for the new environment control text processing until you change the parameters in the new environment or turn over processing to another environment.

Not all **nroff** parameters change when you switch to a new environment. For example, different environments do *not* have independent page offsets; the **po** command affects all environments. Parameters that may be set to different values in different environments are *environmental parameters*; parameters that cannot be switched according to environment, like page offset, are *global parameters*. Macro and string definitions are global.

When you first call **nroff**, you are by default in environment 0. In all the examples thus far, everything happened in environment 0. The following example illustrates how to switch back and forth between environments. Invoke **nroff** from your terminal and type in the example so you see the output as you go along.

```
.po 1i              \" set global page offset
.ll 4i              \" set parameters in ev 0
.de PP              \" paragraph macro
.sp
.ti 0.5i
..
.PP
text...
.ev 1
.ll 3i              \" set parameters in ev 1
.ls 2
.PP
text...
.ev                 \" return to ev 0
.PP
.ev 1               \" back to ev 1
.PP
text...
.ev                 \" return to ev 0
```

The first **ll** command sets a line length of 4 inches in environment 0. After defining the paragraph macro **PP** and an initial paragraph in environment 0, you switch to environment 1 with the command

```
.ev 1
```

You now enter a new environment, as if you just entered **nroff** in environment 0. If you do not explicitly set environmental parameters like line length, **nroff** automatically uses default values for them. **nroff** assigns the same default values in environments 1 and 2 as in environment 0.

You set the line length in environment 1 to 3 inches with the output text double-spaced. The *leave space* command

```
.ls 2
```

leaves 2 − 1 = 1 blank lines between each output line. Thus, paragraphs processed in environment 0 have 4-inch single-spaced lines, while paragraphs processed in environment 1 have 3-inch double-spaced lines.

In the example, you use the command line

```
.ev
```

without any number after the **ev** to leave environment 1. This leaves environment 1 and restores ("pops") previous environment 0. The next time you pass to environment 1, you do not need to set the line length to 3 inches again; the value stays in effect in environment 1 until you specifically change it. The same is true of all environmental parameters.

To understand how **nroff** switches between environments, imagine you have a set of plates, each marked with either a 0, a 1, or a 2. You have as many plates of each type as you wish. You stack the plates on a table; the top plate represents your current environment. Start with a 0 plate on the table to represent the initial environment when you enter **nroff**.

Switching to environment 1 with an **ev 1** command corresponds to placing a 1 plate on top of the 0 plate. After you do so, you can change the stack of two plates by placing a new plate on top of the stack or by removing the top plate from the stack. The former corresponds to calling a new environment, while the latter corresponds to restoring the previous environment with the command line **ev**.

Since you have as many plates of each type as you wish, you can call environment 1, then call environment 2, then restore environment 1, then call environment 0, and so on; that is, you can stack a 1 plate, then stack a 2 plate, then remove the 2 plate, then stack a 0 plate... The command **ev** *N*, where *N* is 0, 1, or 2, puts a plate on the stack; the command **ev** removes the top plate from the stack.

To illustrate this, add the following lines to the previous example. You might want to draw a picture of the stack of environments and keep track of how the **ev** commands add or remove "plates". Since the line lengths are different in each environment, it should be easy to tell in which environment **nroff** processes each paragraph.

```
.ev 2           \" environment 2
.ll 5i          \" set parameters there
.in 1i
.PP             \" paragraph in ev 2
text...
.ev 0           \" go to ev 0
.PP
text...
.ev 1           \" go to ev 1
.PP
text...
.ev 2           \" go to ev 2
.PP
text...
.ev 0           \" go to ev 0
.PP
text...
.ev             \" return to ev 2
.ev             \" return to ev 1
.PP
text...
.ev             \" return to ev 0
.ev             \" return to ev 2
.PP
text...
```

In Section 2, you learned that **nroff** uses a buffer to assemble words from its input into output lines. Actually, each environment has its own buffer. Switching to a new environment does *not* cause a break. Suppose you are currently in environment 1 with an unfinished line in the buffer. When you give the command ev 2, the unfinished line remains undisturbed in the environment 1 buffer until you return to environment 1. Text you process in the meantime in environment 2 or in environment 0 has no effect on the partial line in the environment 1 buffer, since **nroff** assembles text processed in other environments in different buffers.

In the following example, you process some text in environment 0 and then switch to environment 2. Any partial line collected in environment 0 when you switch to environment 2 waits patiently in

the buffer until you return to environment 0 and issue the break command to flush the buffer. You then return to environment 2 and flush any partially filled line left when you restored environment 0.

```
.ll 3i
.po 2i
text for ev 0...
.ev 2
text for ev 2...
.ev
.br             \" flush buffer 0
.ev 2
.br             \" flush buffer 2
```

A common use of environment switching is for the creation of header and footer macros. As the following example suggests, the length of title set by the **lt** command is an environmental parameter. The example constructs header and footer macros which print strings of asterisks in the margins above and below the text.

```
.wh 0 hd
.wh |2.5i fo
.de hd
.ev 1
.lt 5i
'sp 3v
.tl '*****''*****'
'sp 2v
.ev
..
.de fo
'sp 2
.ev 1
.tl '*****'%'*****'
.ev
'bp
..
.ll 4i
.pl 3i
.in 1i
.po 2i
.de PP
.sp 1
.ti 1.5i
..
.PP
text...
```

The following section explains why header and footer macros often use a different environment.

More About Fonts

As described in some detail in Section 1, **nroff** output includes representations for **boldface** and *italic* characters, in addition to normal Roman characters. The visual appearance of boldface and italic characters depends on the device you use to "print" your **nroff** output.

If you want a single word or a short phrase to appear in boldface, enclose the word or phrase between **\f3** and **\f1**:

```
The last word of this sentence is in \f3boldface\f1.
```

The sequence **\f3** tells **nroff** to print in boldface, while the sequence **\f1** tells **nroff** to return to the Roman font. Similarly for italics:

An entire phrase \f2appears in italics\f1.

To print more than a few words in a different font, you should use the *font* command **ft**:

```
.ft I
Here is text you want to
appear in italics...
.ft R
```

The initial **ft I** switches to italic font, while the concluding **ft R** returns to Roman font. As you might suspect, the command **ft B** switches to boldface.

You have two additional options when you use the **ft** command. The command **ft P** returns to the *previous* font. You can use **ft P** within a macro or a string to return to the previous output font, even though you may not be sure which font was previously in effect. You can also use the sequence **\fP** to return to the previous font. The **ft** command without an argument tells **nroff** to return to the Roman font.

In scripts with frequent font changes, you should switch to a new environment for header and footer macros. Suppose you have a header macro that prints a title and date and a footer macro that prints page numbers. If the header and footer macros contain no font specifications, they usually print in Roman. If they use the main environment, a problem like the following could arise. Suppose the input includes a block of boldface or italic text which happens to extend across more than one page. The text trips the footer trap while in a different font, so **nroff** prints the header and footer in a different font. The unpleasant effect is that sometimes **nroff** prints the header and footer in Roman, sometimes in boldface, and sometimes in italics.

To avoid this problem, take advantage of the fact that the current font is an environmental parameter. Pass to a new environment for the header and footer and set the font as you wish; every time each macro is called, it prints in the same font.

Diversions

Suppose you use **nroff** to format a chapter of a book. The chapter includes footnotes at various places in the text that you want **nroff** to collect and print at the end of the chapter. You want to enter each footnote at the point in the input text where the reference to it occurs, but you do not want it to appear there in the output. You want to store the processed text of the footnote somewhere until you tell **nroff** to print it.

The major question is: if you do not want **nroff** to print the text of a note when it processes it, where do you store the text until you want it to appear? **nroff** provides a *diversion* mechanism to handle this problem: you can *divert* text to temporary storage in a macro. Diverted text does not appear in the output when **nroff** processes it. It is stored in a macro, so it appears in the output when you invoke the macro.

Diversion normally involves passing to a new environment to process the footnote without causing a break in the main environment. When the text of the note ends, **nroff** returns to the main environment, again without causing a break, so processing continues just as if the text of the note had not been in the input.

Before you attempt to construct a footnote macro, consider the following simple example. It illustrates the basic features of diversion. The net effect of the example is to interchange the two paragraphs, so **nroff** prints the second before the first.

```
.di X           \" divert following to macro X
.sp
text of first paragraph,
printed last...
.br             \" send last line of paragraph to X
.di             \" end diversion
.sp
text of second paragraph,
printed first...
.X              \" print the paragraph diverted to X
```

The new command here is the *divert* command **di**. The command **di X** tells **nroff** to divert the following text to macro **X** and the matching **di** with no argument marks the end of the diversion.

The break is necessary before the end of the diversion because **nroff** diverts *processed* text into the macro. Without the break, **nroff** would not divert any partially filled line in its buffer to **X**; the last few words of diverted text might not form a complete line in the buffer, so **nroff** might not divert them. But if you cause a break before you end the diversion, **nroff** also diverts the last words.

Edit the previous example by deleting the **.br** before the end of the diversion and try it to see what happens. **nroff** prints any trailing words that were not diverted at the beginning of its output. The trailing words are left in the buffer, so **nroff** prints them when it encounters the **sp** command preceding the text of the second paragraph.

The next example illustrates a similar point.

```
.br             \" clear buffer
word1           \" put 'word1' in buffer
.di X           \" divert to X
Put your own
lines of text here.
.br             \" divert last line
.di             \" end diversion
.X              \" print text in X
```

Here **nroff** diverts **word1** to **X** along with the text between **di X** and **di**. Why did this happen? The command **di X** does *not* cause a break. Since you do not pass to a new environment in this example before you divert, **nroff** forms the diversion text in the same buffer in which it stored **word1**. You do not cause a break, so **nroff** appends the diverted text to **word1**.

To make sure **nroff** diverts only text between **di X** and **di** to **X**, you should do one of the following. If you want to process the diverted text in the current environment, empty the buffer by causing a break with the **br** command before you start the diversion. If you switch to a new environment before starting the diversion, you probably want to flush the buffer for the new environment before you start processing diverted text.

Diverting processed text to a macro which is already defined destroys the previous definition of the macro. In some cases, such as the footnote example, you want instead to append information to the same macro. The *divert and append* variation **da** of the diversion construction allows you to do so.

```
.ll 3i
.po 2i
.de PP
.br
.sp 1
.ti 0.5i
..
.di X
.PP
text of paragraph 1
.br
.di
.X                \" see what is in X
.da X             \" add another paragraph to X
.PP
text of paragraph 2
.br
.di
.X                \" see what is in X
```

In this example, you first divert a single paragraph to the macro **X**. The text **nroff** stores in **X** is the *processed* paragraph. In other words, the command line **.PP** is *not* stored in **X**; its output *is*. When you invoke **X** with the command line **.X**, **nroff** uses the processed text in **X** as input. To **nroff**, there is no difference between processed text and unprocessed text as input: it processes the contents of **X** in the current environment, just like any other text. **nroff** processes diverted text *twice*: first when it stores the text in the macro, then again when you invoke the macro.

The fact that **nroff** processes diverted text twice can cause problems if you are not careful. Fortunately, nothing strange happens in the example above. You store a processed paragraph with three-inch long lines in **X**. When you invoke **X**, the line length is three inches. Since each line in **X** is already exactly three inches long, nothing

happens to it when reprocessed; the layout of the output paragraph is unchanged.

But now consider the following example:

```
.ll 3i
.po 2i
.de PP
.sp 1
.ti 0.5i
..
.di X
.ev 2
.ll 4i
.PP
text
.br
.ev
.di
```

A paragraph processed in environment 0 in this example has three-inch lines; you want your diverted paragraph to have four-inch lines. If you print the diverted paragraph with the command line **.X**, what happens? **nroff** does *not* print four-inch lines. Reprocessing takes place in environment 0, with a three-inch line length, so the output paragraph has three-inch lines, contrary to your wishes.

There are two ways to prevent such disasters. If you want to invoke **X** in the main environment, use no-fill mode:

```
.nf               \" no-fill mode
.X
.fi               \" back to fill mode
```

In no-fill mode, **nroff** outputs lines of input exactly as it receives them, so it keeps four-inch lines four inches long and does not change the format of the diverted text.

Another alternative is to return to environment 2 and then invoke **X**; again, the format of the diverted paragraph does not change, since the line length in environment 2 is four inches.

```
.ev 2           \" switch to env. 2
.X
.ev             \" restore original environment
```

The footnote example which follows does not print notes at the bottom of each page, but rather prints all at the end of the chapter. In the processed text, the footnote number appears in square brackets at the point you refer to it.

```
.de FN
[\\n+(fn]       \" footnote reference in main environment
.ev 1           \" environment 1
.da Z           \" append footnote to Z
.sp
\\n(fn. \\$2, \\f2\\$1\\f1,
     \\$3, \\$4.
.br             \" flush diversion buffer
.di             \" end diversion
.ev             \" restore original environment
..
```

Note that requests to change fonts are preceded by double backslashes, since they are inside a macro. The change to italic prints the first macro argument, which should be the title of the work, in italics. Register **fn** contains the number of the last footnote; you should initialize it with the command

```
.nr fn 0 1
```

In your input text, each footnote looks like this:

```
.FN "The Single Man's Guide to Husbandry"\
"Gomez Adams" "Plutonian Press" "1956"
```

When you print the diversion **Z** at the end of the chapter, each footnote has the format

8. Gomez Adams, *The Single Man's Guide to Husbandry*, Plutonian Press, 1956.

8. Command Line Options

In the previous sections, you learned how to control **nroff** by including *commands* in the input along with the *text*. You can also supply information in another way: on the COHERENT command line you type to call **nroff**. Unlike the commands discussed above, this information is *not* part of the input. This section provides more details on options available when calling **nroff**.

You already know about some simple **nroff** command lines. For example, the command

```
nroff
```

accepts input from the terminal (sometimes called the *standard input*) and prints output on the terminal (the *standard output*). Type **<ctrl-D>** (that is, hold down the ctrl key and type **D**) to exit from **nroff** if it is reading input from your terminal.

The command line

```
nroff script.r
```

takes input from the file **script.r** instead of your terminal, while

```
nroff -ms script.r
```

processes **script.r** with the **ms** macro package. You can also redirect **nroff** output to another file **target**:

```
nroff -ms script.r >target
```

The general form of the **nroff** command line is:

nroff *[option ...] [file ...]*

This means that the command line consists of the name **nroff**, followed by zero or more *options*, followed by zero or more *files*. **nroff** processes each given *file* and prints the result on the standard output (the terminal, unless redirected). If no *file* argument is given, as in the first example above, **nroff** reads from the standard input (the terminal, unless redirected).

Each *option* on the command line must begin with the character '–' to distinguish it from a *file* specification. Using **nroff** with the **ms** macro package is one example of using an option. In general, the **–m** option takes the form

-m*name*

which means the option consists of the characters **-m** immediately followed by a *name*. This tells **nroff** to process the macro package found in the COHERENT file

/usr/lib/tmac.*name*

For example, the **ms** macro package discussed in Section 1 is in the file **/usr/lib/tmac.s**, while the **man** macro package used for the **man** command and to process the *COHERENT Command Manual* and *COHERENT System Manual* is in the file **/usr/lib/tmac.an**.

The **-i** option tells **nroff** to read input from the standard input after processing each given *file*. This allows you to supply additional input interactively from your terminal.

The **-x** option tells **nroff** not to move to the bottom of the last output page when done. This is especially useful if you want to see the output on the screen of a CRT terminal.

The **-n***N* option sets the page number of the first output page to the given number *N*, rather than starting at page 1. This is useful for processing large documents with input text in several files which **nroff** processes separately.

The **-r***aN* option sets the value of number register *a* to the given number *N*. Here *a* stands for a single character which identifies a number register. This option lets you initialize number registers when you invoke **nroff**. Section 5 gives more information about using number registers.

The COHERENT system provides many useful features which can be helpful while you are using **nroff**. In particular, you can use a number of special characters. The *stop-output* and *start-output* characters, usually <ctrl-S> and <ctrl-Q>, stop and restart output on your terminal. The *interrupt* character, ususally DEL, interrupts program execution; you can use it to stop an **nroff** command if you typed the command line incorrectly. The *kill* character, usually <ctrl-\>, also terminates program execution. Some COHERENT systems use different characters than those mentioned above; consult the *COHERENT Command Manual* or the *Introduction to the COHERENT System* for details.

Conclusion

This concludes the **nroff** tutorial. By now you should understand enough about **nroff** to create macros on your own and to understand the macros in the **ms** package. The summary in the following section provides a brief description of the most important **nroff** commands discussed in this tutorial.

9. Summary

Macros defined in the ms macro package:

.AB	Abstract begin
.AE	Abstract end
.AI	Author's institution
.AU	Author
.B	Boldface
.BD	Block-centred display
.CD	Centred display
.DE	Display end
.DS	Display start
.FE	Footnote end
.FS	Footnote start
.I	Italic
.ID	Indented display
.IP	Indented paragraph
.KE	Keep end
.KS	Keep start
.LD	Left display
.NH	Numbered heading
.PP	Paragraph
.QE	Quoted paragraph end
.QS	Quoted paragraph start
.R	Roman
.RE	Relative indent end
.RS	Relative indent start
.SH	Subheading
.TL	Title

Basic nroff commands:

.ad	Adjust
.bp	Begin page
.br	Break
.ce	Center
.da	Divert and append
.de	Define macro
.di	Divert
.ds	Define string
.el	Else
.ev	Environment
.fi	Fill
.ft	Font
.ie	If/else
.if	If
.in	Indent
.ll	Line length
.ls	Leave spaces
.lt	Length of title
.na	No adjust
.nf	No fill
.nr	Number register
.pl	Page length
.pn	Page number
.po	Page offset
.sp	Spaces
.ta	Tab set
.tc	Tab character
.ti	Temporary indent
.tl	Title
.vs	Vertical space
.wh	When (set trap)

Number registers:

.i	Indent
.l	Line length
.o	Page offset
.p	Page length
%	Page number

Special character sequences:

\\	Backslash character
\$1	Macro argument 1
\"	Comment
*s	String s
*(st	String st
\nx	Number register x
\n(xy	Number register xy
\<newline>	Embedded newline, ignored
\{	Start of conditional commands
\}	End of conditional commands
\fX	Font (*X* is B, I, P, R)

Index