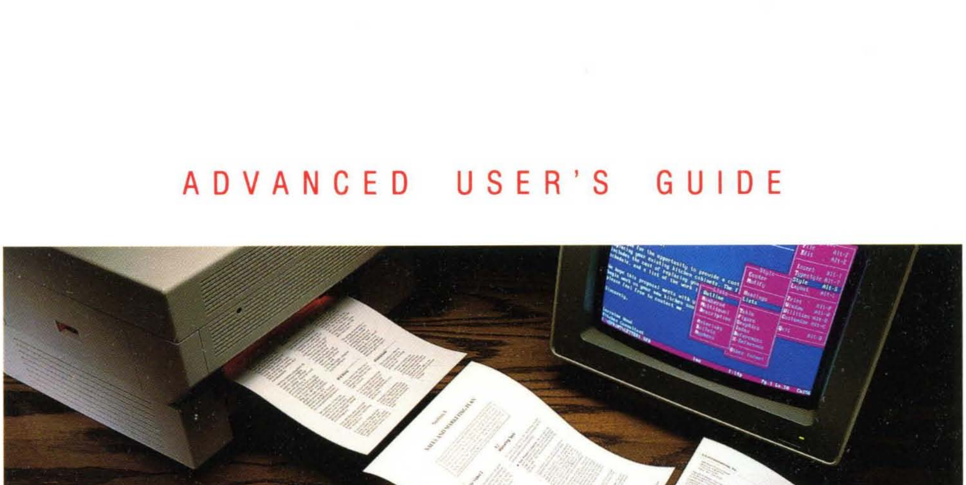
SPRINT®

THE PROFESSIONAL WORD PROCESSOR

ADVANCED USER'S GUIDE



BORLAND
INTERNATIONAL

SPRINT®

The Professional Word Processor

Borland's No-Nonsense License Statement!

This software is protected by both United States copyright law and international treaty provisions. Therefore, you must treat this software *just like a book,* with the following single exception. Borland International authorizes you to make archival copies of the software for the sole purpose of backing-up our software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another, so long as there is **no possibility** of it being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated).

LIMITED WARRANTY

With respect to the physical diskette and physical documentation enclosed herein, Borland International, Inc. ("Borland") warrants the same to be free of defects in materials and workmanship for a period of 60 days from the date of purchase. In the event of notification within the warranty period of defects in material or workmanship, Borland will replace the defective diskette or documentation. **If you need to return a product, call the Borland Customer Service Department to obtain a return authorization number.** The remedy for breach of this warranty shall be limited to replacement and shall not encompass any other damages, including but not limited to loss of profit, and special, incidental, consequential, or other similar claims.

Borland International, Inc. specifically **disclaims** all other warranties, expressed or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the diskette and documentation, and the program license granted herein in particular, and without limiting operation of the program license with respect to any particular application, use, or purpose. **In no event shall Borland be liable for any loss of profit or any other commercial damage, including but not limited to special, incidental, consequential or other damages.**

GOVERNING LAW

This statement shall be construed, interpreted, and governed by the laws of the state of California.

First Edition
Printed in U.S.A.
9 8 7 6 5 4 3 2

SPRINT® The Professional Word Processor

Advanced User's Guide

Borland International
4585 Scotts Valley Drive
P.O. Box 660001
Scotts Valley, CA 95066-0001

Printed in the U.S.A.

10 9 8 7 6 5 4 3 2

This manual was produced with
Sprint:® The Professional Word Processor

Table of Contents

Part 1 Advanced Formatting

Part 3 Appendixes

List of Figures

List of Tables

INTRODUCTION

This book is designed for people who are familiar with Sprint or are sophisticated computer users. This book is for you if you're interested in taking full advantage of Sprint's considerable advanced formatting capabilities, or if you want to customize the program. Although you needn't be a programmer to read this book, you should be ready and willing to delve into more complex techniques.

The *Advanced User's Guide* contains a hefty section on advanced formatting. We give you hands-on experience with cross-referencing, variables, the STANDARD.FMT file, precise layout and design, modifying and creating your own formatting commands, and more. This book also contains information about programming the Sprint editor. We tell you how to use the built-in macro language to reconfigure the menu system or even write your own. There is also information about programming Sprint to work with "non-standard" hardware, such as unusual printers or terminals.

Sprint is really far more than a word processor, or even a desktop publishing tool. It is also a complete, high-level programming language. Using this language, you can make Sprint behave any way you want it to. The Sprint *Advanced User's Guide* contains a complete tutorial for the macro programming language, so if you've been waiting for the right opportunity to increase your computer skills, now's your chance.

To become familiar with the program, you'll probably first want to work through the *User's Guide* for basic how-to information about the Sprint editor and formatter. The *Reference Guide* contains alphabetically listed information about all aspects of the editor and formatter.

This manual consists of the following sections:

Part 1, "Advanced Formatting," contains an advanced tutorial and subsequent chapters on advanced formatting techniques. The chapters build in complexity, with later chapters explaining the nitty-gritty of modifying .FMT files and creating your own formatting commands.

Part 2, "Programming Editor Macros," contains a complete tutorial and alphabetically organized reference to the Sprint macro language. If you're a power user looking to build a new UI, or even if you're just curious about how you can make Sprint behave in different ways, this section is for you.

Part 3, "Appendixes," contains appendixes on more technical information, such as built-in format commands, parameters, key codes, internal file format, the configuration library, hardware control strings, and so on.

Typographic Conventions

All typefaces used in this manual were produced by Sprint, and output on a PostScript typesetter. Their uses are as follows:

`Monospace type`	This typeface represents text as it appears on the screen as well as text you type from your keyboard.
Italic	Italic type is used for emphasis, to introduce a new term, and to represent parameters, variables, and non-primitive editor macros.
Keycap	This special typeface indicates a key on your keyboard. It is often used when describing a particular key you should type, for example, "Press Esc to cancel a menu."

Hardware and Software Requirements

Sprint runs on the IBM PC family of computers, including the XT and AT, along with true IBM compatibles. A two- or three-button mouse is optional.

Sprint requires:

- DOS 2.0 or higher
- At least 384K of RAM

Sprint is not copy-protected, so you can easily transfer it to a hard disk or RAM disk. However, you should read Borland's No-Nonsense License Agreement at the front of this manual for an explanation of your responsibilities with respect to copying Sprint, and then sign and mail it to us.

Borland's No-Nonsense License Statement

This software is protected by both United States Copyright Law and International Treaty provisions. Therefore, you must treat this software *just* *like a book* with the following single exception: Borland International authorizes you to make archival copies of Sprint for the sole purpose of backing up your software and protecting your investment from loss.

By saying, "just like a book," Borland means, for example, that this software may be used by any number of people and may be freely moved from one computer location to another so long as there is **no possibility** of its being used at one location while it's being used at another. Just like a book that can't be read by two different people in two different places at the same time, neither can the software be used by two different people in two different places at the same time. (Unless, of course, Borland's copyright has been violated.)

How to Contact Borland

The best way to contact Borland is to log on to Borland's Forum on CompuServe: Type `GO BOR` from the main CompuServe menu and select "Enter Business Products Forum" from the Borland main menu. Leave your questions or comments there for the support staff to process.

If you prefer, write a letter detailing your comments and send it to:

Technical Support Department
Borland International
P.O. Box 660001, 4585 Scotts Valley Dr.
Scotts Valley, CA
95066-0001, USA

You can also telephone our Technical Support department. Please have the following information handy before you call:

- Sprint version number and user interface name
- computer make and model number
- operating system and version number

PART

1

Advanced Formatting

CHAPTER

1

Advanced Tutorial

The Advanced Tutorial goes beyond the simple text-editing techniques you learned in the Quick Start Tutorial (in the *User's Guide*) to introduce Sprint's more advanced editing and formatting commands. In Quick Start, you learned how to start Sprint; choose a command; create, open, and close files; enter, correct, change, and move text; and other basic functions. In this chapter, we'll build on the concepts and commands introduced in Quick Start, so we assume you've worked through those nine lessons.

The features covered in the Advanced Tutorial give you the desktop publishing power to create professional business proposals and reports and organize large, integrated documents like this manual.

When you complete this tutorial, you'll know how to

- use windows to edit multiple files
- create numbered and unnumbered headings
- define your own menu shortcuts
- title and reserve space for figures and tables
- format tables using precise ruler settings
- add headers and footers to the top and bottom of printed pages
- insert footnotes and cross-references
- check for correct spelling and conditionally hyphenate words
- modify Sprint formats (like lists or tables)
- display a formatted file onscreen to check pagination and error messages
- correct error and warning messages and override default page breaks
- print a file using several different print options

We assume you're using Sprint's advanced user interface; if not, you'll load it in Lesson 1. If you're using one of the alternative user interfaces, see the *Alternative User Interfaces* booklet for information.

Before You Start

Floppy disk systems:

If you're using a system with two floppy disks (no hard disk), make sure that your Data Disk—the disk created by SP-SETUP, which must be in Drive B whenever you use Sprint—contains the files PROPOSAL.SPR and CABINET.SPR. If you don't have these files on your Data Disk, you need to run the SP-SETUP program described in the "Before You Begin" chapter of the *User's Guide.*

Hard disk systems:

If you'll be working on a hard disk, make sure the files PROPOSAL.SPR and CABINET.SPR are in the Sprint directory or the directory of files you'll be working with. You'll need both files for this tutorial. If these files don't appear in the Sprint directory list, run the SP-SETUP program described in Chapter 1 of the *User's Guide* again.

When you used the SP-SETUP program to install Sprint, it automatically added a path to Sprint in your AUTOEXEC.BAT file (see "Before You Begin" in the *User's Guide* for details). This allows you to load Sprint from anywhere on your system (by typing SP), not just from the directory where you store your Sprint files.

If any of these instructions don't make sense, refer to the "DOS Primer" appendix in the *User's Guide.*

What You'll Create

In this tutorial, you'll combine part of the kitchen proposal you completed in Quick Start (the file PROPOSAL.SPR) with an existing work order for a set of cabinets (the file CABINET.SPR). You'll create a new work order for custom cabinets, which you'll modify to suit the requirements of the job. You'll also search for and replace text, add numbered section headings and new list formats, add headers and footers, insert cross-references and a footnote, and use precise ruler settings to format a columnar table.

This type of real-world scenario begins to show off some of Sprint's speed and flexibility; you'll see how easy it is to make a better-looking document in a hypothetical "produce-a-spec-yesterday" situation.

Each lesson contains a brief explanation of the commands you'll use, a step-by-step practice session, and then a table summarizing any commands that weren't covered in the Quick Start Tutorial.

Note: If you find you're having trouble at any time while you're in Sprint, press *F1*, and Sprint will display help information about whatever you're doing.

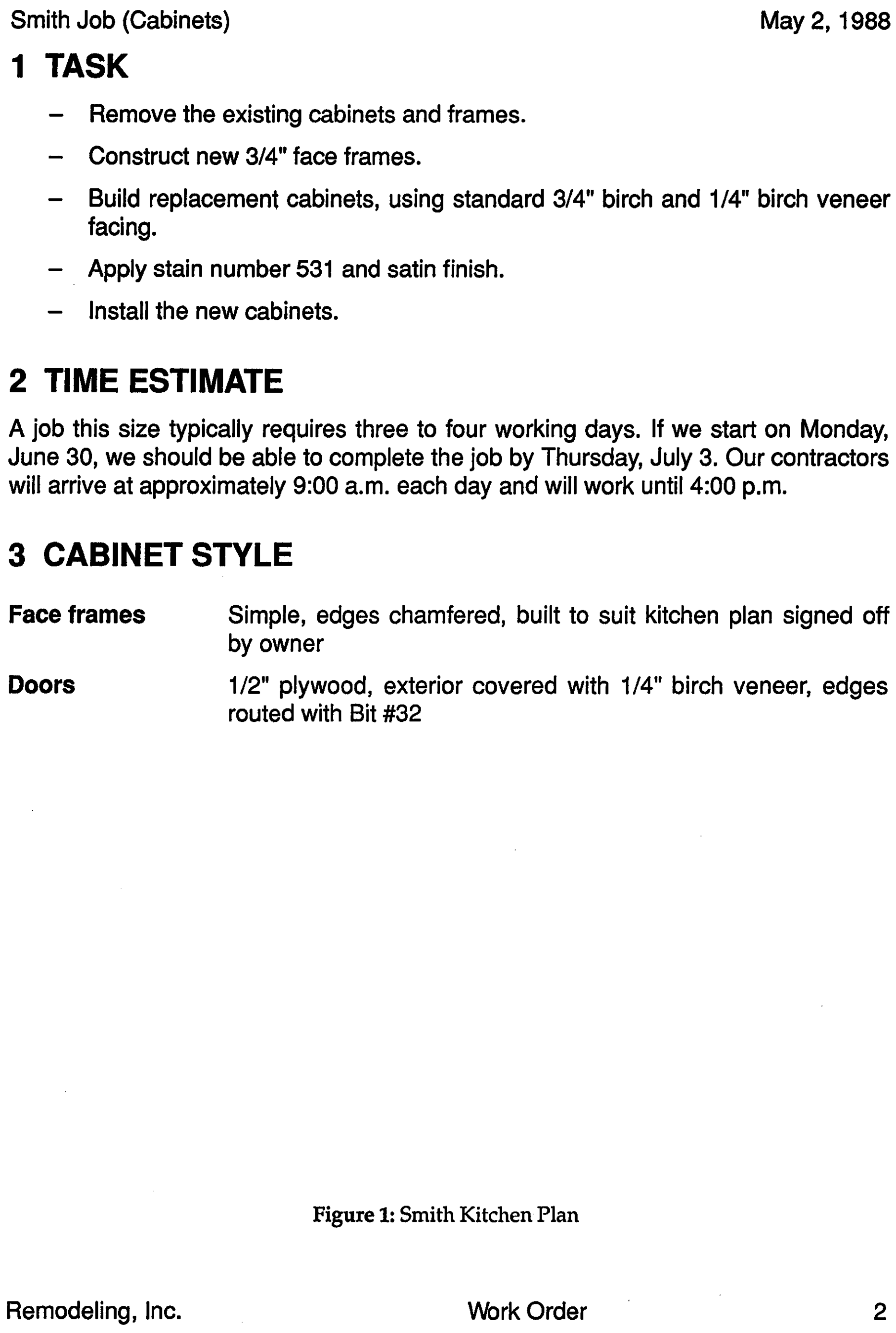
Note to two-floppy system users: If the help files you need to use context-sensitive help are not on your Program Disk, Sprint will prompt you to remove the Program Disk from Drive A and insert the disk that contains the files.

If any of the concepts or commands you see here are unclear, or if you want more information about a particular subject, be sure to refer to the *Reference Guide*. The "Editing: Tips, Tricks, and Techniques" and "Basic Formatting" chapters of the *User's Guide* also cover Sprint functions in greater detail.

The Final Result

The following two pages show you the printed result of this tutorial.

Note: We used a PostScript typesetter to produce the work order. Your printed copy may look somewhat different, depending on the kind of printer you're using.



Smith Job (Cabinets) May 2, 1988

1 TASK

- Remove the existing cabinets and frames.
- Construct new 3/4" face frames.
- Build replacement cabinets, using standard 3/4" birch and 1/4" birch veneer facing.
- Apply stain number 531 and satin finish.
- Install the new cabinets.

2 TIME ESTIMATE

A job this size typically requires three to four working days. If we start on Monday, June 30, we should be able to complete the job by Thursday, July 3. Our contractors will arrive at approximately 9:00 a.m. each day and will work until 4:00 p.m.

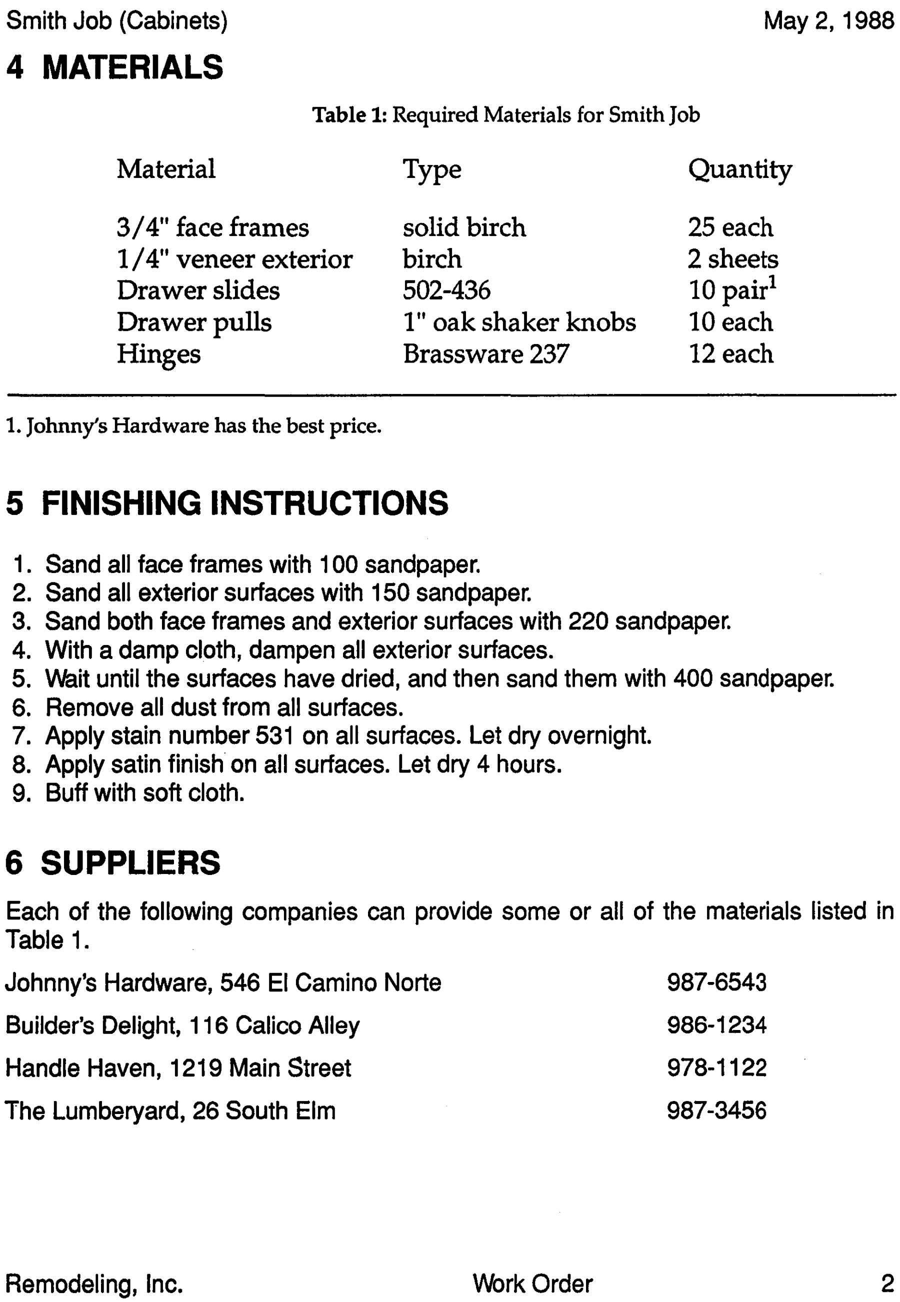
3 CABINET STYLE

Face frames Simple, edges chamfered, built to suit kitchen plan signed off by owner

Doors 1/2" plywood, exterior covered with 1/4" birch veneer, edges routed with Bit #32

Figure 1: Smith Kitchen Plan

Remodeling, Inc. Work Order 2



Smith Job (Cabinets) May 2, 1988

4 MATERIALS

Table 1: Required Materials for Smith Job

Material	Type	Quantity
3/4" face frames	solid birch	25 each
1/4" veneer exterior	birch	2 sheets
Drawer slides	502-436	10 pair[1]
Drawer pulls	1" oak shaker knobs	10 each
Hinges	Brassware 237	12 each

[1] Johnny's Hardware has the best price.

5 FINISHING INSTRUCTIONS

1. Sand all face frames with 100 sandpaper.
2. Sand all exterior surfaces with 150 sandpaper.
3. Sand both face frames and exterior surfaces with 220 sandpaper.
4. With a damp cloth, dampen all exterior surfaces.
5. Wait until the surfaces have dried, and then sand them with 400 sandpaper.
6. Remove all dust from all surfaces.
7. Apply stain number 531 on all surfaces. Let dry overnight.
8. Apply satin finish on all surfaces. Let dry 4 hours.
9. Buff with soft cloth.

6 SUPPLIERS

Each of the following companies can provide some or all of the materials listed in Table 1.

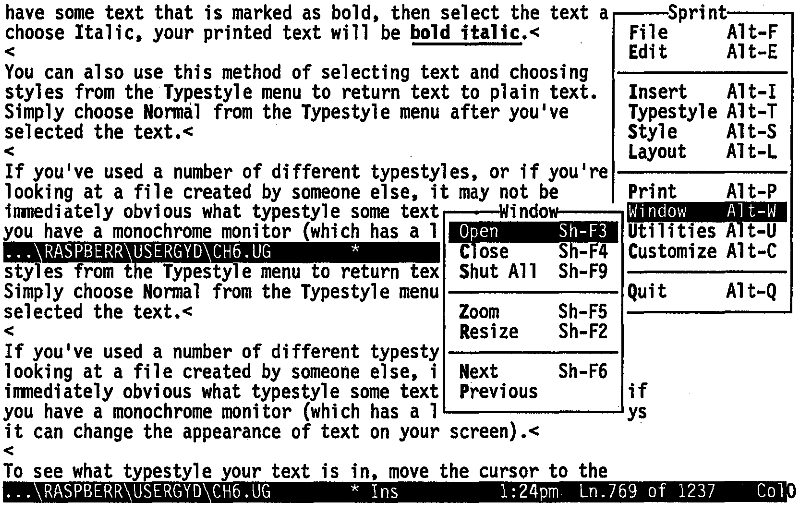
Johnny's Hardware, 546 El Camino Norte	987-6543
Builder's Delight, 116 Calico Alley	986-1234
Handle Haven, 1219 Main Street	978-1122
The Lumberyard, 26 South Elm	987-3456

Remodeling, Inc. Work Order 2

Lesson 1: Opening Files and Windows

Window Commands

If you want to work with two files at one time, as you will in this lesson, the easiest way to do it is to open up a window. This lets you view the two files at once, on a single screen. When you need to work with multiple files, Sprint allows you to open as many as 24 files in up to six windows. You'll find the commands you need to handle multiple windows on the **Window** menu (Figure 1.1); to reach it, press either *F10 W* or *Alt-W.*



```
have some text that is marked as bold, then select the text a
choose Italic, your printed text will be bold italic.<
<
You can also use this method of selecting text and choosing
styles from the Typestyle menu to return text to plain text.
Simply choose Normal from the Typestyle menu after you've
selected the text.<
<
If you've used a number of different typestyles, or if you're
looking at a file created by someone else, it may not be
immediately obvious what typestyle some text
you have a monochrome monitor (which has a l
...\RASPBERR\USERGYD\CH6.UG          *
styles from the Typestyle menu to return tex
Simply choose Normal from the Typestyle menu
selected the text.<
<
If you've used a number of different typesty
looking at a file created by someone else, i
immediately obvious what typestyle some text                if
you have a monochrome monitor (which has a l                ys
it can change the appearance of text on your screen).<
<
To see what typestyle your text is in, move the cursor to the
...\RASPBERR\USERGYD\CH6.UG          * Ins         1:24pm  Ln.769 of 1237    Col0
```

Figure 1.1: The Window Menu

Opening a window not only lets you view multiple files at once; you can also view different parts of the same file, which is useful when you're working on a large document.

Sprint has set of useful shortcut keys for Window menu commands; also, as you'll see in Table 1.1, several of these shortcuts have no menu equivalent.

Table 1.1: Window Shortcuts

Keystroke	Menu Command	Function
Shift-F2	Window/Resize	Adjusts the size of the active window with the plus (+) and minus (–) keys.
Shift-F3	Window/Open	Opens a window onscreen.
Shift-F4	Window/Close	Closes a window.
Shift-F5	Window/Zoom	Instantly expands the active window to fill the screen (pressing *Shift-F5* again returns the windows as they were before the zoom).
Shift-F6	Window/Next	Moves the cursor between open windows.
Shift-F7		Scrolls everything in the window up one line.
Shift-F8		Scrolls everything in the window down one line.
Shift-F9	Window/Shut All	Closes all windows.

The File Manager

When you choose File/File Manager, Sprint displays a menu of convenient file-handling commands. With these commands, you can copy, rename, move, or erase files. You can use DOS wildcards as part of the file name—just as you would on the DOS command line—to list multiple files. (Refer to Appendix A of the *User's Guide* if you need information about DOS wildcards.) The File Manager menu also displays your current directory path and offers a Change Directory command, which allows you to change the current directory, and a List Directory command, which lists the files in your current directory according to your specifications.

Practice

In this lesson, you'll start Sprint and load the advanced user interface (if you haven't loaded it already). Then, you'll use the Sprint File Manager to make sure a file (the work order CABINET.SPR) is in your current directory; if the file isn't there, you'll use the File Manager again to copy it from the directory and path where SP-SETUP copied it to your current directory. After opening CABINET.SPR, you'll split the screen into two windows and create a new file (SMITH.SPR) in the second window.

Starting Sprint and Opening an Existing File

1. Start Sprint by typing `SP` on the DOS command line. (Two-floppy system users: put the Sprint Program Disk in Drive A and your Data Disk in Drive B, then type `SP` to start Sprint.)

2. Now, choose Customize/User Interface/Load to load the advanced user interface (SPADV) that you need to work through this tutorial:

   Press: *F10 CUL* to choose Customize/User Interface/Load
   Press: the arrow keys to choose SPADV
   Press: *Enter*

   Note to two-floppy system users: If you have a low-density drive (360K), your Program Disk can only hold one user interface at a time. The simplest way to load the advanced user interface is to run SP-SETUP and choose the advanced user interface as your default. Or, if you want to load the advanced user interface from inside Sprint using Customize/User Interface/Load, you first need to run SP-SETUP and let it copy the advanced user interface (as an *alternate* user interface; that is, as an alternate to whatever default user interface SP-SETUP placed on your Program Disk) to a separate disk (see "Choosing a User Interface" in Chapter 1 of the *User's Guide* for details on how to do this). Then, when you want to load the advanced user interface, insert the disk that contains it in Drive A before you choose Customize/User Interface/Load. You can also insert the distribution disk that contains the advanced user interface (SPADV.UI) into Drive A before choosing Customize/User Interface/Load.

   If you have a high-density drive (720K or more), you have room for more than one user interface on your Program Disk. So, if the advanced user interface was among any alternate user interfaces you chose when you ran SP-SETUP, you'll see it listed when you choose Customize/User Interface/Load with your Program Disk in Drive A. Otherwise, if you want to load the advanced user interface, you must run SP-SETUP again and choose it either as your default user interface or as an alternate—depending on how much you plan to use it. See "Choosing a User Interface" in Chapter 1 of the *User's Guide* for details.

3. Use the File Manager to make sure the file CABINET.SPR is in the current directory before trying to open it. To do this, choose File/File Manager/List Directory and ask to see a list of all files with the .SPR extension (the Sprint default) in your current directory:

   Press: *F10 FFL*

   Sprint prompts `Files to list:`

   Press: *Enter* to see a list of all files with the .SPR extension

   Sprint lists all files with the .SPR extension in the current directory.

   Tip: If you want to see a list of all files in the current directory, type the DOS wildcard for "all files" (`*.*`) at the prompt.

4. If CABINET.SPR is not present, choose **File/File Manager/Duplicate-Copy**. When Sprint prompts `File to copy:`, specify the target file like this:

   a. **Hard Drive Users:**

      i. Enter the path of the directory where SP-SETUP copied the Sprint files and the file name—for example, `C:\SPRINT\CABINET.SPR`—or place the distribution disk that contains CABINET.SPR in Drive A and type `A:CABINET.SPR`.

      ii. When Sprint prompts for the target path, type your current directory path (for example, `C:\SPRINT`).

      Sprint returns your cursor to its previous position in your file when the copy process is complete. For information about paths and directories, refer to Appendix A, "A DOS Primer," in the *User's Guide.*

   b. **Floppy-Drive Users:**

      When you installed Sprint with the SP-SETUP program, it automatically placed all the tutorial files you might need on the Data Disk it created in Drive B. If for some reason you cannot find the file CABINET.SPR, make sure the Data Disk in Drive B is the one created by the SP-SETUP program.

5. Now that you're sure CABINET.SPR is in your current directory, open the file with the **File/Open** command:

   Press: *F10 FO* (or use the shortcut *Ctrl-F3*)

   Sprint prompts `File to open:`

   Type: `CABINET` (you don't need to add the Sprint default file extension .SPR)

   Press: *Enter*

   Sprint opens the file CABINET.SPR, which is a work order for building some cabinets. This file doesn't yet contain any typefaces or formatting commands; you'll be adding those in upcoming lessons.

Opening a Window and a New File

1. Choose the **Window/Open** command:

   Press: *F10 WO* (or use the shortcut *Shift-F3*)

   Tip: See Table 1.1 on page 13 for other window shortcuts.

The screen is now split into two windows, each containing the file CABINET.SPR. Your cursor is in the bottom window, which means this window is the *active window.*

2. Create a new file called SMITH.SPR in the bottom (*active*) window by choosing File/New:

   Press: *F10 FN* (or use the shortcut *Ctrl-F3*)

   Sprint prompts `File to create:`

   Type: `SMITH` (Sprint automatically adds the default extension .SPR)
   Press: *Enter*

   Your screen now displays two windows. CABINET.SPR, which contains text you want to copy into your new file, appears in the top window; your new, empty SMITH.SPR file appears in the bottom window.

The following table summarizes the tasks presented in Lesson 1:

Table 1.2: Tasks in Lesson 1

Task	Action
List files in a directory	Choose File/File Manager/List Directory and specify the path of the directory whose files you want to list (you can use DOS wildcards; see "A DOS Primer" in the *User's Guide* for details).
Load advanced user interface	Choose Customize/User Interface/Load, then choose SPADV and press *Enter.*
Copy a file to/from another directory	Choose File/File Manager/Duplicate-Copy and type the path and/or file name of the file(s) you want copied.
Open a window and a new file	Choose Window/Open, then choose File/New and enter a file name.

This completes Lesson 1. The next step is to copy some text from the CABINET.SPR file to the SMITH.SPR file. Go on to Lesson 2 for instructions.

Lesson 2: Copying Text between Windows

Block Select Commands and Windows

In Quick Start, you learned how to select a block of text and use block commands to manipulate text in one file. Once you learn how to move between open windows, you can easily perform any of the block commands on the Edit menu (Move-Cut, Insert-Paste, and Copy, for example) on multiple files.

In this lesson, you'll learn how to use a new Window command (Window/Next) with the Block Select and Edit commands you learned in Quick Start. In this way, you'll learn how to select a block of text in one open window and copy it into a file in the other open window.

Practice

Follow the instructions below to switch back to the top window, select text from the file displayed in that window (CABINET.SPR), and copy the text to the Clipboard. Then, switch back to the window with the new file (SMITH.SPR) and paste the text from the Clipboard into the new file.

Copying Text from One Window to Another

1. Return to the CABINET.SPR window by choosing Window/Next:

   Press: *F10 WN* (or use the shortcut *Shift-F6*)

2. The section you need to copy is *CABINET STYLE*. Instead of paging through the file, you can quickly search for these words with the Find command:

   Press: *F7* or *Ctrl-QF*

   Sprint prompts `Forward search:`

   Type: `CABINET STYLE`
   Press: *Enter*

   Sprint's search starts at the current cursor position; it highlights the words *CABINET STYLE* when it finds them.

3. Select the text from the start of the words *CABINET STYLE* to the bottom of the file:

   Press: *Home* to move the cursor to the start of the line
   Press: *F3* (the shortcut for Turn Select Mode On)
   Press: *Ctrl-PgDn* to move the cursor to the end of the file

Sprint highlights everything from the beginning of the words *CABINET STYLE* to the end of the file. That block of text can now be deleted, moved, or copied.

4. Copy the text into temporary storage (the Clipboard):

   Press: *F4* (the shortcut for **Edit/Copy**)

5. Switch to the window containing the SMITH.SPR file (which is empty except for a ruler line) by choosing **Window/Next**:

   Press: *F10 WN* (or use the shortcut *Shift-F6*)

6. Paste the text from the Clipboard into the new file:

   Press: *F6* (the shortcut for **Edit/Insert-Paste**)

   The block you selected is copied from CABINET.SPR and pasted into SMITH.SPR.

You now have most of the text you want; however, you still need some text from the proposal done in Quick Start (PROPOSAL.SPR). So the next thing to do is select a portion of the PROPOSAL.SPR file and copy it into your new work order, SMITH.SPR.

1. Return to the other window (the one containing CABINET.SPR):

   Press: *Shift-F6*

2. Open the file PROPOSAL.SPR, which is the file you modified in the Quick Start tutorial:

   Press: *Ctrl-F3*

   Sprint prompts `File to open:`

   Type: `PROPOSAL` (Sprint supplies the default .SPR extension)
   Press: *Enter*

   Note: When you open a new file in a window, the file it replaces in the window remains open and easily accessible, even though it's no longer visible. You can always check which files are open by choosing **File/Pick from List** (the shortcut is *Ctrl-F9*); Sprint will display a list of every open file. To display any file in the list in your active window, choose it with the arrow keys and press *Enter*. (If you prefer to switch between open files without viewing a list, just press *Ctrl-F6* and Sprint will move from file to file in the same order as the **Pick from List** command displays them.)

3. Place the cursor at the beginning of the *TASK* heading and select the text down to but not including the *COST ESTIMATE* heading:

   Press: *F3* to turn Select mode on
   Press: *F7* (the shortcut for **Edit/Search/Find**)

   Sprint prompts `Forward search:`

Type: `COST ESTIMATE`
Press: *Enter*

Sprint finds *COST ESTIMATE* and highlights everything between your starting cursor position and the end of the *COST ESTIMATE* line. Press *Home* to un-select the the line *COST ESTIMATE,* which you don't want to copy.

4. Copy the selected block into the Clipboard:

   Press: *F4*

5. Switch back to the window containing SMITH.SPR:

   Press: *Shift-F6*

6. Move the cursor to the top of the file (make sure not to move it above the ruler, though):

   Press: *Ctrl-PgUp*

7. Paste the text from the Clipboard:

   Press: *F6*

8. You've now completed copying text for the moment, so you'll get some more room to work by closing the other window. Switch to the window containing the proposal file, and close that window:

   Press: *Shift-F6*
   Press: *F10 WC* (or use the shortcut *Shift-F4*)

   Remember, you're not closing the proposal file; you're just closing the *window* showing you this file. The proposal file remains open until you deliberately close it with *Ctrl-F4* or the File/Close command.

The following table summarizes the tasks presented in Lesson 2:

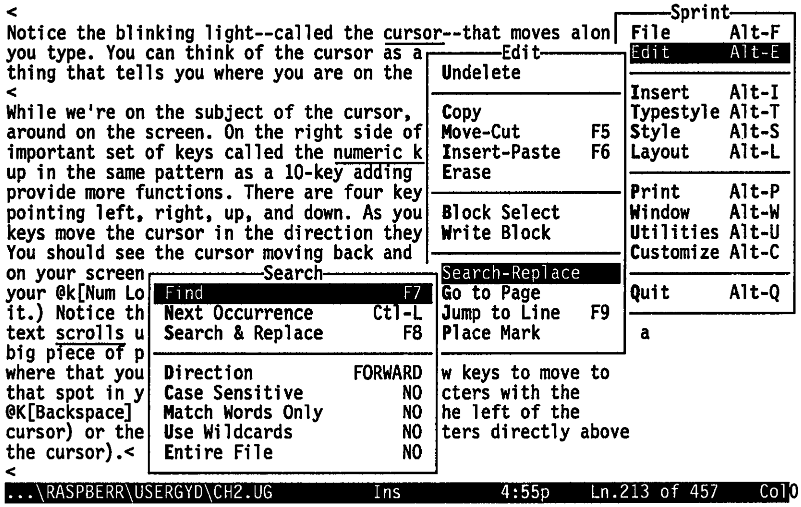
Table 1.3: Tasks in Lesson 2

Task	Action
Copy text between windows	Select the text you want to copy in one window, press *F4* to copy, and switch windows with the shortcut *Shift-F6*. Then, position your cursor where you want to insert the block and press *F6*.
Switch between open files with your cursor in one window	Choose **File/Pick from List**, use the arrow keys to choose a file name from the list, and then press *Enter* to display the file in the active window. (The shortcut for **Pick from List** is *Ctrl-F9*.)

This completes Lesson 2. The next step is to modify the work order.

Lesson 3: Search and Replace Operations

Besides the search option you've already used (the shortcut *F7* for a forward search), Sprint also allows you to search for and replace text using the Search & Replace command from the Search-Replace menu. To reach this menu, shown in Figure 1.2, choose Edit from the main menu and press *S* for Search-Replace. You can also use the shortcut *F8*.



```
<                                                           Sprint
Notice the blinking light--called the cursor--that moves alon  File      Alt-F
you type. You can think of the cursor as a          Edit       Edit      Alt-E
thing that tells you where you are on the   Undelete
<                                                              Insert    Alt-I
While we're on the subject of the cursor,   Copy               Typestyle Alt-T
around on the screen. On the right side of  Move-Cut      F5   Style     Alt-S
important set of keys called the numeric k  Insert-Paste  F6   Layout    Alt-L
up in the same pattern as a 10-key adding   Erase
provide more functions. There are four key                     Print     Alt-P
pointing left, right, up, and down. As you  Block Select       Window    Alt-W
keys move the cursor in the direction they  Write Block        Utilities Alt-U
You should see the cursor moving back and                      Customize Alt-C
on your screen             Search           Search-Replace
your @k[Num Lo  Find                    F7  Go to Page         Quit      Alt-Q
it.) Notice th  Next Occurrence      Ctl-L  Jump to Line  F9
text scrolls u  Search & Replace        F8  Place Mark          a
big piece of p
where that you  Direction          FORWARD  w keys to move to
that spot in y  Case Sensitive          NO  cters with the
@K[Backspace]   Match Words Only        NO  he left of the
cursor) or the  Use Wildcards           NO  ters directly above
the cursor).<   Entire File             NO
<
...\RASPBERR\USERGYD\CH2.UG       Ins          4:55p    Ln.213 of 457    Col0
```

Figure 1.2: The Search-Replace Menu

When you choose Search & Replace, Sprint prompts

```
Search for:
```

Type the string you want to replace and press *Enter*; Sprint prompts

```
Replace with:
```

Enter the desired replacement string. Each time Sprint finds the search string, it highlights the string and displays a small menu. You choose one of the commands on the menu to indicate that you want to replace the string (Yes), ignore it (No), or globally replace all occurrences (And the Rest) without individual confirmation. Be careful when you decide to use And the Rest; you must be sure that you want to replace the search string in *every* case.

Another command you'll use in this lesson is Entire File, which toggles (switches) from Yes to No. Setting Entire File to Yes is useful when you want to search for a string throughout a file; when Entire File is set to No, your search operations will start at your current cursor position and continue to the end of the file.

For information on using search options like wildcards and case, see the "Search and Replace" section of the "Editing: Tips, Tricks, and Techniques" chapter in the *User's Guide.*

Practice

Assume there is a minor change to the cabinet materials: The old work order specified oak veneer for the exterior, while the new job calls for birch veneer. You'll change all the references to oak veneer in the old work order to birch veneer in the file SMITH.SPR by searching the file for the word "oak" and replacing it with "birch."

Searching and Replacing Text

1. Set Edit/Search-Replace/Entire File to Yes to let Sprint search the entire file (no matter where the cursor is) for the word *oak,* then choose the Search & Replace command:

   Press: *Alt-ES* to choose Search-Replace
   Press: *F* to toggle Entire File to Yes (No is the default)
   Press: *S* to choose Search & Replace

   Sprint prompts `Search for:`

2. Enter your search string:

   Type: `oak`
   Press: *Enter*

   Sprint prompts `Replace with:`

3. Enter your replacement string:

   Type: `birch`
   Press: *Enter*

   Sprint finds the first occurrence of the word *oak* and displays the Replace This? menu choices: Yes, No, and And the Rest.

   Tip: To stop a search at any time, just press Ctrl-U, the "abort" key; Sprint will return to your original cursor position.

4. You want to replace *oak* with *birch* in this case, so choose Yes from the Replace This? menu:

   Press: *Y*

Sprint replaces *oak* with *birch* and continues to the next occurrence of *oak*.

Note: You don't want to use the And the Rest option this time, since that would automatically replace all occurrences of the word *oak* with the word *birch*. This is not what you want to do, as you'll see shortly.

5. The next two occurrences of *oak* also involve veneer, so answer Yes twice:

   Press: *Y* each of the two times Sprint asks for confirmation

   Sprint replaces *oak* with *birch* twice. Again, the search continues.

6. The next occurrence of *oak* involves the Shaker knobs, which you are not replacing for this work order, so answer No:

   Press: *N*

   Sprint leaves this occurrence of *oak* as it stands and continues the search. Since this is the last occurrence of *oak*, Sprint returns to the place you started the search and tells how many times you replaced the word.

The following table summarizes the search-and-replace task presented in Lesson 3:

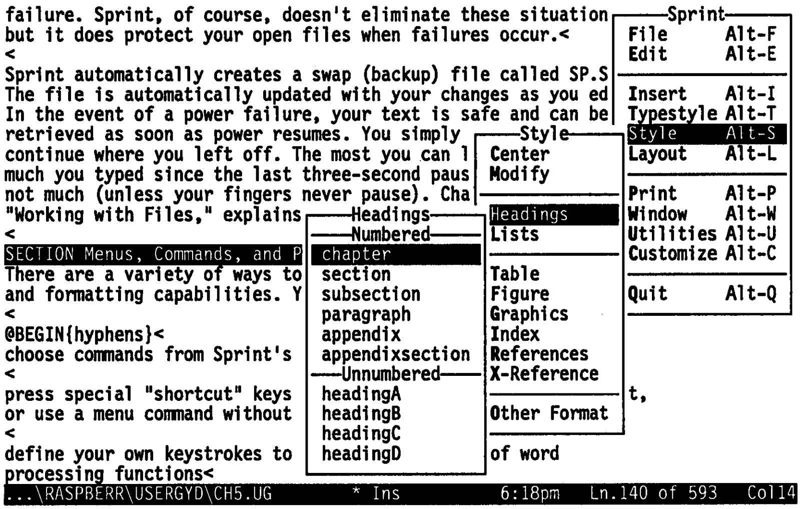
Table 1.4: Task in Lesson 3

Task	Action
Search and replace a string	Choose Edit/Search-Replace/Search & Replace (or press *F8*), enter the search string, and press *Enter*. Enter the replacement string and press *Enter*. Then, each time Sprint finds an occurrence of the search string and asks for confirmation, reply either Yes, No, or And the Rest.

This completes Lesson 3. Assuming that these are the only changes you need to make to the actual text of the document, you're now ready to improve the appearance of the document. You start that process in the next lesson.

Lesson 4: Adding Section Headings

One of the best ways to improve a plain document is to emphasize the headings of different sections. You've already learned about the different typestyles Sprint offers for emphasis; in this lesson, you'll use commands from the Headings menu (Figure 1.3), which lists a variety of heading formats for text.



```
failure. Sprint, of course, doesn't eliminate these situation
but it does protect your open files when failures occur.<
<
Sprint automatically creates a swap (backup) file called SP.S
The file is automatically updated with your changes as you ed
In the event of a power failure, your text is safe and can be
retrieved as soon as power resumes. You simply
continue where you left off. The most you can l
much you typed since the last three-second paus
not much (unless your fingers never pause). Cha
"Working with Files," explains
<
SECTION Menus, Commands, and P
There are a variety of ways to
and formatting capabilities. Y
<
@BEGIN{hyphens}<
choose commands from Sprint's
<
press special "shortcut" keys
or use a menu command without
<
define your own keystrokes to
processing functions<
...\RASPBERR\USERGYD\CH5.UG          * Ins          6:18pm   Ln.140 of 593   Col14
```

Figure 1.3: The Headings Menu

You can choose either numbered or unnumbered headings. If you choose a Numbered heading command, Sprint automatically numbers the heading and creates a table of contents for the document listing all the headings you inserted in the file and the page numbers where they appear. If you move the heading and section to another place in the document, Sprint automatically updates both the heading number and the table of contents!

For example, suppose you want to create numbered section headings for a paper on economics, starting with a section called "Free Market System." To create the first numbered section heading, you'd choose Headings/ Section from the Style menu. Sprint prompts for the heading title, so type `Free Market System` and press *Enter*. Onscreen, the heading looks like this:

```
SECTION Free Market System
```

When you print the document, Sprint automatically numbers the heading as 1 (since it's the first in the document), and prints the heading number and heading text in a large font (if your printer has one) and a bold typestyle. Two blank lines will appear above and below the heading, like this:

1. Free Market System

If you choose Subsection from the Style/Headings menu, Sprint prints a second-level numbered heading. This heading also prints in a large, bold font, but the type size is somewhat smaller than the Section command. A single blank line appears above and below a Subsection heading. For example, a Subsection heading for a subsection of "Free Market System" called "Supply and Demand" would print like this:

1.1 Supply and Demand

Note: When you use numbered headings with a Chapter command, a Section command prints as 1.1., a Subsection command prints as 1.1.1, and so on, like this:

1.1 Free Market System

1.1.1 Supply and Demand

Also Note: If you don't want your headings numbered, choose Unnumbered headings; for example, Headings/HeadingA. You won't get a table of contents, however, unless you do one of the following:

- Choose a Numbered heading somewhere in your file (that is, Chapter, Section, Subsection, Paragraph, Appendix, or AppendixSection), in which case Sprint will create a table of contents and will include all headings (numbered and unnumbered) in the table.
- Move the cursor to the top of the file (below the ruler), choose Style/ Other Format, and type `MAKETOC`. Sprint prompts

```
Type (R) for Region or (C) for Command:
```

- Type `C`, and Sprint will insert the `MAKETOC` (make a table of contents) command in your file.

Note: Heading formats are defined in the STANDARD.FMT file. As with all command definitions listed in STANDARD.FMT, they can be changed to suit your particular word-processing needs. See the "Advanced Formatting" section of this manual for information about the STANDARD.FMT file.

Refer to the "Headings" section on page 78 of Chapter 2 for more information about headings.

Practice

You'll insert numbered Section headings for each section heading in the work order, and then assign that command to a single keystroke.

Choosing Heading Commands

1. Move the cursor to the *1 TASK* (line 2, column 0).
2. Choose Section from the list of Numbered headings:

   Press: *F10 SH* to choose Style/Headings
   Press: the arrow keys to choose Section
   Press: *Enter*

3. When you've created a heading for *TASK*, choose the Section command for each of the five remaining section headings (*TIME ESTIMATE, CABINET STYLE, MATERIALS, FINISHING INSTRUCTIONS,* and *SUPPLIERS*):

   Press: *F7* (shortcut for Find command)
   Type: `TIME ESTIMATE` at the prompt
   Press: *Enter*

   Sprint finds the *TIME ESTIMATE* heading and highlights it.

   Press: *F10 SH* to choose Style/Headings

   The highlighted choice on the Headings menu should still be Section.

   Press: *Enter*

   You can repeat the "Press *F10 SH,* then choose Section" process for each of the remaining headings in the file after *TASK*. Before you do, however, see the next part of the lesson for another option.

Assigning a Menu Command to a Key

To make it easier to use any menu command, you can assign a shortcut key to perform the command for you. In this part of the lesson, you'll pick a keystroke combination (for example, *Alt-1*) to perform the Style/Headings/Section command.

1. Choose Style/Headings from the main menu and highlight the Section command:

   Press: *F10 SH*
   Press: the *Down arrow* key until Section is highlighted—don't press *Enter*; just highlight the command!

2. With the menus still displayed and your chosen command highlighted, you can assign any command from the menus to a key:

   Press: *Ctrl-Enter* to assign Style/Headings/Section to a key

   Sprint prompts `Shortcut for menu item:`

3. Think of a keystroke combination that's easy to remember (for example, *Alt-1*) and press those keys:

Press: *Alt-1*

Now, whenever you press *Alt-1,* Sprint will insert the Section command. For more information on assigning and saving Sprint shortcuts, see the "Sprint Utilities" chapter in the *User's Guide.*

The following table summarizes the tasks presented in Lesson 4:

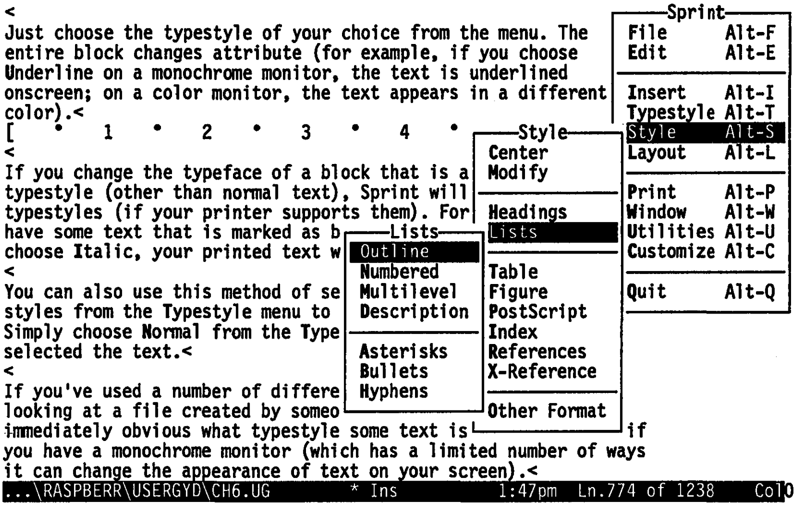
Table 1.5: Tasks in Lesson 4

Task	Action
Insert a numbered heading	Choose **Style/Headings**, then choose any of the Numbered commands listed.
Insert an unnumbered heading	Choose **Style/Headings**, then choose one of the commands listed as Unnumbered.
Make a table of contents for a file without numbered headings	At the top of the file, choose the **Style/Other Format** command and type `MAKETOC`. At the prompt, type `C` for Command.
Assign a menu command to a key	Step through the menus until you reach the command you want to assign to a key; make sure it's highlighted. Press *Ctrl-Enter* and press a key to which you want to assign the command.

This completes Lesson 4. In the next lesson, you'll format a section of the work order with the **Style/Lists/Description** command.

Lesson 5: The Description List

The Quick Start Tutorial introduced the Lists menu (Figure 1.4) and you formatted two sections of PROPOSAL.SPR with list commands: the **Numbered** command, which produces a numbered list; and the **Hyphens** command, which produces a list with items set off by hyphens.



```
<
Just choose the typestyle of your choice from the menu. The
entire block changes attribute (for example, if you choose
Underline on a monochrome monitor, the text is underlined
onscreen; on a color monitor, the text appears in a different
color).<
[   •   1   •   2   •   3   •   4   •
<
If you change the typeface of a block that is a
typestyle (other than normal text), Sprint will
typestyles (if your printer supports them). For
have some text that is marked as b
choose Italic, your printed text w
<
You can also use this method of se
styles from the Typestyle menu to
Simply choose Normal from the Type
selected the text.<
<
If you've used a number of differe
looking at a file created by someo
immediately obvious what typestyle some text is              if
you have a monochrome monitor (which has a limited number of ways
it can change the appearance of text on your screen).<
...\RASPBERR\USERGYD\CH6.UG        * Ins         1:47pm  Ln.774 of 1238    Col0
```

Figure 1.4: The Lists Menu

There are two ways to use the commands on the Lists menu; the first is suited to formatting existing text, the second to setting up a list as you're entering the text:

- Select a block of text using *F3* or one of the commands on the Edit/Block Select menu, then choose a Lists command. Sprint inserts BEGIN and END commands for the list format before and after the selected block.
- Choose a command from the Lists menu and press *B* for Begin command; Sprint inserts a BEGIN command for the list format. Type the text of the list, making sure the items are double-spaced. Then choose the same List command and press *E* for End command; Sprint inserts an END command for the list format in your text.

The Description command, which is used in this lesson, creates a two-column list; the column on the left (which prints in a bold typestyle) typically contains a "subject," and the column on the right describes the subject. You separate the subject from the descriptive text by pressing the *Tab* key. The descriptive text is automatically indented 1/4 of the line length (for example, if the lines in your text are 6 inches long, the descriptive text will be indented 1.5 inches).

Practice

The *CABINET STYLE* section of the work order lends itself to a Description format. *Face frames* and *Doors* are the subjects in this list; the text to the right of the subjects describes the style of the cabinet components.

Creating a Description List Format

1. Press *F3* to select the text immediately following the *CABINET STYLE* heading, down to but not including the *MATERIALS* heading, and then choose the Style/Lists/Description command:

   Press: *F10 SLD*
   Press: *Enter*

   Sprint inserts BEGIN and END DESCRIPTION commands around the text you selected.

2. Now replace the two " – " (space, hyphen, space) character strings in this section with a *Tab* using the Edit/Search-Replace/Search & Replace command (shortcut *F8*):

   Press: *F8*

   Sprint prompts `Search for:`

   Type: [space][hyphen][space]
   Press: *Enter*

   Sprint prompts `Replace with:`

   Press: *Tab*
   Press: *Enter*
   Press: *Y* for Yes twice to replace both occurrences of " – " in this section with a *Tab*

   Sprint will continue to find the " – " string after the END DESCRIPTION command; don't replace these. Just press *Ctrl-U* (the "abort" key) to escape from the Replace This? menu and return to your previous cursor position. This is how your onscreen text should look now:

   ```
   BEGIN DESCRIPTION
   Face frames   Simple, edges chamfered, built to suit kitchen plan
   signed-off by owner
   Doors         1/2" plywood, exterior covered with 1/4" birch
   veneer, edges routed with Bit #32
   END DESCRIPTION
   ```

   The text looks a little strange on the screen, but when you print the file the subjects *Face frames* and *Doors* will appear at the left margin, in boldface, with the descriptive text indented, something like this:

Face frames Simple, edges chamfered, built to suit kitchen plan signed-off by owner

Doors 1/2“ plywood, exterior covered with 1/4” birch veneer, edges routed with Bit #32

The following table summarizes the list format task presented in Lesson 5:

Table 1.6: Task in Lesson 5

Task	Action
Insert a two-column list	Select the block of text you want to format as a list. Make sure the items are double-spaced. Choose Style/Lists/Description, press *Enter,* and then insert a *Tab* to separate the two columns.

This completes Lesson 5. Now assume you want to insert an illustration showing the kitchen plan referenced in the *Face frames* description. The following section explains how to leave room for this figure in your text.

Lesson 6: Figures and Tables

In this lesson, you'll learn how to insert figures into your files. The Figure command on the Style menu automatically assigns a number to the figure and lets you create an optional caption. Use the Page Breaks/Reserve Space command on the Layout menu to allow blank space for a figure in your printed document (if desired). When you opt to give your figure a caption, the formatter automatically assigns a number to the figure and references the figure number, the caption, and the page number on a special List of Figures page (part of the Table of Contents).

When you choose the Style/Figure command, Sprint first prompts for the caption. The caption is optional; if you choose not to have one, just press *Enter.* Sprint inserts the BEGIN and END FIGURE commands with your cursor between them. If you typed in a figure caption at the prompt, it inserts the CAPTION command on the line below your cursor. A List of Figures page will be generated only if you give the figure a caption; if you do, the figure will be referred to by assigned number, caption, and page number.

You can specify the desired amount of blank space required for the figure on the line after the BEGIN FIGURE command. You can always press the *Enter* key a number of times to create the blank space, but the Layout/Page Breaks/Reserve Space command lets you specify more precise dimensions for the figure (using, for example, inches, points, centimeters, lines, or a portion of a page).

By definition, the Figure command tells Sprint to immediately begin the figure format. You can, however, force figures to begin at the top or bottom of a page. To do this, you need to modify the Figure command to include either the *above* or *below* parameter. Lesson 16, beginning on page 60, explains how to do this.

The Table command on the Style menu works the same way as the Figure command; it prompts for an optional table caption, which generates a List of Tables page in the Table of Contents. You can also modify a Table format to automatically begin at the top or bottom of a page (see Lesson 16, beginning on page 60).

Practice

First, you'll create a figure caption and reserve space in your work order for a figure (in this case, a manually sketched illustration of the kitchen plan to be pasted in later). Then, you'll format a block of text in the file with the Style/Table command.

Inserting a Figure

1. Move the cursor to the line above *SECTION MATERIALS* and press *Enter.*
2. Choose **Style/Figure** from the main menu. When Sprint prompts for a caption, enter the figure caption:

   Press: *F10 SF* for Style/Figure

   Sprint prompts `Caption:`

   Type: `Smith Kitchen Plan`
   Press: *Enter*
3. Now tell Sprint how much space to reserve for the figure. For this exercise, let's assume the kitchen plan takes up 2.5 inches of space. With your cursor positioned on the line below the BEGIN FIGURE command, choose the Layout/Page Breaks/Reserve Space command.

   Press: *F10 LPR*
   Type: `2.5 inches`
   Press: *Enter*

   The group of commands for the figure will appear onscreen like this:

```
BEGIN FIGURE
RESERVE 2.5 INCHES
CAPTION Smith Kitchen Plan
END FIGURE
```

When you print the file, Sprint will leave 2.5 inches blank, and then print the caption Figure 1: Smith Kitchen Plan (the figure number will be "1" because it's the first in the file). The figure number and caption print in small type, centered between the left and right margins (see page 10).

If there aren't at least 2.5 inches of blank space remaining on the page when Sprint sees the BEGIN FIGURE command, Sprint will break the page and leave room for the figure and the figure caption at the top of the next page.

Creating a Table

1. Select the text under the *MATERIALS* heading, down to but not including the *FINISHING INSTRUCTIONS* heading, and then choose Style/Table. Sprint prompts you for a table caption:

   Type: `Required Materials for Smith Job`
   Press: *Enter*

   Sprint inserts BEGIN and END TABLE commands and the TCAPTION command in the file. When you print, Sprint automatically numbers the table and places the table number and caption in the Table of Contents.

2. The group of commands for the table will appear onscreen like this:

```
BEGIN TABLE
TCAPTION Required Materials for Smith Job
3/4" face frames - solid birch - 25 each
1/4" veneer exterior - birch - 2 sheets
Drawer slides - 502-436 - 10 pair
Drawer pulls - 1" oak shaker knobs - 10 each
Hinges - Brassware 237 - 12 each
END TABLE
```

The following table summarizes the tasks presented in Lesson 6:

Table 1.7: Tasks in Lesson 6

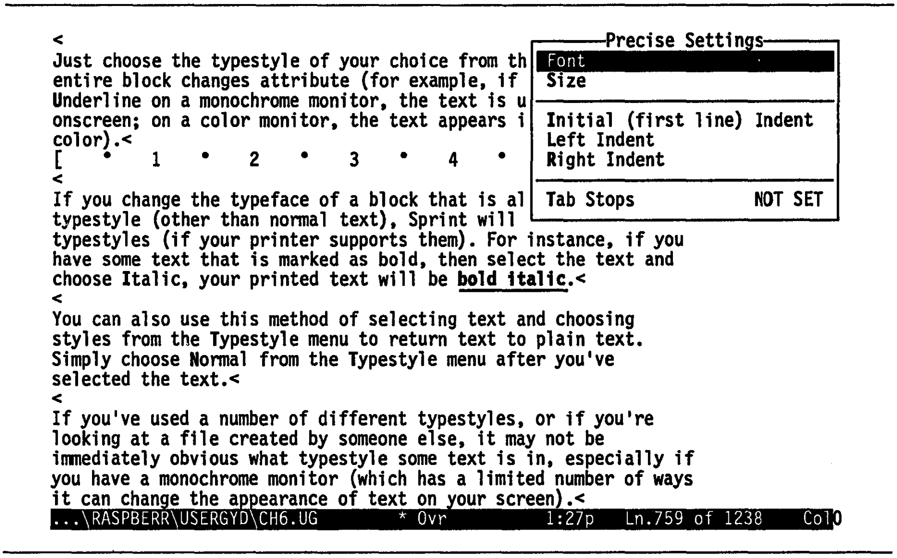
Task	Action
Create a figure	Choose Style/Figure, then specify a caption or press *Enter* for no caption.
Create a table	Choose Style/Table, then specify a caption or press *Enter* for no caption.
Reserve blank space	Choose Layout/Page Breaks/Reserve Space, then specify the amount of space in inches, lines, points, centimeters, or a portion of a page.

This completes Lesson 6. In the next lesson, you'll format the text of this table with precisely measured tab stops.

Lesson 7: Precise Ruler Settings

You've already learned how to justify text and set margins, tabs, and indents on the ruler line: choose Layout/Ruler/Edit on Screen, move to a column on the ruler, and enter the appropriate ruler editing code (see "Editing the Ruler" in Chapter 4 of the *User's Guide* for details on how to do this). However, there may be times when you want to set tabs or margins other than with column numbers. For example, you may want to set tabs 1, 2, and 4 inches from the left margin, or offset the left margin by 4 picas in an area of your file. The Layout/Ruler/Precise Settings menu (see Figure 1.5) lets you be more precise with your formatting dimensions after the ruler line.

Note: If your printer doesn't support proportionally spaced fonts, your results may not be identical to the example shown on page 11.



```
<
Just choose the typestyle of your choice from th
entire block changes attribute (for example, if
Underline on a monochrome monitor, the text is u
onscreen; on a color monitor, the text appears i
color).<
[    •    1    •    2    •    3    •    4    •
<
If you change the typeface of a block that is al
typestyle (other than normal text), Sprint will
typestyles (if your printer supports them). For instance, if you
have some text that is marked as bold, then select the text and
choose Italic, your printed text will be bold italic.<
<
You can also use this method of selecting text and choosing
styles from the Typestyle menu to return text to plain text.
Simply choose Normal from the Typestyle menu after you've
selected the text.<
<
If you've used a number of different typestyles, or if you're
looking at a file created by someone else, it may not be
immediately obvious what typestyle some text is in, especially if
you have a monochrome monitor (which has a limited number of ways
it can change the appearance of text on your screen).<
...\RASPBERR\USERGYD\CH6.UG        * Ovr          1:27p    Ln.759 of 1238     Col 0
```

Figure 1.5: The Precise Settings Menu

The **Precise Settings** commands affect all text under the ruler line *until you insert a new ruler*. So, at the point where you want to return to the settings on the default ruler, use **Layout/Ruler/Insert** to insert another default ruler.

Font	Allows you to choose from a list of fonts your printer supports
Size	Lets you specify any type size your printer supports
Initial (first line) Indent	Lets you specify how much the first line of any paragraph should be indented (the default is 0)
Left Indent	Offsets the text from the left margin
Right Indent	Offsets the text from the right margin
Tab Stops	Allows you to specify tab stops

You can use any unit of measurement for **Left Indent**, **Right Indent**, or **Tab Stops** when Sprint prompts you for a dimension (inches, picas, points, centimeters, and so on).

When you choose the **Font** command, Sprint displays a list of fonts for your printer. Some printers may only have two fonts, while others may have several. Pick a font other than *default* and press *Enter*. Sprint will "hide" a font code on the new ruler, instructing the formatter to print the following text in the chosen font (you can display it by pressing *Alt-Z* to show codes). You'll also see the name of the chosen font displayed next to the **Font** command on the **Precise Settings** menu.

Practice

In this lesson, you'll insert a new ruler to format the *Required Materials for Smith Job* table, choose a font that's different from your printer's default font, set precise tab stops, and offset the left and right margins from the current margin settings.

Creating Precise Ruler Settings

1. Move the cursor to the line containing the TCAPTION command and insert a blank line below it.
2. Press *Alt-R* to insert a new ruler. Delete the tab stop (`T`) preset at column 5 and type two new `T`'s on the ruler line as placeholders for the two precise tabs you'll set with the **Tab Stops** command. (The column numbers you choose aren't important; try 25 and 50.) The precise tabs will not print correctly without the placeholding tab symbols. Press *Esc* to get out of the highlighted ruler line.
3. Specify a different font in which to print the section affected by the new ruler. Choose **Layout/Ruler/Precise** Settings/**Font**:

   Press: *F10 LRPF*

   Sprint displays a list of fonts available on your printer.

   Press: the arrow keys to choose a font other than the default
   Press: *Enter*
4. Change the left margin by choosing the **Left Indent** command:

   Press: *L* for Left Indent

   Sprint prompts for the left indent value. Type `.75 inches`, which will offset the text below the ruler by .75 inches. This distance is measured from the current left margin setting (1 inch), so your text will begin printing 1.75 inches from the edge of the paper.

   If you'd rather specify the indent in *picas* (approximately 6 picas per inch) or *centimeters,* feel free to do so.
5. Change the right margin by choosing the **Right Indent** command:

   Press: *R* for **Right Indent**

   When Sprint prompts for the right indent, type `.75 inches`. This adds .75 inches to the current right margin setting.

6. Set tabs precisely measured in a dimension, not in columns, by choosing the **Tab** Stops command:

   Press: *T* to choose **T**ab Stops

   Sprint prompts `Place tabs at:`

7. Set tabs at 1.75 inches and 3.5 inches:

   Type: `1.75 inches, 3.5 inches`

   Sprint inserts a TABSET command below the ruler line and displays the precise tab settings onscreen. Press *Enter* twice to insert two blank lines (you'll need the space later on).

8. Press *Esc* to remove the menu.

   Note: The commands you chose for font, left indent, and right indent are displayed on the **P**recise Settings menu; they're also hidden on the ruler line. You can see them if you press *Alt-Z* to toggle **C**odes On.

Formatting a Table

1. Place your cursor on the line below the TABSET command and create some headings for the columns you're going to format:

   Type: `Material`
   Press: *Tab*
   Type: `Type`
   Press: *Tab*
   Type: `Quantity`
   Press: *Enter* twice to insert two blank lines

   Don't be concerned that the tabs don't appear to be set 1.75 and 3.5 inches from the left margin. They will be precisely set when you format and print the file.

2. Now you need to align the text of the materials table with the new tab stops and column headings. Using the **S**earch & **R**eplace shortcut (*F8*), replace the " – " (space, hyphen, space) characters with *Tab*:

   Press: *F8*

   Sprint prompts `Search for:`

   Type: [space][hyphen][space]
   Press: *Enter*

   Sprint prompts `Replace with:`

   Press: *Tab*
   Press: *Enter*
   Press: *Y* for **Y**es 10 times

Your text aligns onscreen because of the `T` symbols on the ruler line; when you print, however, the first *Tab* prints text 1.75 inches from the new margin you set with Left Indent, and the next *Tab* will print text 3.5 inches from the same margin. Your text should look like this onscreen:

```
BEGIN TABLE
TCAPTION Required Materials for Smith Job
+ ----------------------------------------------------------------------------------------+
[     ·     1     ·     2     ·     T     ·     4     ·     T     ·     6     ]L   7   ·
+ ----------------------------------------------------------------------------------------+
TABSET 1.75 inches, 3.5 inches

Material                            Type                  Quantity

3/4" face frames                    solid birch           25 each
1/4" veneer exterior                birch                 2 sheets
Drawer slides                       502-436               10 pair
Drawer pulls                        1" oak shaker knobs   10 each
Hinges                              Brassware 237         12 each
END TABLE
```

3. Since you don't need these precise settings for the rest of the work order text, move the cursor to the line before the *FINISHING INSTRUCTIONS* section and insert another ruler. This new ruler will look exactly like the ruler at the top of the file.

The following table summarizes the tasks presented in Lesson 7:

Table 1.8: Tasks in Lesson 7

Task	Action
Use a different font	Choose Layout/Ruler/Precise Settings from the main menu, then choose Font. Use the arrow keys to pick a font other than the default, then press *Enter.*
Set a new left margin	Choose Layout/Ruler/Precise Settings from the main menu, then choose Left Indent. Enter the desired distanced from the current left margin at the prompt, and press *Enter.*
Set a new right margin	Choose Layout/Ruler/Precise Settings from the main menu, then choose Right Indent. Enter the desired distanced from the current right margin at the prompt, and press *Enter.*
Set precise tabs	Choose Layout/Ruler/Precise Settings/Tab Stops. Enter the specified dimensions of the tabs, separating each one with a comma (for example, `8 picas, 15 picas, 22 picas`). Press *Enter.*

This completes Lesson 7. In the next lesson, you will view your formatted file and verify that your table formats correctly.

Lesson 8: Previewing the Text

In the Quick Start tutorial, you learned how to use **Print/Paginate** to preview page breaks in a file before printing.

In this lesson, you'll choose Screen Preview from the **Print** menu so you can preview your formatted file. When you choose **Print/Screen Preview**, Sprint saves your file to disk, then interprets the formatting commands you've chosen and inserted into your file and checks for any errors you may have made in entering these commands. If Sprint doesn't find any errors, it displays the file one screen page at a time, as if it were printing the file on your currently selected printer. If Sprint detects one or more errors during formatting, it displays an error message that explains the error and references a line number.

The appearance of your screen preview depends on the currently selected printer. (Usually, the most capable printer is installed as the *default* printer, and any other printer(s) as *alternate(s)*. If you're not sure how your printers were installed for Sprint, see "Before You Begin" in the *User's Guide* for installation instructions.)

If your currently selected printer (perhaps a dot-matrix) supports only fixed-pitch fonts (for example, pica, elite, or courier), the output of a screen preview will be similar to the way the printed output will appear. Most screens can only display text in fixed-width fonts (typically 10 characters per inch).

On the other hand, if your currently selected printer supports proportional spacing, various fonts, and different character sizes (like some laser printers do), the output onscreen will look strange at times. The reason is that your screen can't display literally what your printer is capable of printing. For example, most screens can't display enlarged characters (like those in a Section heading) or reduced characters (like footnotes). With proportionally spaced fonts, moreover, not all characters are the same width; when Sprint tries to display proportionally spaced output on your screen, you'll see characters overwriting each other, as if text were missing. Don't worry; this won't be the case when your file is printed.

Note to two-floppy system users: When you want to print a document, you must have the Sprint Program Disk in Drive A.

Practice

In this lesson, you'll view your formatted file onscreen as well as any error messages the formatter might generate.

Previewing Your File Onscreen

1. If you want to preview your text as if it were going to print on an alternate printer (any printer other than the *default*), you need to identify the printer you want to use. Choose the **Print/Current Printer** command and pick the printer you want to use. *Do this before you choose the Print/Screen Preview command.*

   Press: *F10 PC*

   Sprint displays a list of printers that have been installed for Sprint use.

   Press: the arrow keys to highlight the desired printer
   Press: *Enter*

2. Choose **Print/Screen** Preview.

   Sprint immediately saves the file. If no errors are found, your file diplays one screen at a time. After each page, Sprint prompts

   ```
   [Press any key for more, Esc to quit.]
   ```

   If your printer has some of the capabilities we mentioned above (like multiple fonts or character sizes), remember that the display will look strange because your screen doesn't have the same capabilities.

3. If you get an error message, note it down, and then choose **Edit/Jump** to Line (or *F9*). When Sprint prompts for the line number, enter the line number displayed in the error message.

   Tip: Sometimes, Sprint doesn't recognize an error until after it passes the line containing the error; in these, cases, the line number Sprint cites in the error message may not be precise. If you don't see anything unusual on a particular line number, start looking backwards for the offending command.

4. Correct any errors, and then choose **Print/Screen** Preview again. You don't need to save the file because **Print** commands automatically write your file to disk before formatting and printing.

   If you get several error messages, don't worry about writing them all down before you press a key to continue; go on to Lesson 13, beginning on page 52, which explains how to log all formatter/error messages to a file on disk. When you're done with that lesson, come back to this lesson, repeat the instructions for previewing your file on the screen, and then continue this tutorial.

For more information about previewing your file and checking error messages, see the "Printing" chapter in the *User's Guide.*

The following table summarizes the tasks presented in Lesson 8:

Table 1.9: Tasks in Lesson 8

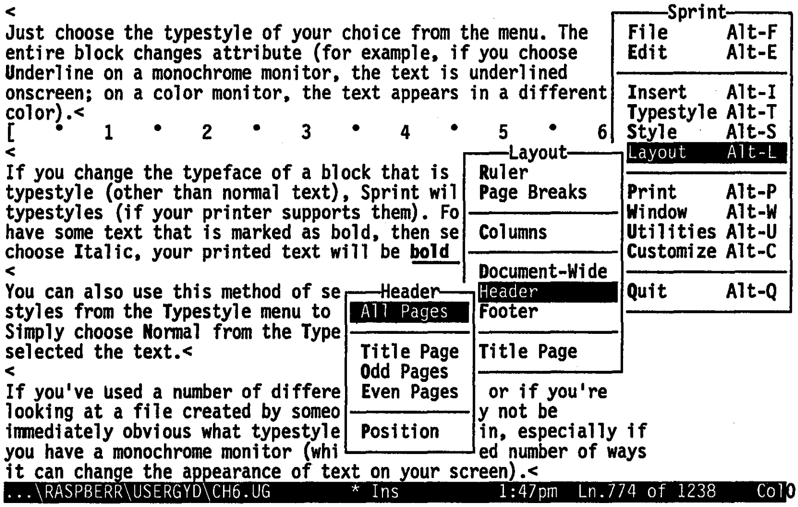
Task	Action
Switch printers	Choose the Print/Current Printer command and pick the printer you want to use with the arrow keys. Press *Enter.*
Preview your formatted file	Choose Print/Screen Preview from the main menu, or press *Ctrl-F8.*

This completes Lesson 8.

Lesson 9: Adding Headers and Footers

This part of the tutorial explains how to create running headers and footers (also called page headings and page footings). Headers appear within the top margin of each page; footers appear within the bottom margin of each page.

Sprint supports multiple-line headers and footers and lets you specify how the header and footer text should be formatted. You can place the text at the left margin, aligned at the right margin, and/or centered between the left and right margins. The commands to create headers and footers are on Layout/Header and Layout/Footer menus. The menus are identical; the Header menu is shown in Figure 1.6.



```
<
Just choose the typestyle of your choice from the menu. The
entire block changes attribute (for example, if you choose
Underline on a monochrome monitor, the text is underlined
onscreen; on a color monitor, the text appears in a different
color).<
[   •   1   •   2   •   3   •   4   •   5   •   6
<
If you change the typeface of a block that is
typestyle (other than normal text), Sprint wil
typestyles (if your printer supports them). Fo
have some text that is marked as bold, then se
choose Italic, your printed text will be bold
<
You can also use this method of se
styles from the Typestyle menu to
Simply choose Normal from the Type
selected the text.<
<
If you've used a number of differe               or if you're
looking at a file created by someo              y not be
immediately obvious what typestyle              in, especially if
you have a monochrome monitor (whi              ed number of ways
it can change the appearance of text on your screen).<
...\RASPBERR\USERGYD\CH6.UG        * Ins        1:47pm  Ln.774 of 1238    Col0
```

Figure 1.6: The Header Menu

Sprint's Header and Footer commands can appear anywhere in your document. By default, the text of the page header or footer will appear on all pages following the command except the first page.

As you can see in Figure 1.6, both the Header and Footer menus offer the All Pages command. When you choose All Pages, Sprint inserts BEGIN and END HEADER commands (for a header) or BEGIN and END FOOTER commands (for a footer) into your file; you just enter the text of the header or footer between those commands. The All Pages command prints the header or footer you create on every page *except* the first. If you want your header or footer to print on every page *including* the first, you must also choose Title Page and enter the same header or footer text as you did for All Pages. You can also use Title Page to create a special header or footer for the first page *only*.

Two other commands the Header and Footer menus share are Odd Pages and Even Pages: choose Header/Odd Pages and Header/Even Pages to print different headers on odd and even pages (as we do in this manual); choose Footer/Odd Pages and Footer/Even Pages to do the same for your footers.

The headers and footers you create can be positioned on the page with the Position command; specify the exact distance from the top of the page (headers) or the bottom of the page (footers) in any dimensions your printer supports.

By default, Sprint automatically prints the page number in the footer line of every page; you need not choose a Footer command. However, if you do choose a **Footer** command, you override this default function. If you want page numbers in your footer, you need to tell Sprint where to place them.

Specifying a page number is a little abstract because it involves the concept of *variables*, discussed in the "Advanced Formatting: Tips, Tricks, and Techniques" chapter of this book. To print page numbers, you need to insert the variable *page* and tell Sprint to print its current value.

Practice

You'll create a page heading for the Smith work order that contains the name of the job and the date. The page footing will contain the company name (Remodeling, Inc.) on the left, the words `Work Order` in the center, and the page number on the right.

Inserting a Header

1. Move the cursor to the top of the file (but stay below the ruler line).
2. Choose **Layout/Header/All** Pages; this tells Sprint to print a header on every page *except* the first:

   Press: *F10 LHA*

   Sprint inserts the following commands into your document:

   ```
   BEGIN HEADER
   END HEADER
   ```
3. With the cursor between the BEGIN and END HEADER commands, enter the job name:

   Type: `Smith Job (Cabinets)`
4. Choose **Insert/Wide** Space (Spring) to force the rest of the header (the date) against the right margin, and then type the date:

   Press: *F10 IW* to force what you type next to the right margin
   Type: `May 2, 1988`
   Press: *Enter*
5. Immediately below the END HEADER command, create a header for the first page with **Header/Title** Page:

   Press: *F10 LHT*

   Sprint inserts the following commands into your document:

```
BEGIN HEADERT
END HEADERT
```

6. Enter the same header information you entered for **Layout/Header/All Pages** in order to get a first page header identical to the one that appears on all the other pages.

Inserting a Footer

1. On the line directly below the END HEADERT command, choose **Layout/Footer/All Pages** and proceed as you did to insert a header (see above).

   Press: *F10 LFA*

   Sprint inserts the following into your document:

   ```
   BEGIN FOOTER
   END FOOTER
   ```

2. Enter the company name, *Remodeling, Inc.*, between the BEGIN and END FOOTER commands, choose **Insert/Wide Space (Spring)**, and enter the words `Work Order`:

   Type: `Remodeling, Inc.`
   Press: *F10 IW* to force what you type next to the right margin
   Type: `Work Order`

3. After the words *Work Order* in the footer, insert another wide space with **Insert/Wide Space (Spring)**:

   Press: *F10 IW*

4. Now, to insert the page number, choose **Insert/Variable** and choose *Page*. (This insertion will force the words *Work Order* back towards the center of the page. See the footer on page 11 for the way this prints.)

   Press: *F10 IV*

   Sprint displays the list of available variables.

   Press: arrow keys to choose *page*
   Press: *Enter*

   Page is a variable; its value changes each time Sprint begins a new page. When you insert the *Page* variable, you're telling Sprint to determine the value of *Page* each time it prints a page of your file, and insert the correct page number in the footer.

   Once you choose *Page*, you'll see a menu that lets you pick a template for *how* Sprint should print the page number (for example, in arabic numbers, roman numerals, and so on).

5. Choose arabic (if you want to try a different numbering template, choose another from the list). Sprint inserts the page variable and codes for the template you chose.
6. On the line immediately below the END FOOTER command, create a footer for the first page with Footer/Title Page:

   Press: *F10 LFT*

   Sprint inserts the following commands into your document:

   ```
   BEGIN FOOTERT

   END FOOTERT
   ```
7. Enter the same footer information you entered for **Layout/Footer/All Pages** in order to get a first page footer identical to the one that appears on all the other pages.
8. Your header and footer commands should look like this onscreen:

   ```
   BEGIN HEADER
   Smith Job (Cabinets)                                May 2, 1988
   END HEADER
   BEGIN HEADERT
   Smith Job (Cabinets)                                May 2, 1988
   END HEADERT
   BEGIN FOOTER
   Remodeling, Inc.            Work Order            PAGE, t="%d"
   END FOOTER
   BEGIN FOOTERT
   Remodeling, Inc.            Work Order            PAGE, t="%d"
   END FOOTERT
   ```
9. Using the instructions provided in Lesson 8, *Previewing Your Text,* view your formatted file, and verify that your header and footer lines print correctly.

The following table summarizes the tasks presented in Lesson 9:

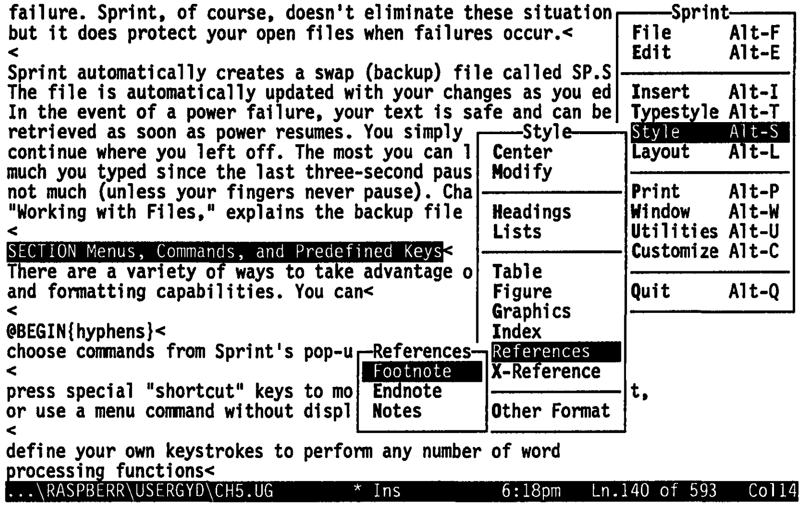
Table 1.10: Tasks in Lesson 9

Task	Action
Insert a header	Choose Layout/Header/All Pages from the main menu. Type the text of the header between the BEGIN and END HEADER commands Sprint inserts. To create a header for the first page, choose Layout/Header/Title Page.
Insert a footer	Choose Layout/Footer/All Pages from the main menu. Type the text of the footer between the BEGIN and END FOOTER commands Sprint inserts. To create a footer for the first page, choose Layout/Footer/Title Page.
Force text to right margin	Choose Insert/Wide Space (Spring) before you enter the text.
Insert page number variable	Choose Insert/Variable, then pick page from the list of variables Sprint displays and choose a template for the way you want Sprint to insert the page number at print time.

This completes Lesson 9.

Lesson 10: Footnotes

The Footnote command (on the Style/References menu, Figure 1.7) allows you to insert footnotes in your printed document.



```
failure. Sprint, of course, doesn't eliminate these situation
but it does protect your open files when failures occur.<
<
Sprint automatically creates a swap (backup) file called SP.S
The file is automatically updated with your changes as you ed
In the event of a power failure, your text is safe and can be
retrieved as soon as power resumes. You simply
continue where you left off. The most you can l
much you typed since the last three-second paus
not much (unless your fingers never pause). Cha
"Working with Files," explains the backup file
<
SECTION Menus, Commands, and Predefined Keys<
There are a variety of ways to take advantage o
and formatting capabilities. You can<
<
@BEGIN{hyphens}<
choose commands from Sprint's pop-u
<
press special "shortcut" keys to mo                              t,
or use a menu command without displ
<
define your own keystrokes to perform any number of word
processing functions<
...\RASPBERR\USERGYD\CH5.UG        * Ins         6:18pm   Ln.140 of 593   Col14
```

Figure 1.7: The References Menu

When you choose Style/References/Footnote, Sprint inserts BEGIN and END FNOTE commands in your file and positions your cursor between them. Just type the text of the footnote between the BEGIN and END commands. When the Sprint formatter encounters a Footnote command in your file, it automatically assigns a number to the footnote and prints the number in small, raised type. At print time, the text of the footnote appears beneath a line drawn across the bottom of the page. If the footnote refers to something in a table, it will print at the end of the table instead of at the bottom of the page. If your printer doesn't support a small font, Sprint prints the number in plain text; if your printer can't perform vertical microspacing, Sprint places the number one-half line above the text to be referenced.

It seems a little strange to see footnote text in the middle of your other text onscreen, but you can get an idea of how a printed table footnote looks on page 11. A footnote in regular text looks the same, but occurs at the bottom of the page.

Note: If you want Sprint to print references at the *end* of the document instead of on the current page, choose Endnote or Note from the **R**eferences menu instead of Footnote. Like Footnote, the Endnote command prints a small, raised number in the text; instead of printing the reference text at the bottom of the current page, however, Endnote prints the text and number of the reference at the end of the document, on a *Notes* page. The Note command doesn't print a reference number in text or on the *Notes* page; it simply prints a note at the end of the document. This is useful when constructing bibliographies or other unnumbered types of references.

If you'd prefer your footnotes to be referenced by an asterisk (*) in the text and at the bottom of the page instead of by a number, you can use the Notes command to get "star" notes. This looks the same as a footnote created with the Footnotes command, except that asterisks are used instead of numbers. The first "star" note on a page will have one asterisk, the second will have two, and so on.

Note: You can place notes at the end of each chapter rather than at the end of the document—just choose Style/Other Format, type `Place Notes`, and press *C* for Command. This only works for notes.

Practice

In this lesson, you'll insert a footnote at the end of the *Drawer Slides* line of the *Required Materials for Smith Job* table, after the words *10 pair.*

Creating a Footnote

1. Move the cursor to the end of the *Drawer slides* line of the *Required Materials for Smith Job* table; your cursor should be at the end of the word *pair.*
2. Choose Style/References/Footnote:

   Press: *F10 SRF*

   Sprint inserts BEGIN and END FNOTE commands and places the cursor between them.
3. Enter the text of your footnote:

   Type: `Johnny's Hardware has the best price.`

   Because the item the footnote is referencing occurs in a table, the footnote will appear at the end of the table instead of at the bottom of the page.

The following table summarizes the task presented in Lesson 10:

Table 1.11: Tasks in Lesson 10

Task	Action
Insert a footnote	Choose Style/References/Footnote from the main menu and enter the text of your footnote between the BEGIN and END FNOTE commands.
Insert an endnote	Choose Style/References/Endnote and enter the text of your endnote between the BEGIN and END ENOTE commands.
Insert a "star" note	Choose Style/References/Note and enter the text of the note you want referenced with an asterisk (*) between the BEGIN and END SNOTE commands.

This completes Lesson 10.

Lesson 11: Cross-References

If you previewed your file on screen, you've seen that Sprint automatically replaces certain commands, such as tables, figures, headings, and footnotes, with numbers. This ability is extended to let you cross-reference any numbered element created with Sprint. For example, you can reference section, table, and figure numbers without knowing the number Sprint will assign when it prints your file. Suppose that, in the *SUPPLIERS* section of your file, you want to cross-reference the *Required Materials for Smith Job* table you created. The *SUPPLIERS* section might read:

```
Each company listed below can provide some or all of the materials
listed in Table
```

You could, of course, enter `1` after the word `Table`, since it's the only table in your file. If you happened to insert one or more tables above it, though, you'd have to go back and change your table reference. This "hard-coding" scheme leaves a lot of room for error in a large document, and makes maintaining a document a lot more work. That's why Sprint provides cross-reference (X-Reference) commands. These commands let you make "soft references;" that is, *tag* names (which you make up) are coded in the file near the information you want to reference, such as your table. When you want to reference an item, you refer to the tag, and let Sprint fill in the correct number. That way, if you add or delete a table or figure, your references will always be correct!

This is a simple exercise to help you understand cross-referencing. Don't worry if it's still a little confusing when you're through with this exercise; after you use these commands a bit, you'll see their effect.

Since you're going to reference the *Required Materials for Smith Job* table in the *SUPPLIERS* section, you need to define a tag for this table.

Defining a Tag

1. Search for the TCAPTION command:

   Press: *F7* for the Find command

   Sprint prompts `Forward search:`

   Type: `TCAPTION`
   Press: *Enter*

   Note: The tag for a table or a figure must come *after* the caption, or else the count that Sprint takes to cross-reference the table or figure number will be wrong.

2. At the end of this line, insert a new line and choose the **Style/X-Reference/Define** a Tag command:

   Press: *End* to reach the end of the line
   Press: *Enter* to insert one line
   Press: *F10 SXD*

   Sprint prompts `Name for new tag:`

   What you want to do is *tag* the table number (which you don't know until you print) with some unique word you'll remember, like *materials*. That way, when you want to cross-reference the table number, you can reference the tag you defined. When you print your file, Sprint will automatically replace the references with the actual numbers of the items you tagged.

3. Use the word *materials* as your tag and tell the formatter that what you're referencing is a *table*:

   Type: `materials=table`
   Press: *Enter*

   The `=table` part of the command is necessary for Sprint to realize you're tagging a table and number it accordingly. (If you were tagging a figure instead, you would have typed `materials=figure`.)

Referencing a Tag

1. Search the file for the the *SUPPLIERS* heading, cursor down one line, insert a blank line by pressing *Enter*, and enter the introductory sentence:

   Type: `Each company listed below can provide some or all of the materials listed in Table`

   Leave a blank space after the word `Table`. Don't press *Enter* yet. You're going to insert a cross-reference there.
2. You now want to tell Sprint which table to reference. To do this, choose **S**tyle/**X**-Reference/**R**eference a Tag and tell Sprint the name of the tag you want to reference (the tag you defined for the table):

   Press: *F10 SXR*

   Sprint prompts `Tag to reference:`

   Type: `materials`
   Press: *Enter*

   Sprint then displays the **R**eference By menu. If you choose **P**age Number, Sprint inserts the page number on which the tagged text appears when it prints your file; if you choose **A**ssigned Number, Sprint will insert the actual number assigned to the tag. In this example, choosing **A**ssigned Number will insert the number Sprint assigns to your table.
3. Choose **A**ssigned Number:

   Press: *A* for **A**ssigned Number

   Sprint inserts a `^V` character in front of the `materials` tag you specified, and a `^N` at the end of it. You can see this by pressing *Alt-Z* (show hidden control codes).
4. Type a period (.) to end the sentence, and then press *Enter.*

That's all there is to it. Use a tag to identify something you want to cross-reference. Reference the tag by name when you want to refer to a tagged item.

If this still doesn't seem crystal clear, don't worry. When you print or preview your formatted text, you'll see the effect of your **D**efine a Tag and **R**eference a Tag commands.

The following table summarizes the tasks presented in Lesson 11:

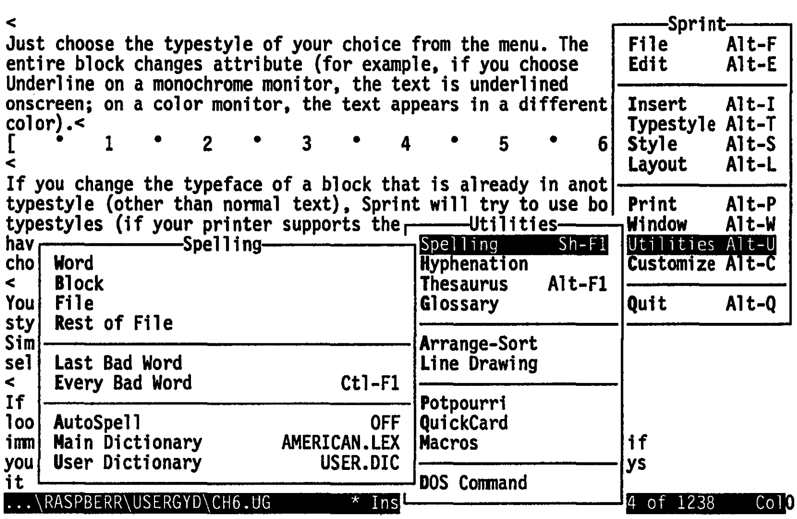
Table 1.12: Tasks in Lesson 11

Task	Action
Define a tag	Choose **S**tyle/**X**-Reference/**D**efine a Tag from the main menu. Enter the name of the tag (a made-up reference word), an equal sign (=), and the Sprint item you're tagging; the latter can be a table, a figure, a page, a chapter, an appendix, etc.
Reference a tag	Choose **S**tyle/**X**-Reference/**R**eference a Tag from the main menu. Enter the tag name of the thing you've defined, then press either *P* for **P**age Number or *A* for **A**ssigned Number (use **P**age Number only for page references, Assigned Number for anything else).

This completes Lesson 11.

Lesson 12: Correcting Spelling

Sprint's spelling utility compares the text in a file with the words in Sprint's built-in dictionary. Sprint can check the spelling of words as you type, or when you've completed a document. You can check the spelling of a single word, a marked block, from a specific point to the end of the file, or your entire file. You'll find the spelling commands on the **U**tilities/**S**pelling menu (Figure 1.8).



```
<                                                      ┌────Sprint────┐
Just choose the typestyle of your choice from the menu. The │File       Alt-F│
entire block changes attribute (for example, if you choose  │Edit       Alt-E│
Underline on a monochrome monitor, the text is underlined   ├──────────────┤
onscreen; on a color monitor, the text appears in a different│Insert     Alt-I│
color).<                                                    │Typestyle  Alt-T│
[   •   1   •   2   •   3   •   4   •   5   •   6│Style      Alt-S│
<                                                           │Layout     Alt-L│
If you change the typeface of a block that is already in anot├──────────────┤
typestyle (other than normal text), Sprint will try to use bo│Print      Alt-P│
typestyles (if your printer supports the┌───Utilities───┐Window     Alt-W│
hav┌──────────Spelling──────────┐│Spelling   Sh-F1│Utilities  Alt-U│
cho│Word                        ││Hyphenation     │Customize  Alt-C│
<  │Block                       ││Thesaurus  Alt-F1├──────────────┤
You│File                        ││Glossary        │Quit       Alt-Q│
sty│Rest of File                │├────────────────┤──────────────┘
Sim├────────────────────────────┤│Arrange-Sort    │
sel│Last Bad Word               ││Line Drawing    │
<  │Every Bad Word       Ctl-F1 │├────────────────┤
If ├────────────────────────────┤│Potpourri       │
loo│AutoSpell               OFF ││QuickCard       │
imm│Main Dictionary AMERICAN.LEX││Macros          │if
you│User Dictionary    USER.DIC │├────────────────┤ys
it └────────────────────────────┘│DOS Command     │
...\RASPBERR\USERGYD\CH6.UG   * Ins└────────────────┘4 of 1238    Col0
```

Figure 1.8: The Spelling Menu

If you want Sprint to check your spelling as you type, set **AutoSpell** to On. Whenever you type a word that's not in Sprint's dictionary, you'll hear a beep. If AutoSpell is On, you can use the Last Bad Word and **E**very Bad Word commands to search for spelling errors that were recorded as you typed.

Note to two-floppy system users: If the dictionary files you need in order to use the Sprint speller or thesaurus are not on your Program Disk, Sprint will prompt you to remove the Program Disk from Drive A and insert the disk that contains the files. When you've finished correcting spelling or investigating synonyms, replace the Program Disk. Unfortunately, you cannot use **A**utoSpell mode on a two-floppy system.

To check a word, a block, or your entire file, choose **U**tilities/**S**pelling, and then select the text you want to check (for example, the current word, block, file, and so on). Once you choose a command, Sprint displays the first unknown word it encounters, provides a list of similarly spelled words, and allows you to choose from five options:

Add to Dictionary — Adds the highlighted word to Sprint's dictionary, so it won't be considered a misspelled word.

Replace With — Lets you retype the word correctly, and then inserts the correctly spelled word in the file.

Lookup	Lets you choose the desired word from a list of alternate similarly spelled words.
Skip Once	Ignores the spelling this time only.
Ignore	Ignores this word throughout the file.

Note: If you wish to stop the spelling checker, press *Esc* at any time, rather than choosing one of the five options.

Practice

Before printing your final document, you will want to make sure that everything is spelled correctly.

Checking Your File's Spelling

1. If you have a two-floppy system, remove the Program Disk from Drive A and replace it with the Spell Disk.
2. Choose **Utilities/Spelling/File** to tell Sprint you want to check the entire file:

   Press: *F10 USF*

   Any words Sprint doesn't recognize (words not in the Sprint dictionary) will now be brought to your attention, one at a time, and you'll see the options listed above. When you are finished correcting your document, be sure to save your file.

The following table summarizes the task presented in Lesson 12:

Table 1.13: Task in Lesson 12

Task	Action
Spell-check a file	Choose Utilities/Spelling/File from the main menu

This completes Lesson 12.

Lesson 13: Logging Error Messages

As we mentioned in Lesson 8 (beginning on page 37), Sprint displays an error message on the screen whenever it detects an error during formatting, and will not print the document until you correct the error. For example, if you accidentally delete an END HEADER command and try to print the file, Sprint displays a message like this:

```
annual.rpt line 11 Error: Begin Header on line 8 missing End
```

Before Sprint will print your file, you'll need to edit your file and add the missing command.

Sprint may also display *warning* messages during formatting. These are different from error messages because Sprint will continue formatting and print your file if it doesn't find an error message along the way. Warning messages occur if Sprint can't do what you wanted, but can "work around" the problem to let you print. For example, many commands in the STANDARD.FMT file are set up so that when you print on a fancy printer (like an Apple or HP laser printer), Sprint will use different fonts and/or typestyles. If you're printing a draft on a printer that can't support this type of formatting, Sprint will display a warning message, like this:

```
\SPRINT\standard.fmt line 8 Warning: Printer does not have 'Times' font.
```

This means that a command you entered on or near line 8 calls for Sprint to print the text in the *Times* font, but your currently selected printer can't print that font. Sprint ignores the font change part of the command and prints the text in a font supported by your printer. When you print your document on a different, more capable printer, these warnings will no longer appear.

When you choose **Log** Errors to File from the **Print/Advanced** Options menu and set it to Yes, the formatter saves all error and warning messages to a file with the same name as the file you're formatting; it adds a .LOG extension to the error-log file name to distinguish it from your text file. When you choose this option, you don't have to check the onscreen display of error messages and manually note the location and nature of each error; you can display the log file in one window and the file you're correcting in another window, and switch between the two (using *Shift-F6*), to correct all the errors listed in the .LOG file.

If you're formatting a large document, or a heavily formatted file, you may end up with more formatting errors than you expect. Don't worry; it's quite common.

Practice

In this lesson, you'll ask the formatter to log errors to a file.

Logging Errors to a File

1. Make sure to set **Print/A**dvanced Options/**L**og Errors to File to Yes:

   Press: *F10 PAL*

When toggled to Yes, this print option logs all formatter messages to a file on disk.

2. Print the file:

   Press: *Esc* to remove the Advanced Options menu
   Press: *G* to choose Go

   Sprint writes the file to disk, and then begins formatting. If it finds any error messages, it will display them on the screen and continue formatting. When it completes formatting, you'll see a message saying to press any key to continue.

3. Press any key and then open a window:

   Press: *Shift-F3* to open a window

4. Now open the log file:

   Press: *Ctrl-F3*

   Sprint prompts `File to open:`

   Type: `Smith.log`

   Both the SMITH.SPR and SMITH.LOG files should be displayed on the screen.

5. With the cursor in the SMITH.LOG window, search for the word *Error*. Note the line number listed in the error message.

6. Move to the other window and search for the line number listed in the error message:

   Press: *Shift-F6* to move from one window to the next
   Press: *F9* to choose Jump to Line
   Type: the line number of the error message at the prompt

   In most cases, the error appears on the line number displayed in the error message. There are times, though, when Sprint approximates the line on which the error appears. If you can't find an error on this line, begin looking backward for the missing or offending command.

7. Correct the error. If you have other errors in your file, repeat the steps above for each error listed in your error log file.

8. If no errors were found, try making a couple deliberately. Delete the command `END HYPHENS` from line 28, and deliberately misspell "inches" in the `RESERVE 2.5 inches` command on line 50 by deleting a "c" so it reads "2.5 inces." Go ahead and print your file with the errors logged to a file called SMITH.LOG (press *F10 PAL*, then *PG*). When you open the SMITH.LOG file, you'll see these error messages:

```
SMITH.SPR line 50 Error: Unknown unit of measure 'inces'.
SMITH.SPR line 114 Error: Begin Hyphens on line 17 missing End.
```

Note: An error message saying that a format doesn't have an end or a beginning is commonly caused by accidental deletion of a command or by formats nested incorrectly (usually, this means the end format commands are not listed in reverse order of the begin format commands, as they should be).

9. Undo the experimental errors you created in the last step.
10. Now that you've checked for formatting errors and corrected them, you're ready to print your file. You don't have to log error messages to a file each time you want to print, but it's a good habit to get into. If Sprint finds any errors, you'll have an accurate list to work from; if your file doesn't contain any errors, Sprint will begin printing it automatically.

The following table summarizes the tasks presented in Lesson 13:

Table 1.14: Tasks in Lesson 13

Task	Action
Log errors to a file	Choose **Print/Advanced Options/Log Errors to File** and toggle to Yes. Send the file to the printer or to a file as you normally do.
Strip errors from a file	Open the .LOG file Sprint creates. Read the error message. Switching back to the file you were trying to print (preferably in an open window), jump to the line number referenced in the error message and correct it. Repeat the correction process for each error message in the .LOG file.

This completes Lesson 13.

Lesson 14: Paginating and Adjusting Page Layout

Sometimes a page of text doesn't end the way you'd like it to. For example, you might end up with four lines of a paragraph at the bottom of one page, and the last line of that paragraph at the top of the next page. Or, let's say the formatter can only print step 1 of a procedure before a page becomes full and has to place the remainder of your list on the next page. You can always correct these problems after you print your file, but there's an easier and faster way to determine where Sprint is going to break your pages—the **Print/Paginate** command, which you used in the Quick Start Tutorial.

Paginate saves and formats your file, checks for all page breaks, and then displays a solid bold line in your file to indicate each automatic page break. If these page breaks are unacceptable, use the commands on the Layout/ Page Breaks menu to override the formatter page breaks and use Paginate again until you're satisfied with the results.

If you aren't satisfied with a formatter page break, you can edit your file and override the automatic page break. The commands on the Layout/ Page Breaks menu let you group text on the page and specify where the formatter should break a page.

Insert (unconditional) — Inserts a hard (unconditional) page break in the file, and also displays a solid bold underline to indicate a page break. Text following this command will appear on the next page.

Conditional Page Break — Specifies where Sprint can break the page if it has to.

Reserve Space — Inserts a specified amount of blank space.

Blank Page(s) — Inserts the specified number of blank pages.

Group Together on Page — Keeps selected text together on a page.

Keep with Following Text — Prohibits Sprint from breaking the page at the location of this command.

Widow-Orphan Control — Specifies the minimum number of lines that may appear at the bottom or top of the page.

There is a disadvantage to the Insert (unconditional) command. If you later add or delete text in your file, the new page you inserted may no longer be appropriate. For example, you may have inserted an Insert (unconditional) command so that a numbered list begins at the top of a new page. If you later add a few lines of text before the text of this list, and it fills the page, Sprint would begin a new page automatically, see the Insert (unconditional) command, and then insert a new page before printing your list. You'd end up with a blank page between your added text and the text of your list. Then you'd have to go back to your file, remove the Insert (unconditional) command, Paginate the file again to see if any of your other changes, additions, or deletions affected the page breaks, and then print your file.

If you are producing a lengthy document and/or periodic drafts of a "growing" document (like a specification or a manual whose content is continually changing), you should stay away from the Insert (unconditional) command and use either Group Together on Page, Conditional Page Break, or **Keep with Following Text**. Group Together on Page keeps text together, no matter where it appears during formatting; it tells Sprint that the text within this format must appear together, on the same page, regardless of any page break or formatting commands.

For example, if you select the text of a numbered list and choose **Layout/ Page Breaks/Group Together on Page**, and the entire list can't fit on the current page, Sprint automatically begins the list on the following page. If you later add or delete text, and your page breaks are affected, you don't have to worry about unnecessary **Insert** (unconditional) commands; Sprint will always group that text on one page (unless the group's too large for the page, in which case you'll get a warning message when you format)

You can also use the **Conditional Page Break** command. This allows you to specify where Sprint can break a page if necessary. To enter a conditional page break, choose **Layout/Page Breaks/Conditional Page Break** from the main menu and press *Enter*.

Practice

Since we don't know what type of printer you're using, and where your page breaks occur, we can't easily advise you on where to insert any of these commands. Try changing the default page breaks in your file, using at least one of the commands listed above, before going on to the next lesson.

The following table summarizes the tasks presented in Lesson 14:

Table 1.15: Tasks in Lesson 14

Task	Action
Override automatic page breaks	Choose a command from the Layout/ Page Breaks menu.

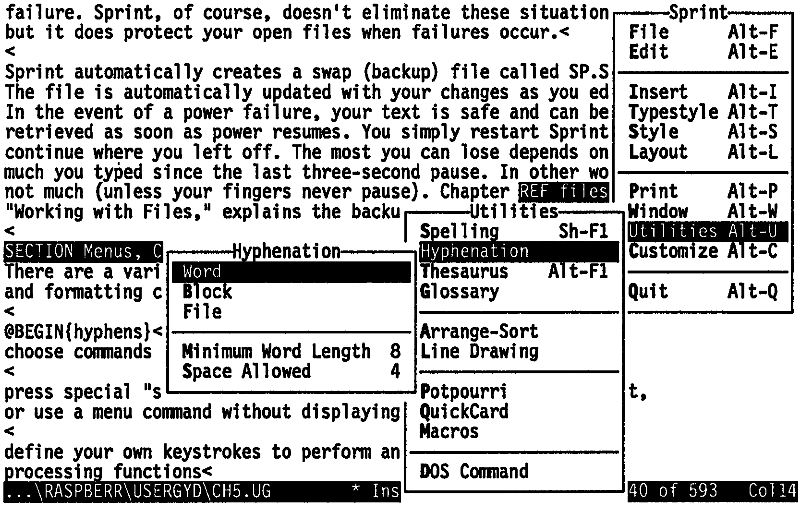
This completes Lesson 14.

Lesson 15: Conditionally Hyphenating Text

Sprint's **Hyphenation** menu allows you to specify discretionary hyphenation within your file. When you insert a discretionary (soft) hyphen, Sprint will only break a word into two hyphenated parts where necessary to justify the line. If the line justifies well without breaking any words, Sprint ignores the discretionary hyphen.

To use a hyphenation command choose **Utilities/Hyphenation** from the main menu. Sprint displays the **Hyphenation** menu (Figure 1.9).

Note to two-floppy system users: If the dictionary files you need to use commands on the **Hyphenation** menu are not on your Program Disk, Sprint will prompt you to remove your Program Disk from Drive A and replace it with the disk that contains the files. When you've finished using commands from the **Hyphenation** menu, replace the Program Disk.



```
failure. Sprint, of course, doesn't eliminate these situation ───Sprint───
but it does protect your open files when failures occur.<     File      Alt-F
<                                                             Edit      Alt-E
Sprint automatically creates a swap (backup) file called SP.S ────────────────
The file is automatically updated with your changes as you ed Insert    Alt-I
In the event of a power failure, your text is safe and can be Typestyle Alt-T
retrieved as soon as power resumes. You simply restart Sprint Style     Alt-S
continue where you left off. The most you can lose depends on Layout    Alt-L
much you typed since the last three-second pause. In other wo ────────────────
not much (unless your fingers never pause). Chapter REF files Print     Alt-P
"Working with Files," explains the backu ───Utilities─────── Window    Alt-W
<                                        Spelling     Sh-F1   Utilities Alt-U
SECTION Menus, C ───Hyphenation──────── Hyphenation           Customize Alt-C
There are a vari  Word                   Thesaurus    Alt-F1  ────────────────
and formatting c  Block                  Glossary             Quit      Alt-Q
<                 File                   ──────────────────
@BEGIN{hyphens}< ─────────────────────── Arrange-Sort
choose commands   Minimum Word Length  8 Line Drawing
<                 Space Allowed        4 ──────────────────
press special "s ─────────────────────── Potpourri            t,
or use a menu command without displaying QuickCard
<                                        Macros
define your own keystrokes to perform an ──────────────────
processing functions<                    DOS Command
...\RASPBERR\USERGYD\CH5.UG        * Ins                      40 of 593   Col14
```

Figure 1.9: The Hyphenation Menu

Word	Lets you conditionally hyphenate a word with the cursor placed anywhere on it.
Block	Lets you conditionally hyphenate a block (select it first).
File	Lets you conditionally hyphenate a file, starting at the top.
Minimum Word Length	Lets you change the minimum number of characters required before a word may be hyphenated. The default is 8. Sprint will prompt you for the new minimum word length.
Space Allowed	Lets you specify the widest space allowed between characters on a line to justify that line. The default is 4.

When you choose **Hyphenate/File**, Sprint checks the file, beginning at the top, for words that might need hyphenation. Each time it finds a word to hyphenate, it displays a menu of hyphenation choices—to choose one, just highlight it and press *Enter.*

Practice

For this lesson, you'll specify a minimum word length of six characters, set the space allowable between words to two, and conditionally hyphenate the entire file.

Hyphenating Your File

1. Change the minimum word length by choosing **Utilitites/Hyphenation/Minimum** Word Length:

   Press: *F10 UHM*

   Sprint prompts `Shortest word length to be hyphenated:`

   (If you have a two-floppy system, and the dictionary files you need are not on your Program Disk, Sprint will prompt you to insert the disk that contains the files).

2. Specify six characters as the shortest word to be hyphenated:

   Type: `6`
   Press: *Enter*

   Now Sprint will look at all words that contain at least six characters and decide whether it should insert a conditional hyphen.

3. Change the amount of space that Sprint can add to a line when justifying it. Choose Space Allowable from the **Utilities/Hyphenation** menu:

   Press: *S* for Space Allowable

   Sprint prompts `Widest allowable justification space:`

4. Tell Sprint that, when it justifies a line, it can only add two extra space characters between words. If Sprint needs to insert more than two space characters between words to justify the line, it will "stretch" the characters within one or more words on the line:

   Type: `2`
   Press: *Enter*

5. Have Sprint hyphenate the entire file. Choose the **File** command from the **Hyphenate** menu:

   Press: *F*
   Press: *Enter*

Sprint will read the entire file, conditionally hyphenate the text (all words with at least six characters that also appear near a line break), and insert a maximum of two extra spaces between words (where necessary) to justify the right margin.

The following table summarizes the tasks presented in Lesson 15:

Table 1.16: Tasks in Lesson 15

Task	Action
Insert a discretionary hyphen	Choose **Utilities/Hyphenate/Word** and pick a hyphenation option from the list Sprint displays.
Hyphenate a block	Select a block, then choose **Utilities/Hyphenate/Block.**
Hyphenate a file	Choose **Utilities/Hyphenate/File.**
Change the number of spaces Sprint can insert into a line to justify it	Choose **Utilities/Hyphenate/Space Allowable** and enter the desired number at the prompt (default is 4).
Change the length of the shortest word Sprint can hyphenate	Choose **Utilities/Hyphenate/Minimum Word Length** and enter the desired word length at the prompt (default is 8 characters).

This completes Lesson 15.

Lesson 16: Modifying Formats

Now that you've become familiar with some of Sprint's advanced functions, you're ready to build on format skills.

Typically after printing you will want to change some of your text within formats. You may wish to change the spacing of text within a format, force Sprint to print a table at the top of a page, or change the justification of text in a list. Appendix D in this book lists all the *parameters* you can add or change in a format command. These parameters affect a chosen format only; that is, if you modify a particular **Numbered** format, only the text in that format will be affected. Text within other **Numbered** formats will not be affected.

When you choose the **Style/Modify** command, Sprint begins searching backward (toward the top of the file) for a BEGIN command (for example, BEGIN TABLE, BEGIN FIGURE, and so on). As soon as it locates a BEGIN command, it displays the Modify menu. This menu asks if you want to modify This Format or the Previous Format. If you choose This Format, Sprint prompts for the parameters you want to add; if you choose Previous Format, it continues the search for a BEGIN command.

Practice

In this lesson, you're going to modify two formats. First, you're going to modify the Numbered format in the *FINISHING INSTRUCTIONS* section. By default, Numbered inserts a blank line between each paragraph within the format. Let's modify the *spread* (distance) between paragraphs so there are no blank lines; that is, you want the list to print single-spaced.

The second format you're going to modify is the Table format in the *MATERIALS* section. By adding the *Above* parameter, you can force Sprint to print the table at the top of the page.

Modifying Formats in Your File

1. Go to the end of the file and choose Style/Modify:

   Press: *Ctrl-PgDn* to reach the end of the file
   Press: *F10 SM*

   Sprint searches backwards in the file and stops at the first format command it encounters, BEGIN NUMBERED. displaying the This Format or Previous Format choices.

2. Choose This Format:

   Press: *T* for This Format

   Note: If Sprint stops at any other BEGIN command, choose Previous Format until Sprint finds BEGIN NUMBERED. Then choose This Format.

3. Specify single-spacing when Sprint prompts `Modify by adding:`

   Type: `spread 0`
   Press: *Enter*

   Sprint automatically adds this parameter to the BEGIN NUMBERED command line.

4. Choose Style/Modify again. After BEGIN NUMBERED, Sprint will find the BEGIN FNOTE command. You don't want to modify this format, so choose Previous Format.

5. When Sprint finds BEGIN TABLE, choose This Format and add the following parameters:

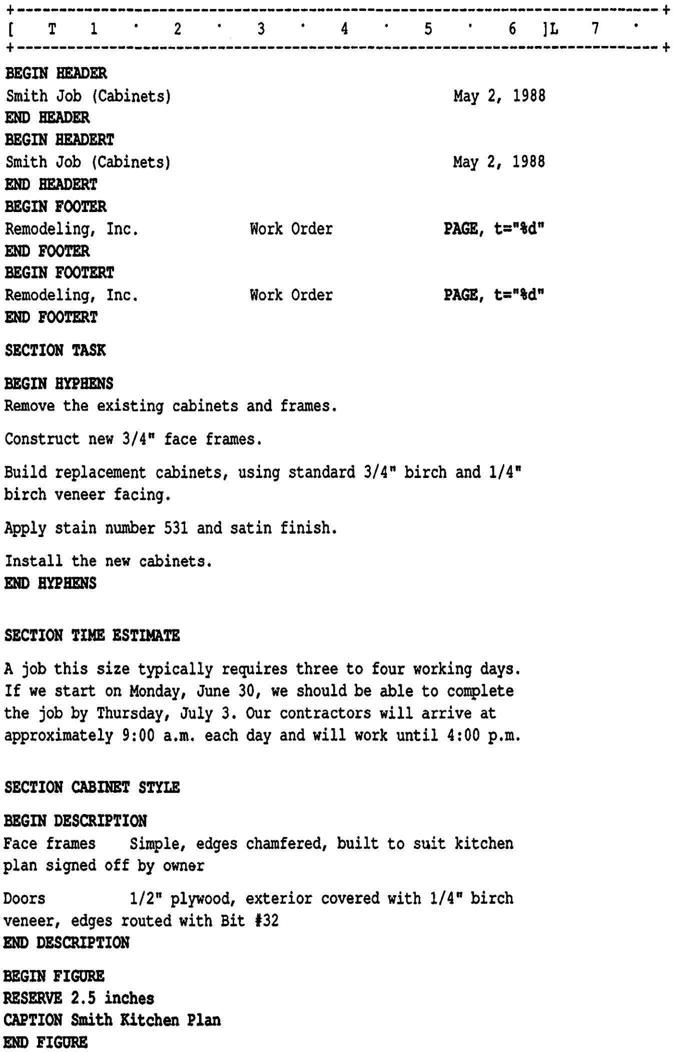
   Type: `above, spread 0`
   Press: *Enter*

The *above* parameter tells Sprint to print the table text at the top of the page. The *spread 0* parameter specifies that the table should print single-spaced.

You can add any number of parameters, as long as you separate each parameter with a comma.

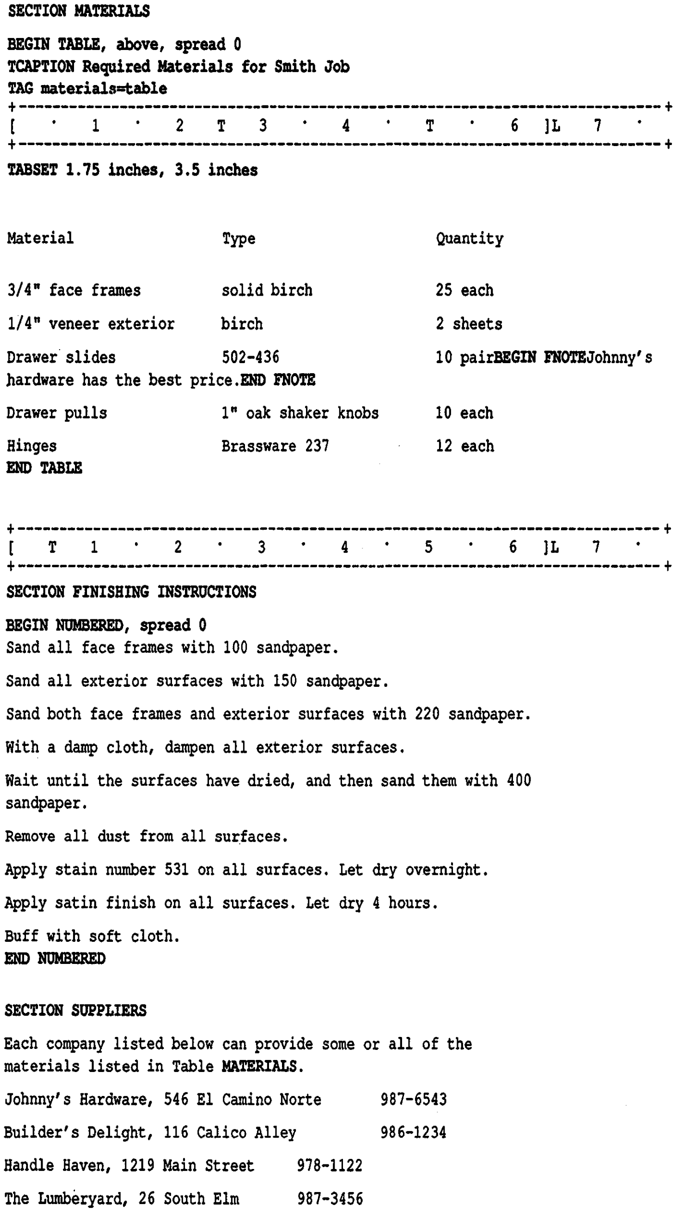
6. To see the effect of your format changes, preview the file on your screen or go on to the next lesson and print the final document.

Your file should look like Figures 1.10 and 1.11 at this point:



```
+-----------------------------------------------------------------------------+
[    T    1    ·    2    ·    3    ·    4    ·    5    ·    6   ]L    7    ·
+-----------------------------------------------------------------------------+
BEGIN HEADER
Smith Job (Cabinets)                                May 2, 1988
END HEADER
BEGIN HEADERT
Smith Job (Cabinets)                                May 2, 1988
END HEADERT
BEGIN FOOTER
Remodeling, Inc.             Work Order           PAGE, t="%d"
END FOOTER
BEGIN FOOTERT
Remodeling, Inc.             Work Order           PAGE, t="%d"
END FOOTERT

SECTION TASK

BEGIN HYPHENS
Remove the existing cabinets and frames.

Construct new 3/4" face frames.

Build replacement cabinets, using standard 3/4" birch and 1/4"
birch veneer facing.

Apply stain number 531 and satin finish.

Install the new cabinets.
END HYPHENS


SECTION TIME ESTIMATE

A job this size typically requires three to four working days.
If we start on Monday, June 30, we should be able to complete
the job by Thursday, July 3. Our contractors will arrive at
approximately 9:00 a.m. each day and will work until 4:00 p.m.


SECTION CABINET STYLE

BEGIN DESCRIPTION
Face frames    Simple, edges chamfered, built to suit kitchen
plan signed off by owner

Doors          1/2" plywood, exterior covered with 1/4" birch
veneer, edges routed with Bit #32
END DESCRIPTION

BEGIN FIGURE
RESERVE 2.5 inches
CAPTION Smith Kitchen Plan
END FIGURE
```

Figure 1.10: Your New Work Order File, Page 1



```
SECTION MATERIALS

BEGIN TABLE, above, spread 0
TCAPTION Required Materials for Smith Job
TAG materials=table
+----------------------------------------------------------------------------+
[    ·    1    ·    2    T    3    ·    4    ·    T    ·    6   ]L    7    ·
+----------------------------------------------------------------------------+
TABSET 1.75 inches, 3.5 inches


Material                 Type                     Quantity


3/4" face frames         solid birch              25 each

1/4" veneer exterior     birch                    2 sheets

Drawer slides            502-436                  10 pairBEGIN FNOTEJohnny's
hardware has the best price.END FNOTE

Drawer pulls             1" oak shaker knobs      10 each

Hinges                   Brassware 237            12 each
END TABLE


+----------------------------------------------------------------------------+
[    T    1    ·    2    ·    3    ·    4    ·    5    ·    6   ]L    7    ·
+----------------------------------------------------------------------------+
SECTION FINISHING INSTRUCTIONS

BEGIN NUMBERED, spread 0
Sand all face frames with 100 sandpaper.

Sand all exterior surfaces with 150 sandpaper.

Sand both face frames and exterior surfaces with 220 sandpaper.

With a damp cloth, dampen all exterior surfaces.

Wait until the surfaces have dried, and then sand them with 400
sandpaper.

Remove all dust from all surfaces.

Apply stain number 531 on all surfaces. Let dry overnight.

Apply satin finish on all surfaces. Let dry 4 hours.

Buff with soft cloth.
END NUMBERED

SECTION SUPPLIERS

Each company listed below can provide some or all of the
materials listed in Table MATERIALS.

Johnny's Hardware, 546 El Camino Norte      987-6543

Builder's Delight, 116 Calico Alley         986-1234

Handle Haven, 1219 Main Street     978-1122

The Lumberyard, 26 South Elm       987-3456
```

Figure 1.11: Your New Work Order File, Page 2

The following table summarizes the tasks presented in Lesson 16:

Table 1.17: Tasks in Lesson 16

Task	Action
Modify a format	Choose Style/Modify, press *P* for Previous format or *T* for This format, and enter the parameter you want at the prompt.
Single-space text in a format	Choose Style/Modify and enter the parameter `spread 0`.
Print a format at the top of the page	Choose Style/Modify and enter the parameter above.

This completes Lesson 16.

Lesson 17: Printing a Final Document

Now your file is formatted exactly the way you want it, so you can print your final document. If you've been previewing and printing your text with an alternate printer, now is the time to choose your most capable printer. Once you've done this, follow the steps below.

1. Assuming you've chosen your best printer for this job, display the Print menu.
2. Check the Destination command:

Printer	Means that Sprint will format the file for output to your currently selected printer. If you want to change to a different printer, choose Current Printer from this menu, and then choose the printer you want to use.
File	Means that Sprint will format the file as if it were going to print it on your currently selected printer, but will write the formatted text to a file on disk instead of to the printer. When you choose Destination FILE, Sprint prompts you for a file name. If you don't enter a file name, Sprint will automatically write the formatted text to a file with the same name as your text file, but will append the .PRN extension. For example, if you format the file MYMEMO, Sprint will write the formatted text to a file called MYMEMO.PRN.

The benefit to Destination FILE is that you can print the formatted file with the DOS PRINT command. Why do this? Because a DOS PRINT command doesn't "tie-up" your machine while it's printing. Once you enter a DOS PRINT command, you return to the DOS prompt, so you can enter another command.

For now, if **Destination** doesn't specify PRINTER (the default), toggle the command from FILE to PRINTER.

3. Choose any options you'd like to include (for example, **Number of Copies**, **Log Errors to File**, and so on).
4. Choose **Go**. Sprint begins formatting your file, and then outputs the formatted version to the printer. Since you chose numbered headings, Sprint will automatically print a table of contents at the end of the document. You'll also get a list of figures and a list of tables; the **Figure** and **Table** captions cause Sprint to create and print these lists.
5. If you see anything you'd like to change on the finished work order (like a page break, spacing, tab stops, and so on), go ahead and edit the file and then reprint it.
6. When the file completes printing, give yourself a well-deserved pat on the back. You've just mastered the most comprehensive and flexible word-processing system around!

CHAPTER

2

Advanced Formatting: Tips, Tricks, and Techniques

This chapter expands on the basic formatting commands explained in the "Basic Formatting" chapter of the *User's Guide* and explains Sprint's more advanced features. Please note that you must be an *advanced user* (working with the advanced user interface) to display and select some of the commands explained in this chapter. To load the Advanced user interface, choose Customize/User Interface/Load and then choose SPADV from the list of user interfaces.

Note: The Advanced Tutorial in this manual (Chapter 1) provides hands-on practice using several commands in this chapter.

As an advanced user, you'll learn about the following formatting features:

Precise Ruler-Line Settings
You can use a variety of *dimensions* to specify indents from the left and right margins, the initial (first line) indent of paragraphs following the ruler, and tabs stops set on the ruler. For example, you can use inches, picas, centimeters, or character column to set indent values and tab stops. In addition, you can tell Sprint to print text in a particular font and type size.

On page 71, we explain how to specify precise settings on the ruler and list the various dimensions you can use.

Document-Wide Layout
You'll learn how to specify page size and set up *global* left and right margins (ruler lines are *relative* to these margin settings), top and bottom margins, and margins for page headers and footers. This discussion begins on page 74.

Headings

Sprint's Headings menu lets you select from a variety of heading formats. There are numbered headings, which automatically cause Sprint to produce a table of contents, and unnumbered headings, which visually separate headings from text but don't generate a table of contents. Heading formats also vary in type size and type style. For more information about heading formats, please see page 78.

Figures and Tables

Sprint's Figure and Table commands prompt for an optional caption, and automatically keep the text within the format together on a page. Sprint automatically numbers figures and tables and produces a list of figures and a list of tables at the end of the printed document. The "Figures and Tables" section (beginning on page 85) explains how to create and format figures and tables in Sprint files.

Multiple Columns

You can specify the number of columns you want to print and the spacing between the columns. Sprint supports *snaking columns,* which means that the formatter prints as much text as it can in one column and then begins printing the next column. You type the text in a single column (between normal ruler-line margins), but when you print, Sprint will automatically format the text in the specified number of columns. See page 86 for information about multiple columns.

Footnotes, Endnotes, and Notes

Footnotes are automatically numbered and printed at the bottom of the current page. Endnotes are formatted in the same way as Footnotes, but instead of printing the text of the note on the current page, Sprint prints the endnotes together at the end of the document. Slighty different from Footnote, the Notes command prints asterisks instead of reference numbers. We begin the discussion of these references on page 87.

Indexing

Sprint's Index commands let you quickly select words to be included in the index. You can mark text to be indexed, add text to be indexed but not printed as part of the text, include *see also* references, and specify a range of pages for an indexed entry. Page 88 begins the description of Sprint's Index commands.

File Linking

When creating large documents, you may find it more convenient to create several smaller files and then merge them when you format and print the document. This allows more than one person to work simultaneously on a document. For information on how to link Sprint files, see page 90.

Centered Text
There are several ways to center text on the page: vertically (between top and bottom margins) and horizontally (you can center a line, a region, or the entire file). The centered text discussion begins on page 94.

Page Breaks
Sprint performs automatic page breaks but provides commands that let you override the default page breaks, keep text together on a page, specify an acceptable point at which Sprint can break a page, and control orphan and widow lines at the top and bottom of a page. On page 95, we begin the discussion of Page Break commands.

Spacing
You can vary the spacing between lines, paragraphs, words, and even characters. Other spacing commands let you insert a fixed amount of blank space. Information about spacing commands begins on page 98.

Special Characters
Even though your screen may not be able to display special characters (such as the small box that a LaserWriter Plus prints, which we use as a bullet), you can tell the formatter which character you want printed. You can also specify a character to repeat across the line (for example, specify a period as the character to create leaders in a table). See page 103 for details.

Nonprinting Comments
You can annotate a file with comments and decide whether you want the formatter to print these comments. Page 105 explains how to do this.

Cross-References
You can avoid "hard" references (like typing in `See Chapter 2` or `to page 45`) by using Sprint's **X**-Reference (that is, *cross-reference*) commands. If you reorganize a document that contains these commands, your references will automatically be corrected the next time you print. For cross-reference information and examples, see page 106.

In discussing many of these advanced formatting commands, we make reference to STANDARD.FMT, a file that comes with your Sprint distribution disks. This file defines many of the formatting commands you can use. Chapter 3 explains this file in more detail.

Other Formats
Many Sprint formats are listed on the **S**tyle and **L**ayout menus (for example, **L**ists, **H**eadings, **F**igures, and **T**ables). There are several other formats, though, that aren't listed. For instance, the Example format automatically indents text one-half inch from the left margin and prints the text in a typewriter-style font. The Quotation command automatically indents text from both margins. These formats affect a marked block of text.

There are other formats that affect text at the current cursor position. For example, the MakeTOC command creates a table of contents when you've selected only unnumbered headings in a file. The "Other Formats" section begins on page 125. Within this section, Table 2.7 lists and explains the formats that affect a region of text, and Table 2.9 lists the current-position format commands.

Format Changes
Many formats can be modified to produce a different effect. For example, the Lists/Numbered command automatically inserts a blank line between each paragraph of a numbered list. You can modify this format, though, to remove the blank lines or insert more than one line between paragraphs. You can modify any format that inserts a BEGIN command in your file. The lengthy discussion on modifying formats begins on page 135.

Brand-New Formats
For situations where a Sprint format doesn't quite create the look you want, you can define your own format. See page 147 for more information.

Ruler Lines, Precise Settings, and Document-Wide Margins

Probably the most common need of an advanced user is to adjust margins and indents. Sprint is rich in ways to do this. Before we embark on explaining the numerous commands, here are some basic guidelines for using margin/indent commands:

- Use the Layout/Document-Wide margin commands when you want to change margins for the entire document. Although the ruler line won't reflect the margin command you typed, each page will print within the specified margins. For example, if you choose Document-Wide/Left Margin and type `8 picas`, text will begin printing 8 picas from the left edge of the page.
- *Don't use the Document-Wide margin commands to change the margins for a region of text!* Insert additional rulers and either change the margins on the new ruler(s), or use the Layout/Ruler/Precise Settings commands to change the indent from the document-wide margin(s).
  - If you change the margins on a new ruler line, the margins will be *relative* to the document-wide margins you set. That is, if you set a document-wide left margin at 1 inch, insert a second ruler, and set the left margin at column 10, text below the second ruler will begin printing 1 inch *plus* 10 columns from the left edge of the paper.

- If you choose **Layout/Ruler/Precise Settings**, the indent value you specify will be *added* to (or subtracted from, if you type negative indent values) the document-wide margin. For example, if you choose **Layout/Document-Wide/Left Margin** and type `.75 inches`, insert another ruler, select **Layout/Ruler/Precise Settings**, and set the **Left Indent** to `.5 inch`, the text below the second ruler will begin printing 1.25 inches from the left edge of the page (.75-inch left margin plus a .5-inch left indent).

■ Don't change the default margins on the *first* ruler in a file. Instead, use the **Document-Wide** margin commands.

The Ruler and Precise Settings

In Chapter 8 of the *User's Guide*, you learned how to set and change left and right margins, paragraph indents, justification, and tab stops. The information in that chapter pertained to *columnar* settings; that is, the instructions were for settings at a particular *column* on the ruler.

If your printer supports proportional-width fonts or can vary the size of printed characters, you may want more precise settings for a document's margins and tabs. For example, you may want to set tabs in picas or inches rather than at a particular column number. Sprint's *precise settings* give you this ability. You can use any of the dimensions listed in Table 2.1 to specify a precise setting.

Note: For ease of use and to avoid misspelling, many dimensions have multiple names.

Table 2.1: Formatting Dimensions

char, chars, character, characters	The width of a typical character (sometimes called an *en* space). Since fonts can be different sizes, this measurement varies from font to font. This measurement can only be used to indicate horizontal distances.
cm	Centimeters.
em, ems	Horizontally, the printer unit that is equal to the width of a lowercase *m* (the widest character in a proportional-space font). The *em* space varies from font to font. Vertically, an em is the same distance as a line.
en, ens	The width of a typical character. See the definition of *character* above.
in, inch, inches	Inches.
line, lines	Lines. Vertically, this is the height of a single-spaced line (usually equal to the point size of the current font). Horizontally, this is the distance between the left and right margins.
mm	Millimeters.
page	The height of the paper, which is usually 11 inches. This dimension specifies vertical distance only.
pica, picas	The printer unit that is equal to 12 points, or 1/6 of one inch (there are 6 picas per horizontal inch).
pt, pts, p, point, points	The printer unit that is equivalent to 1/72 of an inch (that is, there are 72 points per vertical inch).
u, unit, units	Derived from the printer definition, *units* represent the minimal horizontal and vertical movement of the print head on the printer. This is useful for special effects, but is a printer-dependent dimension. For example, on a LaserWriter, there are 300 units to an inch. Horizontal and vertical *units* may be different sizes.

Note: If you don't specify a dimension as part of a parameter, Sprint will automatically use *characters* for horizontal measures and *lines* for vertical measures.

Precise Settings also let you

- specify the font and type size you want for text following the current ruler
- vary the line spacing
- set a paragraph indent (first line of each paragraph is indented)
- set up a region of text so that it's all indented from the current left or right margin

When you choose Layout/Ruler/Precise Settings, Sprint searches backward for the first ruler it finds and then displays the **Precise Settings** menu.

To Print Text in a Different Font

Choose Font. Sprint displays a list of fonts for your default printer. Pick the font you want to use, and the formatter prints all text following the ruler in the selected font.

Note: If you want to change the font of a word or a selected area of text, use the Typestyle/Font command.

To Change the Size of Printed Text

Choose Size and specify the size you want for characters following the ruler line (for example, 8 points, 2 lines, or .5 lines). Remember: 1 inch equals 72 points in Sprint.

Note: If you want to change the size of a word or a selected area of text, use the Typestyle/Character Size command.

To Set Precise Indents

Choose Initial (First Line) Indent when you want to specify where the first line of each paragraph will begin printing. For example, if you type `1 inch`, the formatter will indent the top line of each paragraph 1 inch from the left margin. If you want all text indented from the left or right margin, choose the appropriate command (Left Indent or Right Indent) and specify a distance (for example, 3 picas, .75 inches, 10 cm). The changes in indents set this way are not seen until you print your document.

To Set Precise Tab Stops

Choose Tab Stops and type the desired location for your tab(s). If you're setting more than one tab stop, follow each setting with a comma. For example,

```
Place tabs at: 2 points, 6 picas, 12 picas
```

Although the screen doesn't reflect the precise settings, Sprint inserts a special command after the current ruler. (The special command word that appears is TABSET.) When you print the document, the formatter will interpret the precise settings on the ruler and send the desired output to your printer.

Note: The Layout/Ruler/Precise Settings commands affect the current ruler only! If you want to set up document-wide margins and indents, you must have only one ruler in your document (at the top) or use the commands on the Layout/Document-Wide menu. See the following section, "Document-Wide Layout," for details.

Document-Wide Layout

Document-wide layout refers to how Sprint will format the entire document. The default layout is dependent on page size—the dimensions of the paper on which you're going to print. Given a particular page size, Sprint automatically sets up top, bottom, left, and right margins. It also presumes that you want text to print single-spaced in a single column, with a single blank line between paragraphs. Figure 2.1 on page 75 illustrates the default layout for an 8.5 x 11-inch page. (The figure was reduced by 60 percent to fit into the page size of this manual.) Table 2.2 on page 76 explains each aspect of the default page layout. This section explains how to use the Layout/Document-Wide menu to change the default layout.

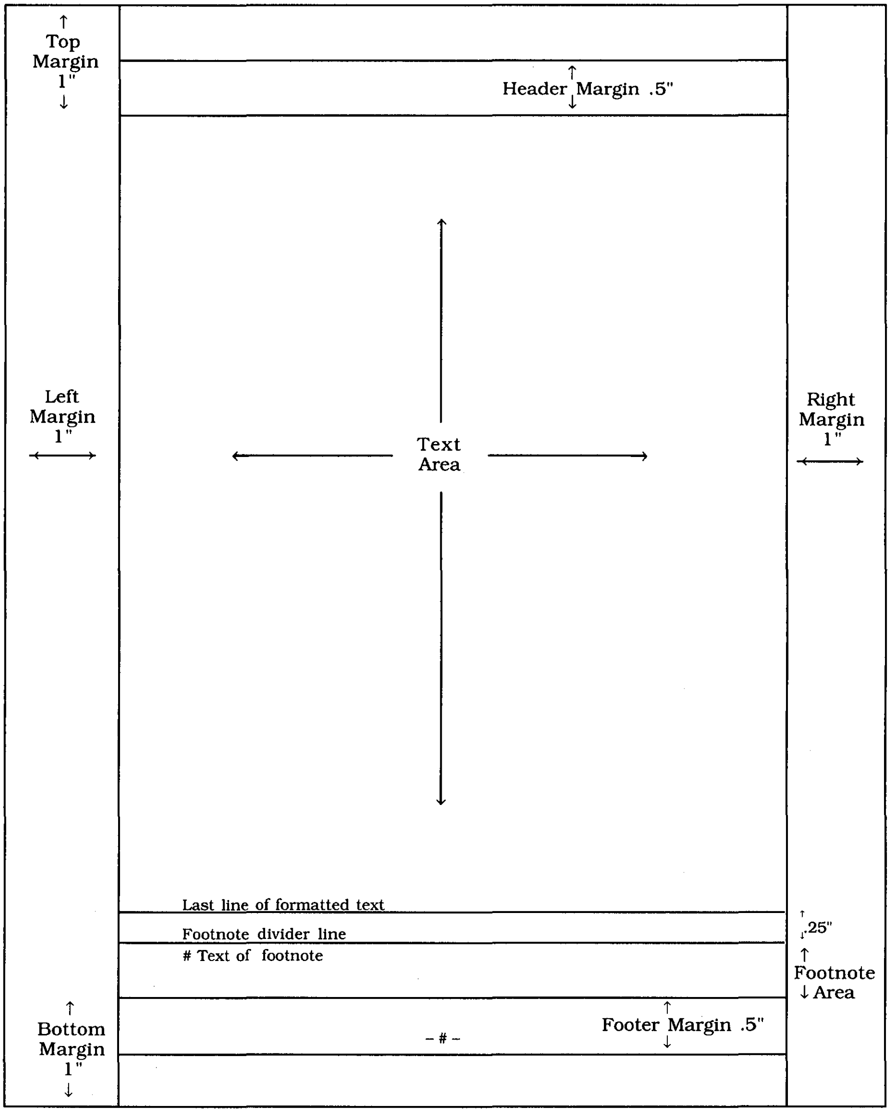


Figure 2.1: Default Page Layout

Table 2.2: Page Layout Defaults

Top Margin	1 inch from the top of the page
Bottom Margin	1 inch from the bottom of the page
Left Margin	1 inch from the left edge of the page
Right Margin	1 inch from the right edge of the page
Header	.5 inches from the top of the page
Footer	.5 inches from the bottom of the page
Paragraph Indent	The default setting is 0, which means the first line of each paragraph is *not* indented from the left margin.
Tabs	A tab is preset (on the ruler line) at column five, which also sets tabs at every fifth column (column 5, 10, 15, 20, etc.).
Justification	Left; text is automatically aligned at the left margin. The right margin is ragged.
Spacing	Single; the printer used determines the default spacing between lines. Normally, printers print six (single-spaced) lines per inch.
Font	All text prints in the default font (the font selected when the printer was installed with the SP-SETUP program).

Paper Size

Sprint supports paper of varying lengths and widths:

- 8.5 × 11 inch
- 8.5 × 12 inch
- 8.5 × 14 inch
- 11 × 14 inch
- A4
- Other (you specify the paper height and width)

The default paper size is 8.5 × 11. If you'll be printing on a different paper size, choose Layout/Document-Wide/Paper Size and then select the correct size. If the correct paper size is not listed on the menu, choose Other. Sprint will ask for the length and then the width of the paper on which you'll be printing.

Margins

The default page layout provides the following margins:

Left	1 inch from the left edge of the page
Right	1 inch from the right edge of the page
Top	1 inch from the top of the page
Bottom	1 inch from the bottom of the page
Offset	Set to 0 (no binding margin)
Header	.5 inch from the top of the page (header prints within the top margin)
Footer	.5 inch from the bottom of the page (footer prints within the bottom margin)

These margins affect the *printed* page only; you won't see the default or changed margin settings onscreen.

To change a document's left, right, top, or bottom margin, choose Layout/ Document-Wide and select the margin you want to change. Sprint will prompt you for the new margin. As with Precise Settings on the ruler line, you can use any valid dimension to specify the new margin.

To change where Sprint prints the header (page heading), choose Layout/ Header/Position. When prompted, specify where the header should appear (distance from the top of the page). To change the location of the footer (page footing), choose Layout/Footer/Position and specify the desired location (distance is measured from the bottom of the page).

Document-Wide Parameters (Using the Style Command)

If you choose a Document-Wide command, Sprint inserts a Style command (not to be confused with the Style *menu*) at the top of the file. If you choose more than one Document-Wide command, Sprint adds to this Style command. A Style command specifies the document-wide formatting parameters, like those you choose from the Document-Wide menu. Style commands affect all text until another Style command overrides the first. A Style command at the top of your document defines what the first ruler's margins settings stand for; subsequent ruler settings, however, are relative to the first ruler.

Some document-wide formatting parameters can't be entered from the menus *per se*. For example, you can format an area of text to print in a special font, but there's no document-wide menu command to specify a different font for the whole document. By entering a Style command at the top of your file and modifying it with the *Font* parameter, you can have your whole file print in the desired font.

You can use the **Other Format** command and type your own Style command to include the parameters you need to create the desired look for a document. You can use almost any Sprint parameter with Style commands, but a few parameters are used *only* with Style. Table D.4 in Appendix D (page 428) lists the parameters that are used only with the Style command. Page 129 explains how to enter **Other Format** commands.

Headings

Commands on the **Headings** menu let you format the text of document headings. Some headings are centered and print in large, bold type. Other headings are bold and left-justified. You can also decide whether you want *numbered* or *unnumbered* headings.

Numbered headings are those that Sprint numbers for you; you don't have to type chapter or section numbers. Numbered headings also mean you don't have to manually create a table of contents. Sprint keeps track of all your numbered heading commands and prints the headings and the page number on which each appears in the table of contents.

Unnumbered headings are formatted just like numbered headings, but are not numbered, do not include a word indicating type of section (like "Chapter"), and do not by themselves create a table of contents.

The following sections explain the two types of headings and provide examples of heading formats.

Numbered Headings

To create numbered headings in a file, choose **Style/Headings** and pick the desired heading format.

Chapter
Begins on the next odd-numbered page and prints a big, bold, centered, sequentially numbered heading, and creates an entry in the table of contents. The word *Chapter*, followed by the chapter number, prints six lines from the top of a new, odd-numbered page. The formatter then inserts

two blank lines and centers the title of the chapter. It inserts two more blank lines and then begins printing the chapter text.

Section
Inserts two blank lines, prints a big, bold, left-justified, sequentially numbered heading, and creates an entry in the table of contents.

Subsection
Inserts one blank line, prints a medium-large, bold, left-justified, sequentially numbered subheading, and creates an entry in the table of contents.

Paragraph
Prints a bold, left-justified, sequentially numbered subheading, and creates an entry in the table of contents.

Appendix
Formats the heading just like the Chapter command, but the formatter gives each appendix a letter (beginning with the letter *A*) and prints the word "Appendix."

AppendixSection
Formats the heading just like a Section command, but precedes the number with the letter assigned to the appendix followed by a number.

Note: The **Heading** commands are defined in the STANDARD.FMT file, so you can change the way they format your text. See Chapter 3, "Modifying and Creating Formats" and Appendix A, "Commands Defined in STANDARD.FMT."

Table 2.3: Sample Numbered Headings

Chapter looks like this at the top of an odd-numbered page:

Chapter 22

Advanced Formatting

Section looks like this:

1 First-Level Section Title

or like this, if you choose a Chapter command earlier in the file:

1.1 First-Level Section Title

Subsection looks like this:

1.1 Lower-Level Section Title

or like this, if you choose a Chapter command earlier in the file:

1.1.1 Lower-Level Section Title

Paragraph looks like this:

1.1.1 Lowest-Level Section Title

or like this, if you choose a Chapter command earlier in the file:

1.1.1.1 Lowest-Level Section Title

Appendix looks like this:

Appendix A

Options

AppendixSection looks like this:

A.1 Hardware Options

When you choose one of these numbered headings, you only need to type the title for the heading; the formatter does the rest. For example, to create a chapter heading, choose Style/Headings/Chapter. Sprint prompts you for the title of the chapter. Type the title of the chapter, press *Enter*, and Sprint inserts the onscreen CHAPTER command. When you print the file, the formatter automatically skips to the next odd-numbered page, drops down six lines, centers the text *Chapter 1* (if this is the first chapter command in the file), inserts two blank lines, and then centers the chapter title you entered. The next time the formatter sees a Chapter command in the file, it increments the chapter number by one. At the end of the document, the formatter creates a table of contents and prints the chapter number, chapter title, and the page on which each chapter begins.

If you rearrange the chapters within a document, Sprint automatically renumbers each chapter.

Tiered (Multilevel) Headings

If all headings in a file are of equal importance, you'll probably choose the same command for each heading. For example, all your headings might be formatted as Section headings. In this case, Sprint will give each heading a single number (for example, 1, 2, 3, and so on) and increment the number of each heading by one digit.

If you choose different types of numbered headings in a file, you'll see multilevel numbers assigned to your "lower-level" sections. For example, if you choose Section and Subsection commands, the Section commands will be a single digit (like 1, 2, 3), and the Subsection commands will have a two-level number (like 1.1, 1.2, 1.3, 2.1, and so on). This is part of Sprint's *parenting* concept. Chapter is the "parent" of Section, Section is the "parent" of Subsection, and so on. When you choose a numbered heading command, the formatter checks to see if you previously selected that command's parent. If you did, the formatter prints the number of the parent before it prints the number assigned to that heading. For example, let's say you create a chapter, and within that chapter, you use two Section commands, three Subsection commands, and then two Paragraph commands. Your sections would be numbered like this:

Chapter 1
1.1
1.2
1.2.1
1.2.2
1.2.3
1.2.3.1
1.2.3.2

This parenting concept applies to figures and tables, too. Chapter is the parent for these two formats. This means that if you choose a Chapter command and within that chapter choose Style/Table, the table number will be preceded by the chapter number (for example, Table 1.1). If you don't want the formatter to number your tables and figures this way (that is, if you want them numbered simply 1, 2 3, and so forth), you need to open your *backup* copy of the STANDARD.FMT file and then search for and delete these two lines:

```
@Parent(Figure = Chapter)
@Parent(Table = Chapter)
```

Unnumbered Headings

Unnumbered headings are formatted just like numbered headings; the formatter just doesn't print a number or section title (like "Chapter") next to the heading text.

The unnumbered heads in Sprint appear below the numbered ones in the Style/Headings menu. Their names are HeadingA, HeadingB, HeadingC, and HeadingD. The heads are ranked insofar as they get smaller and less significant, but because they have no numbers attached to them, they have no "parents."

As explained in the previous section, *numbered* headings force the formatter to create a table of contents; *unnumbered* headings do not. This is because you'll probably use numbered headings for large documents, and unnumbered headings for memos or other short documents that don't require a table of contents.

If you mix numbered and unnumbered heading commands in a file, the formatter will create a table of contents and print all headings (numbered *and* unnumbered) in the table of contents.

If you want only *unnumbered* headings *and* an automatic table of contents, you need to insert the formatter command MakeTOC near the top of the file. To do this, go to the top of your file and choose Style/Other Format.

Sprint prompts you for the format name. Type `MakeTOC`, press *Enter*, and then press *C* to tell Sprint that MakeTOC is a command. When you print the document, the formatter will include all of your unnumbered headings in a table of contents. (For more information about **O**ther Format, see page 130.)

Nesting Formats

Text can be affected by more than one format simulataneously. That is, you can "nest" formats within other formats. For instance, you could select a block that you want to keep together on a page and choose **L**ayout/**P**age Breaks/**G**roup Together on Page. Sprint inserts a BEGIN GROUP command at the beginning of the block and an END GROUP command at the end of the block. If you also want the block to print in the Display format, reselect the block, choose **S**tyle/**O**ther Format, and type `Display`.

Similarly, if some of the grouped text should print as a numbered list, select the text of the list and then choose **N**umbered from the **S**tyle/**L**ists menu. If you want some of the paragraphs to be preceded by hyphens rather than numbers, you could select this same text again and choose **H**yphens. Of course, you can also choose **T**ypestyle commands to affect any of this text as well. The following example shows you what your text might look like (onscreen) if you nested a variety of formats.

Note: When you nest formats, you must end the formats in *reverse order*. For example, if you begin the Group format and then nest the Text format within Group, you must end the Text format before you end the Group format. The following example illustrates this rule. If you don't end formats in reverse order of entry, you'll get an error message when you try to print.

Hint: To quickly fix wrongly nested formats, you might try using the **U**tilities/**P**otpourri/**T**ransposeLines command, which switches the current line with the one following it.

```
BEGIN GROUP
BEGIN TEXT, size 14 points, centered
CAUTION: FLAMMABLE!

DO NOT USE NEAR FIRE OR FLAME
END TEXT

WARNING:

BEGIN HYPHENS, spread 0
Avoid spraying in eyes.

Contents under pressure.

Do not puncture or incinerate.

Do not store at temperatures above 120 degrees F.

Keep out of reach of children.

Use only as directed.

BEGIN NUMBERED
Hold can about 12 inches from hair, with small red arrow on valve pointed
toward hair.

Press valve down firmly.
END NUMBERED
END HYPHENS
END GROUP
```

The onscreen example would print like this:

CAUTION: FLAMMABLE!

DO NOT USE NEAR FIRE OR FLAME

WARNING:

– Avoid spraying in eyes.
– Contents under pressure.
– Do not puncture or incinerate.
– Do not store at temperatures above 120 degrees F.
– Keep out of reach of children.
– Use only as directed.

1. Hold can about 12 inches from hair, with small red arrow on valve pointed toward hair.
2. Press valve down firmly.

Figures and Tables

Sprint's Figure and Table commands prompt for an optional caption and try to keep the text within the format together on a single page. Sprint automatically numbers figures and tables and produces a list of figures and list of tables at the end of the printed document.

You can create tables by setting tabs on the ruler (either *columnar tabs* on Sprint's standard ruler or *precise tab settings*), and then press the *Tab* key to move text to a tab stop. You can also create simple figures from your keyboard, or if you're using a printer that uses the PostScript page-description language, you can choose commands that draw boxes, horizontal bars, and key-cap graphics, as well as insert EPS (Encapsulated PostScript) graphic files. Refer to the "Graphics" entry in the "Menu Encyclopedia" chapter of the *Reference Guide* for more information.

Graphics Commands

Figures and tables often profit considerably from graphics elements like boxes and lines. If you have a printer that supports PostScript, you can easily incorporate these graphic elements.

To take advantage of Sprint's graphics PostScript support, choose Style/ Graphics. The following examples illustrate the effect of Sprint's special Graphics commands.

Table 2.4: Graphics Command Examples

We chose Draw Box to create the box around this table. Onscreen, the table format appears within the BOX format.	
Bar	▬ Sprint prompts for the height of the bar. For this example, we specified 6 points.
KeyCaps	KeyCaps look like this: Press A

Note: If you don't have a PostScript printer, you can still create fancy lines and boxes using the Utilities/Line Drawing command. (This command works best with monospaced fonts.)

Reserving Space for Figures and Tables

Often, you'll want to manually paste in a drawing or photograph. The Reserve Space command allows you to reserve a fixed amount of blank space for these "drop-ins." After you choose Style/Figure and type the figure caption, choose Layout/Page Breaks/Reserve Space. When prompted, type in the desired amount of space. You can use any of the dimensions listed in Table 2.1 on page 72. A sample figure format might look like this onscreen:

```
BEGIN FIGURE
RESERVE 2 inches
CAPTION Another Nice Figure
END FIGURE
```

If you want to reserve one or more pages for full-page figures, choose Layout/Page Breaks/Blank Page(s). Sprint prompts for the number of blank pages to insert. If the document contains Header and Footer commands, these blank pages will contain the header and footer lines but will otherwise be blank. (If you need completely blank pages, just substitute a blank sheet for one of the ones Sprint produced.)

Columns

Newsletters and brochures are often printed in two-, three-, or four-column format. Using Sprint's Layout/Columns commands, you can create from one to six *snaking* columns per page and specify the gutter (spacing) between columns. (*Snaking* means that Sprint formats the text of one column until it reaches the end of the page, and then continues printing the text at the top of the next column.)

To produce multiple columns on the printed page (you won't see the columns on the screen):

1. Type the text in a single column (that is, between ruler-line margins), select the text, and then choose Layout/Columns/Snaking Columns. When prompted, type the desired number of columns. For example, if you want the text to print in three columns, type 3.
2. Now enter the amount of space desired between columns.

   a. If you want evenly spaced columns (the same amount of blank space between columns), enter one dimension. For example, if you type 5 picas, Sprint will insert 5 picas of blank space between each column.

b. If you're happy with the default setting of one-half inch between columns, press *Enter* when Sprint prompts for the space between columns.

3. Sprint automatically fills a column and then begins a new column at the top of the page. If you want to force the formatter to break a column and begin a new column, choose Layout/Columns/Column Break.
4. If you change your mind about the gutter spacing, you can choose Columns/Gutter Between. Sprint prompts you for the Columns command you want to change, and then you can enter the new dimension for the gutter.

Footnotes, Endnotes, and Notes

Footnotes, endnotes, and unnumbered notes are useful ways to refer readers to supplementary information. The References menu provides the commands to produce such references.

Footnotes

Sprint's Footnote command automatically references text with a small, superscripted number and places the text of your footnote at the bottom of the appropriate page, column, figure, or table (assuming your printer can do this). Footnotes are numbered consecutively. Before the formatter prints the footnote text at the bottom of the page, it prints a solid line to separate the footnote text from the rest of the text on the page.

When you want to insert a footnote, place the cursor immediately after the text you want to reference and choose Style/References/Footnote. Sprint inserts the BEGIN and END FNOTE commands and places the cursor between the two commands. Type the text of your footnote and then press the *Right arrow* key to end the command. If you've already typed the text and later decide you want to make it into a footnote, you can select the text and then choose the Footnote command.

It may look a little strange with the footnote appearing in the body of your text, but the formatter will automatically number the footnote and place the footnote text at the bottom of the page when you print the file. If you move the paragraph containing the footnote to a different place in your file, the footnote automatically goes with the text, and so appears at the bottom of the correct page with the correct number.

Endnotes

If you want a document to contain notes at the end, instead of at the bottom of pages of your document, use Sprint's Endnote command. This command works the same way as the **Footnote** command, except that the formatter places the notes at the very end of your document (on the so-called Notes page) instead of at the bottom of each page.

You create endnotes in the same way you create footnotes. You can either choose **Endnote** from the **Style/References** menu and then type the text of your note, or you can type the text of the endnote, select it, and then choose the **Endnote** command. In either case, Sprint inserts the onscreen commands BEGIN ENOTE and END ENOTE.

If you want to print the endnotes elsewhere in the document (for example, at the end of a chapter instead of at the end of the document), choose **Style/Other Format** and type `Place Notes`. For instructions on placing endnotes at the end of a chapter or changing the way Sprint numbers endnotes, refer to the "Endnote" entry in Chapter 1 of the *Reference Guide,*

Notes

Notes are identical to footnotes, with one exception: Notes are not numbered. Instead, they appear with asterisks (*), both in the text and at the bottom of the page. The first note on a page has one asterisk, the second has two, and so on. Sprint inserts the onscreen commands BEGIN SNOTE and END SNOTE (which stands for "star note") when you choose this command.

Indexing

The commands listed on the **Index** menu tell the formatter to create and print an index at the end of your document. To reach the list of index commands, choose **Style/Index**. Sprint displays the following commands:

Word
Tells Sprint to print the current word (the word on which the cursor is positioned or the selected block of text) in its present location and also enter it in the index.

Reference Word
Lets you enter one or more words or a marked block of text in the index. Reference words do *not* appear as part of the text; they appear only in the index.

Master Keyword
Just like **R**eference Word but specifies text as a major topic in the index. Sprint prints this word in regular typestyle, but prints its page number in bold type. The indexed words do not appear as part of the text.

This command is useful for indexing text that appears in a glossary or definition of terms. For example, a document might often refer to *pie charts* and also include this term in a glossary at the end. You could use the **W**ord and **R**eference Word commands to index *pie charts* throughout the document, but should make the glossary entry a *Master Keyword,* since this is where you fully explain the term.

See
Lets you cross-reference index entries. This command lists a term in the index (but not in the text), followed by the italicized word *See,* followed by another entry. For example,

```
horseless buggies See cars, Model T.
```

Also See
Like the **S**ee command, lets you cross-reference related index terms. The cross-reference prints in the index only, not in the text. For example,

```
cars See also trains; planes.
```

Index Under
Allows you to index a word under another specified word. For example, you could index the number 12 so that it appeared in the index where the word *twelve* would appear (that is, under the *T*'s). These references print in the index only, not in the text.

Range of Pages
Lets you specify a range of pages for an index entry (which prints in the index only, not in the text).

For more complete information about these commands, see the "Index" entry in Chapter 1 of the *Reference Guide.*

Once you mark a word with one of the above commands, Sprint automatically creates an index; you do not have to tell Sprint to print one. The index has a two-column layout, and each letter category (*A, B, C,* and so on) is titled with the appropriate letter (in big, bold type, if your printer has this capability). The word *Index* is centered at the top of the first page and also appears in big, bold type. The formatter continues to number pages sequentially in the index.

If you choose any of the Numbered commands on the **Style/Headings** menu, the formatter includes the Index heading in the table of contents and lists its page number there.

Note: The format of the index is defined in the STANDARD.FMT file. If you want to change the way the index prints, you can make a backup copy of the STANDARD.FMT file, save it under another name (like MYSTYLE.FMT), and change the Index definition. Refer to the "Modifying a Format" section beginning on page 135 for examples of how to modify a Sprint command definition.

File Linking

When you're creating large documents like a manual, a book, or a comprehensive report, you might want to create separate files for each chapter or section. This makes editing faster—especially when you're using the search operations. But when you print, you want Sprint to automatically "link" your separate files and print one continuous document so that page numbers, footnote numbers, and all cross-references print correctly.

The Include command gives you this file-merge capability. Enter this command in one file (choose Style/Other Format, type `Include` followed by the file name you want to merge, and then press *C*), and the formatter automatically merges the contents of the file name you specify when it formats and prints your file.

Let's assume you have two files: GRIDBROS.SPR, which is a proposal you've customized for Grid Brothers, and BOILER.SPR, which is the boilerplate text you include in all your proposals. You could create the GRIDBROS.SPR file and use the Include BOILER.SPR command to automatically add the boilerplate text when you print your file. Of course, you could manually type this text in your GRIDBROS.SPR file or copy it from another file, but Include gives you a few advantages over either of these options:

- The Include command is generally a faster way to include information in a file. Instead of inserting a file within another file or selecting text in another file, copying it to the Clipboard, and then pasting it in another file, you only need to type a single command. And entering the Include command is certainly faster than retyping the text you want!

- When you store frequently used text in a single file, and you need to change the text, you change it *once*. For instance, if you include the file BOILER.SPR (using the Include BOILER.SPR command) in 10 customized proposal files, and the boilerplate text changes, you only have to change one file. If you included the boilerplate text by copying or manually retyping it into the 10 files, you'd have to change the text 10 times!
- When you keep text in a file that you include rather than copy or retype, you minimize the chance of typing mistakes.
- The Include command simplifies the task of reorganizing text. If all your text appears in a single file, and you need to move the text around (let's say Chapter 2 needs to be changed into Appendix C), you have a lot of searching, marking, and moving to do. If you use Include commands, you only move the one-line commands, not pages and pages of text.
- Several people can simultaneously work on the separate files. If the document were all one (large) file, only one person could safely edit the file.

The following example shows how to use the Include command to link files. Following this example, we provide a few guidelines on when and how to use the Include command.

```
BEGIN HEADER
Annual Report                                            September 2, 1988
END HEADER

BEGIN FOOTER
Revision 1.0                Page PAGE, t="%d"                MONTH DAY YEAR
END FOOTER

CHAPTER Introduction
INCLUDE INTRO.SPR

CHAPTER Executive Summary
INCLUDE EXECSUM.SPR

CHAPTER Findings
SECTION Preliminary
INCLUDE PRELIM.SPR

SECTION Interactive Research
INCLUDE RESEARCH.SPR

SECTION Problems
INCLUDE PROBLEMS.SPR

SECTION Conclusions
INCLUDE CONCLUSN.SPR

CHAPTER Recommendations
INCLUDE RECOMMND.SPR
```

Guidelines:

1. Create a *master* file, which serves as an outline of sorts. Choose all your major **Headings** commands within this file and enter the titles for your chapters, sections, subsections, and so on. The headings double as onscreen reminders of the contents of the files. (Of course, you can also put the chapter titles within the files themselves. In that case, you would *not* put them in the master file.) Also enter your **Header** and **Footer** commands in this master file.
2. Create a separate file for each chapter. If your chapters are lengthy, you can create different files for each of your major sections or subsections.
3. Within the master file, wherever you want the formatter to include the text of another file, choose **Style/Other Format**, type `Include` and the name of the file to be included, press *Enter*, and then press *C* (for command). As shown in the example, an Include command often follows a **Headings** command.
4. Avoid Include commands within files that are included by other files. For example, if you type `Include INTRO.SPR` in your master file, don't type an Include command within the INTRO.SPR file. The reason for this is basic housekeeping. If you can see all the files you're including by viewing a single file, it's a lot easier to find information when you're editing from a printed copy. You can easily tell which file contains the information you're looking for. If you nest (or, in this case, "bury") Include commands within other "included" files, you can't tell where your information is without opening multiple files and searching through each.
5. When you want to print your entire document, choose your **Print** command from the master file (that is, make sure the master file is the current file). If you're printing from the DOS command line, enter the master file name as part of your SPFMT command. For example,

   ```
   SPFMT MASTER.RPT
   ```

   The formatter begins formatting the master file, sees an Include command, reads and formats the file name specified in the command, and then returns to the master file. This process continues until all text has been formatted, and then the formatter begins printing the document. Pages will be numbered consecutively across all files, unless you've added a command like *Set page* to the file.

Note: For information on how to insert variables in page headers and footers (for example, the date or page number), see the "Variables" section beginning on page 115.

Centering Text

There are a variety of commands you can use to center text on the page. You can center text horizontally (between the left and right margins) and vertically (between the top and bottom margins).

To Center a Line of Text

Type the line to be centered and choose Style/Center. Sprint automatically centers the current line on the screen and between the left and right margins when you print.

To Center a Region of Text

Select the region to be centered and choose Style/Center. But remember, if you choose this command without previously selecting a block, only the line containing the cursor will be centered.

You can also insert a ruler above the text to be centered, and type a `C` on this ruler line. All text will be centered until you insert another ruler and override the centered format (for example, type a `J`, `L`, or `R`).

To Vertically Center Text

When you want text centered between the top and bottom margins on the *first page* of your document, choose Layout/Title Page. This automatically inserts a CENTERPAGE (.5 page) command at the top of the file, followed by a blank line, followed by a page break. On the blank line, type the text to be vertically centered. If you also want this text to be centered between the left and right margins, select the text and choose Style/Center.

If you don't want to create a title page, but want vertically centered text on pages other than the first, you can insert your own CenterPage command with Style/Other Format. First, determine the vertical center for your text. For example, if you want to center text between the top and bottom margins, the vertical center is *.5 page.* If you wanted to center text in the top half of the page, the vertical center would be *.25 page.* The vertical center is simply the point around which you want the text centered (measured from the top of the page).

To specify the vertical center, choose Style/Other Format and type `CenterPage`, followed by the dimension around which the text should be centered (for example, CenterPage .25 page) and press *Enter.* When prompted, type `C` to indicate this is a command.

Move the cursor to the line above the CenterPage command and choose **Layout/Page Breaks/Insert (Unconditional).** Then move the cursor below the last line of text to be centered and insert another Page Break command. With **Screen/Codes** set to On, the screen would look like this (^L is the code for an unconditional page break):

```
^L
^OCENTERPAGE .5 PAGE^N

Copyright (c) 1988 by Borland International. All rights reserved.
^L
```

This example centers the text *vertically* on the page. If you want the text vertically and horizontally centered, use the **Style/Center** command or use rulers above and below the text.

Page Breaks

Sprint provides a variety of commands that let you specify what text (and how much of it) prints on the current page. You can keep a block of text together, intentionally spread text over two pages, prevent a page break, force a page break, or tell the formatter to print one of two possible messages, depending on the amount of space remaining on the current page.

Sprint automatically prevents "bad" page breaks. For example, it won't isolate the first or last line of a paragraph, or separate chapter or section titles or headings from the text to which they belong. If Sprint has to separate text, it does so but will not allow single lines at the top or bottom of a page (often called widow and orphan lines).

When you want to influence the way Sprint breaks a page, you can use these commands on the **Layout/Page Breaks** menu:

Insert (Unconditional)
Forces Sprint to begin a new page. When you choose this command, Sprint inserts a bold, horizontal line to show you where the page break will occur.

Conditional Page Break
Allows the formatter to break the page where it normally would not. For example, the formatter will not break the page between a Headings command line and the text following the command line. When you choose this command, Sprint inserts a PGBREAK command.

Note: The Conditional Page Break command must appear at the beginning or end of a paragraph; you cannot insert the command in the middle of a paragraph.

Reserve Space
Inserts a fixed amount of blank space at the current cursor position. Sprint prompts for the desired amount of blank space; you can specify any of the valid dimensions listed in Table 2.1 on page 72 (for example, inches, points, lines, part of a page, and so on). When you choose this command, Sprint inserts a RESERVE command at the current cursor position.

The **R**eserve Space command is typically used when you want to paste in a figure after the document prints. When the formatter sees this command, it determines whether the specified amount of space will fit on the current page. If there's enough room on the current page, the formatter inserts the blank space immediately. If there isn't enough room on the current page, the formatter breaks the page, begins a new page, and inserts the blank space at the top of the new page.

Blank Page(s)
Inserts one or more blank pages in the file. If you've chosen a Header or Footer command in the file, this information will print on the otherwise blank page. When you choose this command, Sprint inserts a PGBLANK command at the current cursor position.

If you want a *completely* blank page, don't use the **B**lank Page(s) command. Instead, manually insert a blank sheet of paper after the file prints. If you want the formatter to account for this manually inserted page when printing page numbers (for example, the formatter prints page 2, you insert a blank sheet of paper following page 2, and you want the formatter to number the next printed page 4 rather than 3), you can still use the **B**lank Page(s) command, manually substituting completely blank sheets of paper after the file prints.

Group Together on Page
Forces Sprint to keep a region of text together on the same page. Select the text you want to keep together and then choose this command. Sprint inserts a BEGIN GROUP command above and an END GROUP command below the marked text.

Note that you can often achieve the same results by choosing **S**tyle/**M**odify and adding the *Group* parameter to the BEGIN format command.

Keep with Following Text
Ensures that the current line *won't* be the last line on the page, no matter what. For example, you might type `Send all inquiries to:`, insert a blank line, and then type an address. You want to make sure that the *Send all inquiries to:* line isn't the last line on the page; you want it to print with the

address text. Choose the Keep with Following Text command at the end of the line you want to keep together or on the line between the two paragraphs you want to keep together. Sprint inserts the onscreen command KEEPFOLLOWING. For example,

```
Send all inquiries to:
KEEPFOLLOWING
Borland International
P.O. Box 660001
Scotts Valley, CA
```

Widow-Orphan Control
Lets you specify the minimum number of lines required at the top and bottom of a page. If the formatter can't print an entire paragraph at the end of a page, you might want to force the formatter to print at least two lines of the paragraph before it breaks the page, and make sure that at least two lines of the paragraph print at the top of a page. This eliminates *orphan* lines at the bottom and *widow* lines at the top of a page.

You only need to choose this command once. Sprint inserts a Style command at the top of the file and includes the *WidowPrevent* parameter. This command remains in effect throughout the file. The default value is 1.

Adapting Text According to Page Breaks
In some instances, you might want to print a message if text won't fit on a single page. For example, you might want to print "Continued on next page" at the bottom of the page if you can't fit an entire table on a single page. To do this, choose Style/Other Format and type the following HaveSpace command:

```
HaveSpace 3 inches, N "Continued on the next page"
```

The *N* in this command means that if there is less than 3 inches left on the page, print the message "Continued on the next page."

Other times, the way you phrase something depends on the amount of space left on the page. For example, if the current page contains enough space to print a figure, you might say "The following figure..." but if the figure is going to appear on the next page, you might say "The figure on the next page...." You can use the HaveSpace command to let the formatter decide which text to print, based on its knowledge of the remaining blank space on the current page. For example,

```
HaveSpace 4 inches, N "The following figure @newpage", Y "The figure
below"
```

The *N* in the above command says if the page doesn't have 4 inches of space remaining, print "The following figure." The *Y* part of the command says if the current page has at least 4 inches left, print "The figure below." The rest of the sentence is not dependent on the amount of space, so it isn't included in the command text.

Some of the formats defined in the STANDARD.FMT file already include a page break command. Also, you might find yourself nesting one command in another. Therefore, you need to be aware of the following:

- PGBREAK (Conditional Page Break) overrides any surrounding Group (Group Together on Page) format. This means that if you include the PGBREAK command within a Group format, the PGBREAK command takes precedence. If the text following a PGBREAK command falls at the end of a page, the formatter will break the page, even if the text is part of a grouped format.
- If there are multiple PGBREAK and KEEPFOLLOWING (Keep with Following Text) commands in a row, the last one takes precedence. For example, if you begin an Address format (which includes the KeepFollowing command in its STANDARD.FMT definition) but insert a PgBreak command in your address text, the PgBreak command will allow the formatter to split the address text over two pages, if necessary.
- The Layout/Page Breaks/Insert (Unconditional) command overrides a Keep with Following Text command.

Spacing

Sprint has a variety of commands that let you vary the spacing between lines, between paragraphs, and between words. The following sections explain these commands and suggest ways to vary the spacing within a document.

Spacing between Lines

The commands on the Layout/Ruler/Line Spacing menu let you specify the amount of blank space between printed lines within a paragraph. You can choose

Single	A single blank line between each printed line
1.5	One and one-half blank lines between printed lines
Double	Two blank lines between printed lines
Other	You specify the distance between printed lines

Since line spacing is one of the formatter's functions, you won't see the effect of the Line Spacing command until you print.

Setting Spacing for the Entire Document
If the file has a single ruler, choose Layout/Ruler/Line Spacing and specify the desired spacing for the document. If you choose Other, you can specify any valid dimension listed in Table 2.1 on page 72 (for example, 1.2 lines, 15 points, and so on).

Setting Spacing for a Region of a Text
Insert a ruler above the region, choose Layout/Ruler/Line Spacing, and select the spacing value for the region. For example, if you want an area of text to be double-spaced, insert a ruler above this area, choose the Line Spacing command, and choose Double. When you want to resume single-spaced format, insert another ruler. The new ruler is a copy of the topmost one and therefore probably is set to single-spacing (the default). If you had changed the top ruler, the new ruler will be changed, too. Choose Line Spacing again to set the new ruler to Single.

Setting Spacing within a Format
Choose Style/Modify. When the cursor moves to the Begin command at the start of that format, modify the format by adding `Spacing` *`dimension`* (where *dimension* is any valid dimension listed in Table 2.1 on page 72).

For example, if you want to double-space text within a Description format, choose Style/Modify. When the cursor moves to the BEGIN DESCRIPTION command line, choose This Format and, when prompted, modify the format by entering `Spacing 2`. The formatter will double-space the text of the list. As soon as the formatter sees the END DESCRIPTION command, it will resume the spacing set before you started the DESCRIPTION format. (See Chapter 3 for more information on modifying formats.)

Spacing between Paragraphs

By default, Sprint automatically inserts a single blank line between paragraphs. If you want more or less space between paragraphs, choose Layout/Document-Wide/Inter-Paragraph Spread and enter the desired distance. You can use any of the valid dimensions listed in Table 2.1 on page 72. For example, if you type `1 inch`, Sprint will insert a whopping 1 inch of blank space between each paragraph.

Note: Sprint considers a paragraph to be any occurrence of two hard returns in a row.

Inserting Vertical Blank Space

The **Layout/Page Breaks/Reserve Space** command lets you specify blank space in any of the vertical dimensions listed in Table 2.1. For example, you could insert 180 points, 4 inches, or 2.5 lines of blank space. This is useful when you want to reserve space for a figure you'll paste in after a document prints. For more information about the **Reserve Space** command or the **Blank Page(s)** command, refer to page 96.

Spacing between Words

Gaps between Words

If you want text justified (aligned) at the right margin, Sprint normally has to insert extra blank space between words. The **Layout/Document-Wide/Word Spacing** command lets you specify the maximum number of extra spaces Sprint can insert between two words on a line. For example, if you don't want any more than two blank spaces between words, choose **Word Spacing** and type 2. If Sprint needs to insert more space to justify the line, it will stretch the space *between letters* rather than between words.

Note: Choose this command only once. Sprint will use the specified **Word Spacing** value throughout the file.

Tips: When the formatter sees a term containing a slash (for example, *and/or*), it views the term as a single word and will not automatically break the term after the slash. This means that if the end of a line contains the text *arrangement/rearrangement,* Sprint won't try to break up the phrase; it will place the entire phrase on the next line and insert the required amount of space between words and characters to justify the preceding line. This can result in large gaps between words and letters. To avoid this situation, move the cursor below the top ruler in your file, choose **Style/Other Format**, and type the following command:

```
TCT "/" = "/@!"
```

Press *C* for command. The TCT command tells the formatter to change the **Translation Character Table**, so that a / character not only causes the formatter to print this character, but also tells the formatter that it can break the line after it prints this character (that's what the @! command does). For a complete discussion of the TCT command, refer to the "TCT" entry in Chapter 2 of the *Reference Guide.*

Sprint always considers a hyphen a legitimate place to break a line. There are times, however, when you don't want a word containing a hyphen to be broken. The word *co-op* and the number *2-1/2* are two examples. There is a special command called Word that keeps characters together regardless of length or spaces.

If you want to keep a hyphenated word together

- choose **Style/Other Format** and type `Word` followed by the hyphenated word
- press *C* for command

Sprint will now recognize the word(s) governed by the Word command as an "unbreakable" unit.

Putting Extra Space between Words

Wide Spaces (Springs)

Headers and footers are good examples of why you might want to insert a *spring* (a flexible area of blank space) between text on a line. For example, you might want to print the chapter number on the left and align the chapter title at the right margin. You can't be very precise if you press the spacebar to insert the required amount of space, and if the length of your chapter titles varies, you won't achieve a consistent result. That's the purpose of the **Insert/Wide Space (Spring)** command.

This command automatically figures out how much space you need to align text at the right margin. For example, type `Hello`, choose **Insert/Wide Space (Spring)**, and then type `Goodbye`. You'll see (both onscreen and on the printed page) that Sprint automatically aligns the text *Goodbye* at the right margin:

```
Hello                                                          Goodbye
```

Let's say that you decide to add another word or phrase to the line, like *Hello again*, and want a wide space between all three. Move the cursor so that it is immediately after the word *Goodbye* and choose **Insert/Wide Space (Spring)** again. Sprint moves the *Goodbye* text to the center and aligns the *Hello again* text at the right margin. In essence, the **Wide Space** command does the following:

- Looks at the current left and right margins.
- Determines the center point and then aligns text to the right of *center*.

Once you insert a second **Wide Space (Spring)** command on a line, Sprint moves the right-aligned text so that it begins printing at the center and aligns the subsequently entered text to the right margin. If you insert another **Wide Space (Spring)** command and type another word or phrase (like *Goodbye again*), Sprint again moves the right-aligned text (*Hello again*) so that it's centered between the previously entered text and the right margin. For example,

```
Hello                 Goodbye                Hello again              Goodbye again
```

You can also use **Wide Space (Spring)** to exert more "pressure" on one side of the page. For example, if you want *Hello* to print two-thirds of the way across the page and *Goodbye* to print at the right margin, choose **Wide Space (Spring)** twice, type `Hello`, choose **Wide Space (Spring)**, and type `Goodbye`. The text prints like this:

```
                                    Hello                                   Goodbye
```

Specifying a Distance

Tabs on the ruler line let you insert horizontal blank space between words. This is useful in table formats, but if you want to insert a fixed amount of blank space between words on a single line, you don't have to insert a ruler. You can use the Hsp (**H**orizontal **SP**acing) command and specify the precise distance between two words. For example, `Hsp 9 picas` tells the formatter to move 9 picas (to the right) before printing the following text. You can also move in the opposite direction; `Hsp -2.5 characters` tells the formatter to back up (move to the left) two and one-half characters. The maximum backwards movement is to the start of the current word.

To insert a fixed amount of blank space on a line, choose **Style/Other Format** and type `Hsp` *n* (where *n* stands for the horizontal distance you want to insert). Type `C` for command. When the formatter prints the file, it will automatically insert the specified amount of blank space between the words on either side of the command. For example,

```
This is a test**HSP 3 PICAS**of Sprint's formatting functions.

This is another line**HSP -3 CHARS**of the test. It shows how you can move
the printer to the left.
```

Results in:

This is a test   of Sprint's formatting functions.

This is another liofnethe test. It shows how you can move the printer to the left.

Note: The specified backward distance can't be larger than the preceding word. For example, if you type *Turbo* followed by the command `Hsp -4` `inches`, the formatter will move left to the beginning of the word *Turbo* and no further. A negative Hsp command followed by a space character doesn't move the print head at all.

Kerning (Spacing between Characters)

If you're using a printer that supports the PostScript page-description language, Sprint automatically adjusts the spacing between certain pairs of characters that otherwise would appear with too much whitespace between them. That's because Sprint has a set of predefined "kerning pairs" in the file called POSTSCR.TCT.

You can add or change any kerning pair in POSTSCR.TCT or you can also kern "on the fly" using the Kern command.

The Style/Other Format/Kern command lets you specify the distance between two characters. This distance is typically expressed in *points*, but you can use any of the dimensions listed in Table 2.1 on page 72. For example, if you want to close up the extra space between the first three letters of the word WAVER:

1. Type `W`.
2. Choose Style/Other Format, type `Kern .1 em`, and press *C* (for command).
3. Type `A`, choose Style/Other Format, type `Kern .1 em`, and press *C* (for command).
4. Type `VER`.

The text onscreen looks like this:

```
WKERN .1 emAKERN .1 emVER
```

If you print this example, you'll see

WAVER

Note: The distance specified in a Kern command cannot exceed the width of the character to the left of the command. For a list of character widths for the chosen font, refer to your printer manual.

Printing Special Characters

Many printers can print characters that can't be easily typed from the keyboard. Using Sprint's Char command, you can print *any* ASCII character supported by the current font if you know its numerical equivalent. To use the Char command, choose **Style/Other** Format, type `Char` and the decimal number assigned to the character you want to print. Press *C* to insert the command.

For example, the current font may have an *em dash* (—), but your keyboard doesn't. If you want to print a real em dash, you can use the Char command and specify the decimal number that prints this character. The Times font on PostScript printers stores this character at decimal location 208, so you'd choose **Style/Other** Format and type

```
Char 208
```

where you want the em dash to print.

You can make this process a bit less foreign by assigning understandable names to the numbers that generate characters you'll be using. For example, you could define the name *emdash* to be equal to 208 by choosing **Style/Other** Format and entering `Set emdash=208`, and then pressing *C*. Then, whenever you need to print an em dash, you can enter `Char emdash` instead of `Char 208`.

Tip: POSTSCR.TCT, a file on the Sprint distribution disks, contains a number of Char commands. The Char commands are part of numerous "character translation" commands (TCT commands) that tell PostScript printers to automatically print "—" whenever you type --, "«" whenever you type <<, and so on.

If you want to print a special character that isn't available with the current font (but can be printed with an alternate font), you can combine the Char and **Typestyle/Font** commands. For example, to print the box "bullet" on a PostScript printer (a bullet is the symbol we use in this manual to begin each paragraph of an unnumbered list), you need to choose **Style/Other**, type `Char`, and specify the box's decimal equivalent, which is 110. Then you need to reselect the text and define the text as printing in the printer's special Dingbats font. For example:

```
I want to print a box here: DINGBATS CHAR 110
```

prints like this:

```
I want to print a box here: ■
```

To find out which special characters your printer supports, and the decimal equivalents of these characters, refer to your printer manual. If your manual lists codes in hex (H), binary (B), or octal (O), be sure to add the *H*, *B*, or *O* after the number (for example, CHAR 0d0H).

Repeating Text on a Line

If you want to *repeat text* so that it fills up whitespace, you can use the Insert/Repeating Character command. When Sprint prompts for the character, type the character you want to repeat. For example, to create leader dots in a table, choose Insert/Repeating Character and type a period. Sprint inserts a greater-than symbol and then the period on the current line.

For example, type `Shirts`. Choose Insert/Repeating Character and type a period. Type `$10.00 each`. Press *Enter*. Repeat these steps for Blouses at $18.00 each. The onscreen text looks like this:

```
Shirts>.$10.00 each
Blouses>.$18.00 each
```

This example prints like this:

Shirts .... $10.00 each

Blouses .... $18.00 each

It might be helpful to think of the Repeating Character command as working just like the Wide Space (Spring) command, except that it fills the gap between the text with a specified character.

Nonprinting Comments

Sometimes you may want to insert text in your file, but you don't want the formatter to print the text. This kind of text might be a reminder to yourself, an author query, or an explanation of some sort. There are two ways to enter these kinds of *comments*:

- Mark the text that you want to comment, choose Typestyle/Hidden. For example,

  ```
  BEGIN COMMENT
  Staff payroll information needs updating.
  END COMMENT
  ```

- Start a comment line with a semicolon (;) or a Tab followed by a semicolon. Note that the semicolon is *not* a command *per se*. You type it as text at the beginning of a line (you don't choose any menus or commands), or somewhere within the line if you want to comment text from some point within the line to the end of the line. The formatter will interpret the semicolon as the beginning of a commented line. For example,

```
;This line appears in the file but won't print.
Print this. TAB; But don't print this.
```

Note: You can use the semicolon only for single-line comments.

Comments marked with the Typestyle/Hidden command will only print if you print your document unformatted. You can, however, decide whether the formatter prints comments that begin with a semicolon. By default, these comments *are* printed. To instruct the formatter not to print single-line comments, go to the top of your file, choose **Other Format** from the **Style** menu, type `Style comments yes`, and press *C* for command. This tells Sprint not to print any lines that begin with a semicolon.

Cross-References

This section explains how to cross-reference text in a document without using *hard* references. That is, instead of typing something like `See page 45` (that's a hard reference), you can *tag* the page number and type `See page` *`tablepg`* (that's a soft reference). When Sprint prints the file, the tag *tablepg* is replaced with the correct page number automatically. This command keeps your references accurate throughout your file.

When you choose **X-Reference** from the **Style** menu, Sprint displays two options: **Define a Tag** and **Reference a Tag**. Using these two commands, you can label a variable and refer back to that variable. You can use tags anywhere in a document to identify a numbered entity—a chapter, table, figure, section, or any other variable within your text. Sprint allows you to reference these variable by page number or variable number.

Cross-referencing can be a vital part of creating sophisticated documents in Sprint. Sprint's sophisticated cross-referencing commands use *placeholders* to refer to numbers and pages of elements in your document. The actual number gets filled in (by Sprint) only at print time. This "delayed reference" is convenient—and necessary—because *you don't see the effect of your formatting commands until you print*. For example,

- When you enter any of the Headings commands, like Chapter, Section, Subsection, or Appendix, you don't see the number that Sprint assigns to that entry. For example, you can see the Chapter command in your file, but you don't see the actual chapter number until you print your file. To cross-reference a chapter number, then, you need to insert a placeholder (or "tag") that will be replaced by the actual number when you print.
- Sprint automatically numbers your pages, but you don't see the page numbers until you print. Page is a *variable* that Sprint automatically inserts in the footing line of every page. (The "Variables" section

beginning on page 115 explains variables in detail.) To cross-reference a page number, therefore, you must use a tag to it that Sprint fills in later.

- When you type the text for a figure or table caption (that is, when you choose **Style/Figure** or **Style/Table**, Sprint prompts for a caption, and you type the caption text), you know that Sprint assigns a figure or table number, but you don't see the number until you print. To cross-reference a table number, therefore, you must use a tag, not a "hard" reference.

Keeping these things in mind while you read this section will help you understand the way Sprint lets you cross-reference text. Sprint's cross-reference commands make it easy to avoid "hard" references. For example, if you enter "For more information, see Chapter 2," that's a hard reference. If you use hard references throughout your document and then rearrange your document so that Chapter 2 becomes Chapter 5, and Chapter 5 becomes Chapter 3, and Chapter 1 becomes Chapter 2, and so on, you have to go back through your document, find all the erroneous hard references (if you can!), and change them. If you've ever had to do this, you know what an onerous and error-prone job this is.

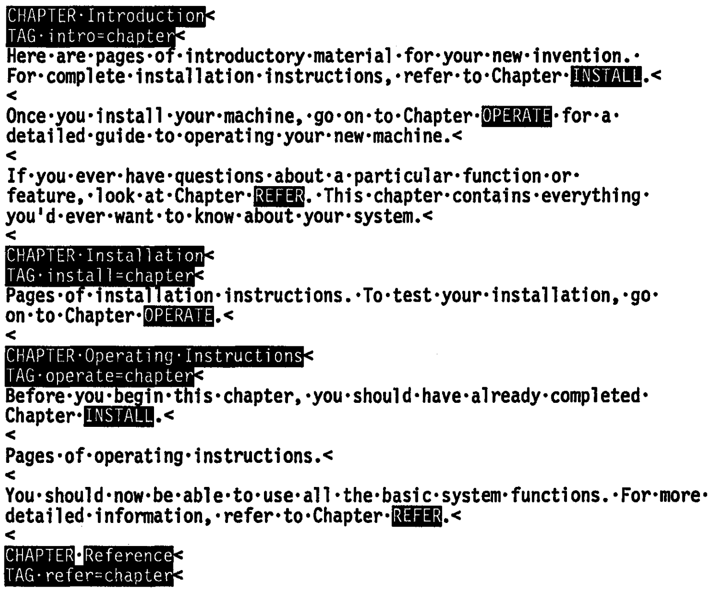
Sprint's cross-reference commands let you create *soft* references. These references let you tag (mark with **Define a Tag** command) text that you want to refer to, without having to know the number or name Sprint will assign to the text when it ultimately formats and prints your file. Using tags and then referring to these tags means that no matter how often you change a document's organization, your cross-references will always be accurate.

Tags and References

Since the concept of tagging is fairly abstract, we'll use several examples and then explain each example. This will give you a general idea of how Sprint's cross-referencing feature works. We encourage you to create these examples as we explain the text, so that you have a better understanding of how this feature works.

Example 1: Chapter References

Suppose you're creating a large document and want to cross-reference the chapter numbers throughout your document. You want to use *soft* references so that if you reorganize your document, you won't have to change your chapter references. To do this, you tag each chapter and then refer to the tag when you want to refer to the chapter number, like the example that follows. **Note:** We abbreviated the chapter text for this example.



```
CHAPTER·Introduction<
TAG·intro=chapter<
Here·are·pages·of·introductory·material·for·your·new·invention.·
For·complete·installation·instructions,·refer·to·Chapter·INSTALL.<
<
Once·you·install·your·machine,·go·on·to·Chapter·OPERATE·for·a·
detailed·guide·to·operating·your·new·machine.<
<
If·you·ever·have·questions·about·a·particular·function·or·
feature,·look·at·Chapter·REFER.·This·chapter·contains·everything·
you'd·ever·want·to·know·about·your·system.<
<
CHAPTER·Installation<
TAG·install=chapter<
Pages·of·installation·instructions.·To·test·your·installation,·go·
on·to·Chapter·OPERATE.<
<
CHAPTER·Operating·Instructions<
TAG·operate=chapter<
Before·you·begin·this·chapter,·you·should·have·already·completed·
Chapter·INSTALL.<
<
Pages·of·operating·instructions.<
<
You·should·now·be·able·to·use·all·the·basic·system·functions.·For·more·
detailed·information,·refer·to·Chapter·REFER.<
<
CHAPTER·Reference<
TAG·refer=chapter<
```

Figure 2.2: Tagging Chapters

How It Looks Printed

Note: Sprint would normally start each chapter on a new even-numbered page; we just condensed the printout to avoid confusion in this manual.

Chapter 1
Introduction

Here are pages of introductory material for your new invention. For complete installation instructions, refer to Chapter 2.

Once you install your machine, go on to Chapter 3 for a detailed guide to operating your new machine.

If you ever have questions about a particular function or feature, look at Chapter 4. This chapter contains everything you'd ever want to know about your system.

Chapter 2
Installation

Pages of installation instructions. To test your installation, go on to Chapter 3.

Chapter 3
Operating Instructions

Before you begin this chapter, you should have already completed Chapter 2.

Pages of operating instructions.

You should now be able to use all the basic system functions. For more detailed information, refer to Chapter 4.

Chapter 4
Reference

How We Did It

In the first line of the example, we used the Chapter command to create the chapter called "Introduction." Next, we choose **D**efine a Tag from the **X**-Reference menu. Sprint prompted

```
Name for new tag:
```

We entered `intro=chapter`, and Sprint displays this tag on the screen. The reason we added = *chapter* is to be sure that, when we used this tag later, the formatter would know to fill in the *chapter* number (not the page number, for example). In this case, the word *chapter* is a *variable* that contains the current chapter number. (For a list of all Sprint variables, see Table 2.6 beginning on page 119.)

Note: All tags either explicitly or implicitly use variables. If you don't give a variable with the tag, Sprint automatically uses the value of the *SectionName* variable.

We continued to enter *tagname=variable* commands for each of the four chapters shown in our example; we set a tag at each chapter and gave each tag an easy-to-remember name. You can use any one-word text in your tag—except for names already assigned to Sprint variables—but mnemonic abbreviations reduce the amount of typing and minimize errors.

Note: You *cannot* use actual variable names as tag names. See Table 2.6 for a list of all reserved variable names.

In the text of the Introduction chapter, we used the **R**eference a Tag command to create *forward references*—references to chapters found after the current chapter. Let's take the first reference in the Introduction chapter. We typed

```
For complete installation instructions, refer to Chapter
```

and then choose the **R**eference a Tag command from the **X**-Reference menu. Sprint displayed the following prompt:

```
Tag to reference:
```

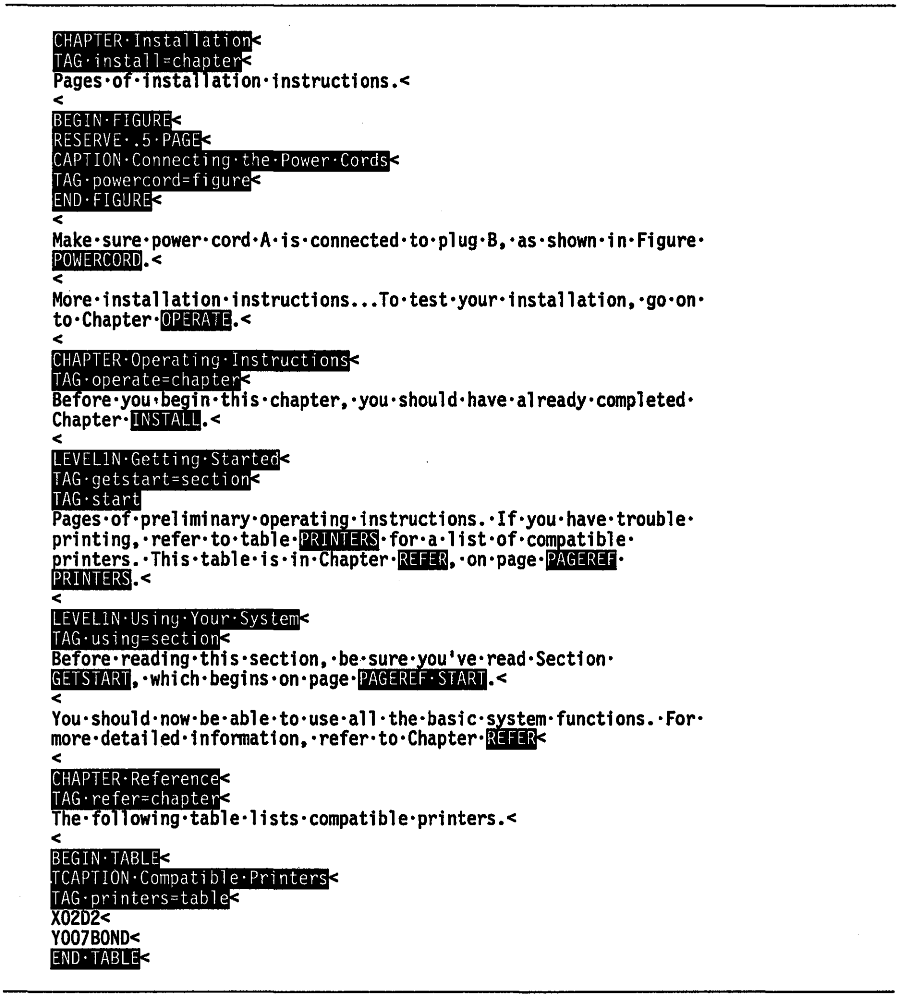
We entered `install`, which is the tag name we assigned to the Installation chapter. Then Sprint displayed the **R**eference By menu, which lets you specify either an **A**ssigned Number or a **P**age Number reference. We chose **A**ssigned Number, which means that Sprint replaces the reference command text with the correct number when it prints the file. This command refers to *any* Sprint-assigned number (except page number)—chapter number, table number, figure number, appendix letter, and so on.

Page Number, on the other hand, tells Sprint you want to reference the page on which the tagged text is located. You'll see how this works in the next example.

We continued this process for each reference included in the example, including the *backward* reference in the Operating Instructions chapter, which told the reader to refer to an earlier chapter in the document. Sprint supports both forward and backward references, and the method for entering them is identical.

Example 2: Page and Figure References

The following example uses parts of Example 1 as a foundation, and creates additional types of references:



```
CHAPTER·Installation<
TAG·install=chapter<
Pages·of·installation·instructions.<
<
BEGIN·FIGURE<
RESERVE·.5·PAGE<
CAPTION·Connecting·the·Power·Cords<
TAG·powercord=figure<
END·FIGURE<
<
Make·sure·power·cord·A·is·connected·to·plug·B,·as·shown·in·Figure·
POWERCORD.<
<
More·installation·instructions...To·test·your·installation,·go·on·
to·Chapter·OPERATE.<
<
CHAPTER·Operating·Instructions<
TAG·operate=chapter<
Before·you·begin·this·chapter,·you·should·have·already·completed·
Chapter·INSTALL.<
<
LEVEL1N·Getting·Started<
TAG·getstart=section<
TAG·start
Pages·of·preliminary·operating·instructions.·If·you·have·trouble·
printing,·refer·to·table·PRINTERS·for·a·list·of·compatible·
printers.·This·table·is·in·Chapter·REFER,·on·page·PAGEREF·
PRINTERS.<
<
LEVEL1N·Using·Your·System<
TAG·using=section<
Before·reading·this·section,·be·sure·you've·read·Section·
GETSTART,·which·begins·on·page·PAGEREF·START.<
<
You·should·now·be·able·to·use·all·the·basic·system·functions.·For·
more·detailed·information,·refer·to·Chapter·REFER<
<
CHAPTER·Reference<
TAG·refer=chapter<
The·following·table·lists·compatible·printers.<
<
BEGIN·TABLE<
TCAPTION·Compatible·Printers<
TAG·printers=table<
X02D2<
Y007BOND<
END·TABLE<
```

Figure 2.3: Referencing Page and Figures

How It Looks Printed

CHAPTER 1
Installation

Pages of Installation instructions.

Figure 1: Connecting the Power Cords

Make sure power cord A is connected to plug B, as shown in Figure 1.

More installation instructions...To test your installation, go on to Chapter 2.

CHAPTER 2
Operating Instructions

Before you begin this chapter, you should have already completed Chapter 1.

2.1 Getting Started

Pages of preliminary operating instructions.

If you have trouble printing, refer to Table 1 for a list of compatible printers. This table is in Chapter 3, on page 114.

2.2 Using Your System

Before reading this section, be sure you've read Section 2.1, which begins on page 113.

You should now be able to use all the basic system functions. For more detailed information, refer to Chapter 3.

CHAPTER 3
Reference

The following table lists compatible printers.

Table 1: Compatible Printers

X02D2
Y007BOND

How We Did It

In the Installation chapter of Example 2, we used the **Define** a Tag command to tag a figure number. We later entered

```
...as shown in Figure
```

and then entered the **Reference** a Tag command. When Sprint prompted for the name of the tag, we entered `powercord` and then chose **Reference** By/ Assigned Number. These steps are identical to those followed when we tagged a chapter number in Example 1.

The next new reference appears in the Operating Instructions chapter. We set a tag for each of the sections (Section headings) in that chapter. Tagging a numbered heading involves the same steps as tagging a chapter or figure. You can see that in the Using Your System section, we cross-referenced the Getting Started section. When Sprint printed the file, it replaced the `GETSTART` tag with the number assigned to the Section heading "Getting Started." In this case, the printed reference looked like this:

> Before reading this section, be sure you've read Section 2.1,...

We also include a *page reference* in this section. When you **Reference** a Tag and enter **Page** Number from the **Reference** By menu, the formatter knows to replace the reference with the page number on which the tagged text appears. For example, when the formatter sees the `PAGEREF GETSTART` tag, it looks for the `TAG GETSTART` command. When it finds the tag, it replaces the `PAGEREF` command with the page number of the "Getting Started" section. For example, the printed text would look similar to this:

> Before reading this section, be sure you've read Section 2.1, which begins on page 113.

In summary, you use the **Define** a Tag command whenever you want to tag a numbered entry or page number, and you use the **Reference** a Tag command when you want to refer to the tagged entry. If you want Sprint to replace the reference command with a page number, choose **Page** Number when Sprint displays the **Reference** By menu. Choose **Assigned** Number when you want Sprint to replace the reference command with the Sprint-assigned number of the entry (chapter number, section number, subsection number, appendix letter, and so on).

If these concepts and commands seem confusing or hard to understand, don't worry. Create a practice file and try using them. Once you use these commands and see their effects, you'll feel comfortable including them in your Sprint documents.

Another important part of cross-referencing is defining text variables, or strings. The following section discusses the concept of strings and explains how to use the **Define** Text Variable command in your Sprint files.

Variables

A *variable* is just what its name implies: something whose meaning varies. For example, as seen in the examples in the previous section, *Chapter* is a variable. The number Sprint assigns to a particular chapter depends on how many chapter commands you entered before it. *Day* is another, different type of variable. When you want to use this variable to tell Sprint to print the current day, Sprint looks at the date set by DOS and then inserts the day from that date when you print your file. *ChapterTitle* is also a variable; it references the title of the current chapter. Variables can contain either numbers or text.

Why would you use variables in a Sprint file? If your page headings include the current date, for example, you could insert the date variables (*month, day,* and *year*), and let the formatter determine the value of these variables each time you print. That way, you never have to change the date in your file.

Some variables are predefined by Sprint. Just like the formatting commands that use them, variables are either *built-in* or defined in the STANDARD.FMT file. These types of variables are defined in Table 2.5 (page 117) and Table 2.6 (page 119), which list built-in variables (*Page, Year, Month, Day*) and variables defined in STANDARD.FMT (*ChapterTitle, SectionTitle, MonthName*), respectively. To get Sprint to print the value of these variables, use the **Insert/Variable** command, and choose the variable you want to reference. Sprint then displays a "template" menu that shows the various ways it can print the variable; you pick the *template* you want Sprint to use.

Before we explain how to do this, look through the following tables, so you have an idea of the kinds of things Sprint calls variables. You can also create your own variables with the Define Text Variable command. We discuss this command in the section that follows the tables.

Table 2.5: Built-in Variables

Variable	Description
Day	The day of the month (values are 1–31).
FirstPage	The last page number of the introductory matter created by the formatter (such as the Table of Contents) plus 1. For example, if the table of contents is three pages long, the number of the *FirstPage* variable is 4. Normally, Sprint prints lowercase Roman numeral page numbers on the introductory matter, and then resets the page counter when it begins the body of the document. This means that the first page of your document begins on page 1. If you don't want Sprint to reset the counter, and would like your pages numbered continuously, insert the Style/Other Format command Set *page=firstpage* at the beginning of the file.
Font	The name of the current font (the full name, including dots). This does not contain that name of any printer attribute like bold or oblique.
Hour	The hour of the day (values are 0–23). Test for *Hour* >=12 to determine if it's am or pm.
Manuscript	The name of the *main* file being printed. The main file is the file that contains all the Include commands that tell the formatter to merge other files. See page 90 for information about file merging.
Minute	The minute of the hour (0–59).
Month	The number of the current month (1–12). If you want to print the *name* of the current month, use *MonthName,* a variable defined in the STANDARD.FMT file.
Page	The current page number. The formatter increments this value every time it begins a new page but, using the formatting command Set *variable=value,* you can set *Page* to any desired value. For example, enter the Style/Other Format command Set *page = 101.*
Plain	Has the value 0 if you're printing without formatting. You can use this variable to test for plain printing so you can create a special setup (like using a particular font) when printing plain.
Printer	The name of the printer being used (like "proprinter" or "thinkjet"). This is not the name of the .SPP file, but the name of the printer echoed on the status line when the formatter starts.
Size	The current font size, measured in printer units.

Table 2.5: Built-In Variables, continued

Variable	Description
SourceFile	The name of the current file being printed. This is usually the same as *Manuscript* unless your file has Include commands.
SourceLine	The current line number in the file being printed.
Version	The current version number of the Sprint formatter.
Weekday	The day of the week (Sunday = 0). The definition of this variable in STANDARD.FMT creates a template that prints the names of the days instead of the number.
Words	The number of words printed so far. The formatter increments this value for every word formatted in the main text (not including formats that float to the top or bottom of a document—like Index—or page headings). You can print the value of this variable in a message to get a word count.
Year	A two-digit number for current year (for example, 88 for 1988). All dates and times are retrieved from DOS as soon as the formatter begins formatting the file.

The best way to explain the use of these variables is to give an example of how they might be used. Many of the variables listed in Table 2.5 are especially useful in your page headings and footings. We discussed these commands in Chapter 8 of the *User's Guide* and will now expand the discussion to include variables.

Let's say you want Sprint to print the current date (month, day, and year) in the center of your page headings. You could type the date in your heading command, like this:

```
BEGIN HEADER
                            December 9, 1987
END HEADER
```

Each time you wanted to print your file, however, you'd have to remember to manually change the date. This is where variables come in handy. If you use the variables that stand for the current date, Sprint will supply the current date automatically, whenever you print your file.

Here's how to use the date variables in a Header command. First, choose the **Header** command and then choose **Insert/Variable**. When Sprint displays the list of variables, choose **MonthName**. When prompted to choose a template for Monthname, choose **None**. Sprint inserts the *MonthName* variable.

Press the *Spacebar* to create a space between the month and the day. Repeat the Insert/Variable command, choose **D**ay and then choose **A**rabic. Type a comma, type 19, and then insert the *Year* variable with an Arabic template. The **H**eader command text looks like this:

```
BEGIN HEADER
MONTHNAME, t = "%d", 19YEAR, t = "%d"
END HEADER
```

From now on, whenever you print your file, Sprint replaces the variable references with the date set by DOS. If today were the fifth of May, 1989, the header on each page would print as:

May 5, 1989

The following table lists the variables defined in the STANDARD.FMT file. An example follows this table.

Table 2.6: Variables Defined in the STANDARD.FMT File

Appendix	The current appendix letter.
AppendixTitle	The name of the last appendix started.
Chapter	The current chapter number. This variable contains the chapter number only if you've chosen the chapter command prior to referencing this variable; if you haven't created any chapters, the formatter prints a zero in place of the variable reference.
ChapterTitle	The name of the last chapter or appendix started.
Figure	The number of the last figure (which included a Caption command).
Footnote	The number of the last footnote or endnote.
MonthName	The name of the current month (January, February, ...).
Paragraph	The current paragraph number.
ParagraphTitle	The name of the last paragraph started.
Section	The current section number.
SectionNumber	The number of the last chapter, section, subsection, or paragraph started.
SectionTitle	The name of the last chapter, section, subsection, or paragraph started.
Subsection	The current subsection number.
SubSectionTitle	The name of the last subsection started.
Table	The number of the last table (which included a TCaption command).
Weekday	The name of the day (Sunday, Monday, ...).

Let's say you want Sprint to print the chapter number and chapter title in your page headings. This information varies, depending on the current chapter number and its title. You could manually insert page heading commands, and include the appropriate information each time you start a new chapter, but that's a lot of work. Instead, as part of your heading command, use the **Insert/Variable** command and choose the desired variable. If the variable you want to use isn't listed on the **Insert/Variable** menu, choose the **Other** command, and type the name of the variable you want to reference. The following example explains this method, and tells you how to include the chapter number and chapter title in a **Header** command.

1. Choose the **Layout/Header** command. Sprint prompts for the type of heading you want to create.
2. Choose **All** Pages. Sprint inserts the BEGIN HEADER and END HEADER commands.
3. Type the word `Chapter` followed by a space, and then choose **Insert/Variable/Other**. Sprint displays the following prompt:

   ```
   Variable to reference:
   ```
4. Type `chapter` and press *Enter*. This tells Sprint you want to print the number of the current chapter next to the word *Chapter*. Sprint then displays a list of *templates*, which tells the formatter how you want the chapter number to print (in Arabic numbers, Roman numerals, and so on).
5. Choose **A**rabic and press *Enter*. Sprint inserts a template parameter next to the chapter variable.
6. Type a colon (:) and press the *Spacebar* (following the chapter number).
7. Now choose **Insert/Variable/Other** again, and type `ChapterTitle` when Sprint prompts for the variable name. Choose **N**one when prompted for a template.

Your page heading command now looks like this onscreen:

```
BEGIN HEADER
Chapter CHAPTER, t = "%d": CHAPTERTITLE
END HEADER
```

When Sprint prints your file, it replaces the *Chapter* variable with the number of the current chapter, and the *ChapterTitle* variable with the name of the current chapter. For example,

Chapter 1
Installation

appears in your page heading as soon as the formatter sees the CHAPTER *Installation* command in your file. When it sees the next Chapter command, Sprint changes the header information to display "Chapter 2:" followed by the name you gave to your second chapter.

So far, we've explained Sprint-defined variables. Other variables can be user-defined; that is, *you* can create a variable and determine what it represents. This is useful when you want to cross-reference unnumbered text or assign values to specific text strings. The following section explains this powerful feature.

Defining Your Own Variables: String Assignments

If you've tried the Define a Tag command discussed on page 107, or the Insert/Variable command explained on page 115, you're familiar with the concept of *placeholders* being changed at printing time to something else. For example, to tag a figure so you can refer to it later (without knowing the figure number), you **Define** a Tag, give the tag a name, and make that tag equal to the variable *Figure* (for example, *mainmenu=figure*). The **Define** a Tag and **Reference** a Tag commands are an ideal way to cross-reference numbered items like figures, chapters, sections, and tables.

The "Variables" section introduced the concept of variables, and how you can get Sprint to replace a variable with its current value. For example, if you choose the Insert/Variable command and tell Sprint you want to refer to the time (by referencing the *Hour* or *Minute* variable), Sprint automatically prints the current time when it prints your file.

You don't have to be a programmer to define and reference your own text variables. You simply use the **Insert/Define** Text Variable command to create a variable, and then type a string of text telling Sprint what your variable means. A *string* is nothing more than a sequence of characters—a word, a phrase, or even a block of text. You can assign a string to something brief (like a name or address) or to something lengthy (like a paragraph of boilerplate text for a contract or proposal).

For example, choose **Insert/Define** Text Variable, and Sprint prompts

```
Name to give the variable:
```

Enter a name that represents something you don't want to type over and over again. For example,

```
Name to give the variable: aaa
```

Note: Variable names cannot start with numbers. So the name *3a*, for example, would have been illegal.

Sprint then prompts for the text you want the formatter to print whenever it sees the variable *aaa* in your file. For example,

```
Enter the text: Alliance of Angry Albanians
```

Now whenever you want to print the text "Alliance of Angry Albanians," you won't have to type it in the file. Instead, you can choose the **Insert/ Variable/Other** command and type `aaa`. When you print your file, the formatter will automatically replace *aaa* with Alliance of Angry Albanians. You've not only reduced the amount of typing you have to do, you've also minimized the potential for typos!

Note: You must define a variable before you refer to it; that is, the **Define** Text Variable command must appear before the **Insert/Variable** command that refers to your string assignment. It doesn't matter where you insert the **Define** Text Variable command, so long as you insert it *before* using the **Insert/Variable** command.

Another equally important function of strings and variable references is that they let you define specific information once; if that information changes, you have to make only one change in your file. Let's say you're writing a manual for a new software package called Strawberry. The package is still in development, and Strawberry is only a code name. You don't know when your company will decide on a real name, but you've got to start writing now. If you use the code name throughout your drafts, you'll have to go back and change every occurrence of Strawberry throughout the document. You could use Sprint's Search-Replace command to do this, but there's an easier and quicker way to resolve this problem.

Create a new file for your document, but before you begin writing, choose the **Define** Text Variable command. When Sprint prompts

```
Name to give the variable:
```

type

```
Name
```

Sprint then prompts

```
Enter the text:
```

Type the current product name (the text you actually want printed; in this case, `Strawberry`). Sprint inserts this information in your file, in the form

```
STRING name="Strawberry"
```

This onscreen command displays in reverse video, or in a different color if you have a color monitor. This makes your strings easy to find, in case you forget the exact text of your variable.

When you start writing, choose the **Insert/Variable/Other** command and type `name` whenever you want to refer to the product's name. When management decides on a real name for the product, all you have to do is change the text of your **Define** Text Variable command at the top of your file. Let's say the company changes the product name to "WonderCalc." All you have to do is edit the Define Text Variable command and replace "Strawberry" with "WonderCalc." You'll see the effect *when you print;* the Insert/Variable commands throughout your manual will be replaced with the name "WonderCalc."

Defining Your Own Numeric Variables

You use the Set command to define a brand new numeric variable. For example,

```
Set NumberOfTeethLeft=25
```

automatically creates a variable called *NumberOfTeethLeft* and assigns the number 25 to it.

You use your own numeric variables just as you use predefined ones. You reference them using the **Insert/Variable** command and change their values using Set (see the next section).

Changing the Value of Variables

As mentioned in the preceding sections, Sprint automatically prints the *current* value of a built-in or user-defined variable. If you want to change the value that Sprint prints or define your own numeric variable, you can use the formatting command Set and make the variable take on any numerica value you'd like.

For example, let's say your document is made up of five different files that are merged with the Include command. Also assume that each file contains Chapter and Figure commands. Sprint will automatically number the pages, chapters, and figures sequentially when you print your document. But if you want to print only one of these files (to proofread it, for example), your chapter and figure numbers won't be accurate. Since you're not merging all the files, Sprint will number the first page, chapter, and figure in this file 1. If you want Sprint to print the chapter and figure numbers as they'll be when you print the entire document, you can set the variables *Chapter* and *Figure* to their correct values.

Continuing this example, let's say that you want to print the last of the five files. The first chapter in this file is actually chapter 10, and the first figure in this chapter is actually Figure 18. (You know this because you've printed out the complete book before.) At the top of this file, choose the **Style/ Other Format** command two times and type the following two commands:

```
Set chapter=9
Set figure=17
```

You set these variables to one *less* than their actual value because when Sprint sees the Chapter and Figure commands, it automatically adds one to the current value. You can expand upon this principle, and set the variable *Page* to the last page number in the preceding (fourth) file.

This format for the Set command works for variables that have a *numeric* value. If you want to change the value of a variable created with the **Insert/ Define Text Variable** command, choose **Define Text Variable** again and give the variable a new value.

A Few Comforting Words

Don't worry if these cross-referencing concepts seem confusing at first; they're hard to understand only until you try using them. Just remember that, when you see a cross-reference command or variable on your screen, think of them as placeholders for numbers or text that can change at any time. It's only when you see the print-out, in which Sprint has filled in the values represented by the variables, that they will make sense. (In fact, that's a good way to learn about variables; refer to the tutorial chapter in this manual for a hands-on lesson.)

In summary, here are some general rules for cross-referencing text and using variables:

- Use **Define a Tag** to "tag" numbered elements, like figures, tables, chapters, sections, and appendixes. Tags refer only to numbers, not text.
- Use **Reference a Tag** when you want to refer to the name of a tagged element in your text. You can also use this command to reference the page number on which a tagged element appears. Choosing a **Print** command causes the formatter to replace each tag with a name and each tag reference with the number it assigned to the tag.
- Use the **Insert/Variable** command when you want Sprint to print:
  - The value of a built-in variable (like *Month, Day, Year*)
  - The value of a variable defined in the STANDARD.FMT file (such as *Chapter, ChapterTitle, Table, Figure*)

  - The string you assigned to your own text variables (as explained in the preceding examples)

- Choose the **Define Text Variable** command when you want to define variables that pertain to your word-processing tasks. Make these variables equal to a string (a sequence of characters enclosed in quotation marks). When you reference the variable with the **Insert/Variable** command, Sprint replaces the variable with the string you assigned. Remember to assign the string before you reference it!
- Use the Set command to alter the value of any numeric variable or to create a new one.

Other Formats

There are two kinds of *other* formats:

- Formats that affect a block of text.
- Formats that take effect at the current cursor position.

They're called other formats because they aren't listed on a Sprint menu, per se. (They are, for the most part, rarely needed.) You choose **Style/Other Format**, type the name of the format you want to use, and then tell Sprint whether the format should affect a region of text or be invoked at the current cursor position on the line. (If a block of text is selected when you choose **Other Format**, Sprint *assumes* you want the type of command that affects a region.)

Table 2.7 lists the formats that affect a region of text, and Table 2.8 shows the effect of each format. Following these tables, we explain how to insert format commands in a file.

Table 2.7: Sprint Formats Affecting Regions

Format	Description
Address	Indents text by one-half of the line length (text begins printing from the center of the line), and inserts two blank lines above and below the text of the address. Sprint keeps the text of this format together with the preceding text. The Address format ignores soft returns (where Sprint wordwraps); you must press *Enter* wherever you want to end a line.
Closing	Identical to Address (used for complimentary closings in letters, and the like).
Display	Leaves a blank line above and below the marked region, and indents the text one-half inch from the current left margin. Display prints your line endings exactly as you've entered them (ignoring soft returns). This is useful for documenting screen displays and other types of text that should be offset from the left margin, but should otherwise appear *verbatim*. You must press *Enter* whenever you need to end a line.
Example	Similar to the Display format, but uses a fixed-width or "typewriter" font if available. This is useful when printing sample computer programs.
FlushLeft	Prints the text so it is aligned at the left margin only.
FlushRight	Aligns each line at the right margin.
Quotation	Indents text one-half inch from both the left and right margins. Sprint wordwraps lines within this format and inserts a blank line above and below the quotation text.
Text	Does not affect the format of text, unless you include a format parameter such as *spacing*, *indent*, *spread*, *font*, and so on. The Text format is generally used when you want to create a special effect that's not defined by any other command (like double-spaced, indented text). For more information on format parameters, see page 429. For examples of how you might use the Text format, see the "Custom Formats for Part of a Document: The Text Format" section on page 148.
Undent	Prints the first line of each paragraph one-half inch to the left of the current left margin, and fills lines.
Verbatim	Prints text exactly as entered. Lines are not wordwrapped, nor indented; leading spaces are retained. (Normally, Sprint discards leading spaces in a line when it reformats your text.)

Table 2.8: Format Examples

Address

Send all inquiries to:

Borland International, Inc.
P.O. Box 660001
Scotts Valley, CA 95066

Closing

Sincerely,

Frank Borland
Governor, State of California

Display

Enter your password:
Enter your userID:

Fill in the blanks:

Name ____

Address ____

Continue? Type Y for yes; N for no.

Example

```
program Overflow
var
  A: integer
begin
  A := 30000 + 30000
end.
```

FlushLeft

Send all inquiries to:

101 Main Street
Anytown, USA

Table 2.8: Format Examples, continued

FlushRight

Today is Friday.
Yesterday was Thursday.
Tomorrow is Saturday.

Quotation

> "Age does not diminish the extreme disappointment of
> having a scoop of ice cream fall from the cone."
>
> —Jim Fiebig

Undent

This line is "undented" by one-half inch. The top line of each paragraph will print like the top line of this example, and subsequent lines in the paragraph will begin printing at the current left margin.

If you don't want a paragraph undented, press *Tab* at the beginning of the paragraph.

When you end the Undent format, text resumes printing at the left margin.

Verbatim

The following example is an excerpt from Shel Silverstein's classic tale, *The Giving Tree,* formatted with the Verbatim format.

```
Once there was a tree...

      and
    she loved
      a
    little boy.

And every day
the boy
would come

   and
he
    would
    gather
   her
    leaves

and make them
into crowns
and play king of the forest.
```

As explained in Chapter 3, you can create your own formats. Once defined, you can type the name of a custom format in response to the Style/Other Format prompt. (For instructions on creating your own formats, see Chapter 3, the "Defining a Unique Format" section (beginning on page 150).

Selecting Other Formats

Remember, in order for Sprint to recognize any of the commands in the preceding table, you must first choose Style/Other Format and then type the desired command. You cannot just type in the word `unjusttext`, for example, and expect Sprint to recognize this as the command.

There are two ways to specify an Other Format for a region of text:

1. Select the text you want to affect, and then choose Style/Other Format. Sprint prompts for the format you want to use. Type the format name (see Table 2.7 for the list of formats you can enter), and press *Enter.*

   Sprint automatically places a BEGIN command above the selected text and an END command on the line following the selected text. For example, type a paragraph, select it, choose Style/Other Format and type `Example`. Sprint inserts a BEGIN and an END EXAMPLE command, like this:

   ```
   BEGIN EXAMPLE
   Here's some text in the Example format.
   END EXAMPLE
   ```

   The example prints like this:

   ```
   Here's some text in the Example format.
   ```

2. If you haven't already typed the text you want to format (for example, you want to choose the format before you type the text):

   a. Choose Style/Other Format and type the name of the desired format.

   b. When prompted, press *R* for Region. Sprint then displays the following message in the status line:

   ```
   Press (B) for Begin command, (E) for End command, or ESC to cancel:
   ```

   c. Press *B* to begin the desired format, and Sprint inserts the correct BEGIN command. For example, if you enter Display, Sprint inserts the BEGIN DISPLAY command.

   d. Now type the text. Once you've typed the text, choose the format again (that is, choose Style/Other Format, type the format name and press *R*), and then press *E* to end the format.

If a format doesn't create the exact look you want, you can either modify it with the Style/Modify command, or edit the STANDARD.FMT file and redefine the format to better suit your needs. "Modifying a Format," beginning on page 135 in Chapter 3, briefly explains how to modify formats, and "Defining a Unique Format," beginning on page 150, introduces the idea of creating your own formatting commands.

Other Format Commands

As mentioned on page 125, the Style/Other Format command lets you specify a format for a region of text, or insert a command that takes effect at the current cursor position. This section discusses the latter—format commands that affect a specific character, cause the printer to move a specified distance, or tell the formatter to make a decision about some formatting aspect of the file and then print something based on that decision.

Table 2.9 lists all of the format commands you can invoke at the current cursor position. For complete information about all of these commands, see Chapter 2 of the *Reference Guide.*

Entering Other Format Commands

To insert a command listed in Table 2.9, choose Style/Other Format and type the text of the command (be sure the cursor is positioned where you want the command to take effect). When Sprint displays the prompt:

```
  Insert for Region (R) or Command (C):
```

press *C.* Sprint inserts the command at the current cursor position.

Unlike Sprint formats that affect regions, these formats do not insert a BEGIN and END command; instead, the command is immediately invoked at the current cursor position.

Table 2.9: Other Format Commands Not Affecting Regions

Case	Lets you set up form letters that print alternative text, depending on certain conditions that you specify.
Column	Lets you set up *parallel* (as opposed to *snaking*) columns.
Escape	Sends raw data to the printer.
HaveSpace	Causes the formatter to make a decision based on the amount of space remaining on a page (see page 97).
Hsp	Moves the print head (horizontally) a specified distance (see page 102).
Include	Inserts the contents of a specified file during printing (see page 90).
Incr	Increments a variable.
Kern	Adjusts the distance between two characters (see page 103).
Label	Sets a tag equal to the variable called *SectionNumber* (the current Assigned Number).
MakeTOC	Forces the creation of a table of contents (see page 82).
Message	Prints a message on the screen during formatting.
Modify	Modifies a defined format (see page 142).
NeedSpace	Causes the formatter to make a page break decision based on an amount of space needed.
NoteChapter	Places the number and title of the current chapter into the endnotes (see page 88).
NoteSection	Places the number and title of the current section into the endnotes (see page 88).
O	Overprints specified single letters.
Ovp	Overprints specified text (one letter or more).
Parent	Sets the parent of one variable to another variable (see page 81).
Set	Changes the value of a numeric variable or defines a new numeric variable (see page 123).
StringInput	Prompts the user to type a string during formatting.
Style	Sets global formatting parameters (see page 77).
TabDivide	Sets tab stops to create evenly spaced, tabbed columns.
TagString	Sets a tag to a text string.
Tct	Changes the value of one string to another (see page 100).
Template	Creates a numbering template for a numeric variable (see page 120).
Timestamp	Inserts current data and time when printed. (The format is like this: 4/29/89 3:32pm.)
Word	Keeps text together on a line (see page 100).
!	Allows a line break where one would not normally occur.
<	Starts a new line that prints on top of the previous line.

CHAPTER

3

Modifying and Creating Formats

Throughout the previous chapter, we've said that if you don't like the way a format affects text, you can change the format or create a unique format to produce the desired effect. This chapter explains both techniques.

Format changes All formats that use BEGIN and END commands can be modified to produce a different effect. For example, the Lists/Numbered format, by default, inserts a single blank line between each paragraph of a numbered list. You can change this format, however, to remove the blank lines. The discussion of modifying formats begins on page 135.

Custom formats When a Sprint format doesn't create quite the look you want, you can define your own format. Instructions for this begin on page 150.

Before you begin modifying or defining formats, you should be familiar with each format's default effect. The STANDARD.FMT file on the Sprint distribution disks contains the definition of many of Sprint's formats. This file is often discussed on the following pages. Once you have worked through this chapter, you should go to Chapter 4 for a detailed illustration of how to modify formats. Also refer to Appendix A for a complete list of all the formats defined in STANDARD.FMT.

STANDARD.FMT: The Formatter's Style Handbook

The formatter isn't naturally intelligent. It uses a comprehensive list of instructions to dictate what it should do when it sees a formatting command in your file. Some of these instructions are *built-in*—they're part of the Sprint program. Most of the formatter's instructions, however, are defined in the STANDARD.FMT file. Since nearly everything the formatter does is specifically defined in STANDARD.FMT, this file must be on the disk you use to format and print Sprint files; without STANDARD.FMT, the formatter won't run.

When Sprint formats your files, it uses the STANDARD.FMT file as its reference guide; when the formatter reads your file before printing and sees a formatting command in your file, it checks the STANDARD.FMT file to see what it should do.

For example, when the formatter finds a Chapter command in your file, it looks to the STANDARD.FMT file for a definition of Chapter. This definition tells the formatter to

- start the chapter at the top of an odd-numbered page
- insert 1 inch of blank space
- give the chapter a number
- center the word *Chapter* and the chapter number, and print this line in a big, bold font
- insert two blank lines, center the text of the chapter title, and print the title in a big, bold font
- create an entry for the chapter number and title in the table of contents
- insert two more blank lines, and then begin printing the text of the chapter

If you like the way Sprint's commands format your text, you'll never have to do anything with the STANDARD.FMT file. It's enough to know that the formatter uses this file to do its job. If you're curious about the content of STANDARD.FMT, open the file and page through the text. (It's a plain text file but, since the @-sign method of entering commands is necessary in .FMT files, the format may look a little strange to you.) You might want to look at the Chapter definition and see how the definition compares with the explanation we provided earlier in this section. You'll see that the Chapter definition specifies *BigCenteredHead,* which is another command defined in STANDARD.FMT.

If you want to change the way a formatting command affects your text, you can edit a version of the STANDARD.FMT file that's been saved under another name (say, MYSTYLE.FMT) and then change the command's definition. At the top of your document, you would choose the **Style Sheet** command from the **Layout/Document-Wide** menu and enter the .FMT you named and customized. For example, you could change the Chapter definition in MYSTYLE.FMT so that it centers the chapter number and title on a blank page, and then starts the text of the chapter on the following page.

Refer to Chapter 4 for more information on writing your own .FMT file.

Table A.1 on page 354 lists the commands defined in the STANDARD.FMT file. If this file doesn't contain a command that creates the look you want, you can open the file, copy it to another name (keep the .FMT extension), and create your own command in the new .FMT file to tell the formatter exactly how you want an area of your file formatted. That's part of the beauty of Sprint—you can create your own word-processing commands and functions. See page 150 for details on creating your own commands.

Note: Always work with a *backup* of STANDARD.FMT, never the master. You must have a working copy of this file for the formatter to run. If you accidentally change your original version of this file, immediately choose **File/Revert to Saved**; Sprint rereads the original version of the file from disk. If you have a customized style sheet that's *fully tested*, you can have Sprint automatically use it by renaming STANDARD.FMT (to ORIGINAL.FMT, say) and then renaming *your* file to STANDARD.FMT. Until you do this, you always have to use the **Style Sheet** command at the top of each of your files to print with your customized .FMT file.

Modifying a Format

Table 2.7 (on page 126) lists the formats defined in STANDARD.FMT and provides an example of each. There are a number of ways that you can change how any of these formats affect your text, and this section explains each of the methods you can use.

When you want to change how a format affects your text, you should decide how often you want the format to be changed. Based on this, you can

- Modify a single instance of a format (for example, modify a single Description format). To do this, use the **Style/Modify** command, or choose **Style/Other Format**, type the format name, and add the modifications to the format name. We discuss this method beginning on page 136.

- Modify all instances of a format within a file (for example, modify all Description formats that occur in a particular file). In this situation, you'll save time by entering the Other Format command called Modify (not to be confused with the Style/Modify menu command). We discuss this method beginning on page 142.
- Edit a backup copy of STANDARD.FMT (and then rename it) and modify the format definition so that the format changes affect all Sprint documents. This discussion begins on page 144.

Regardless of the method, modifying a format requires that you add or change format *parameters*. These parameters tell the formatter how you want the format changed from its original definition. For example, there are parameters that change margins, line spacing, the distance between paragraphs, the current font, and so on. Table D.6 on page 429 lists the valid format parameters you can add to any format that inserts a Begin command in your file (for example, any of the Lists formats, the Footnote, Table, and Figure formats, and so on). For a list of dimensions that can be included in a parameter, refer to Table 2.1 on page 72.

You can add format parameters to any format that inserts BEGIN and END commands. The following sections explain how to add format parameters.

Modify a Single Format

This section explains two ways to modify a single instance of a format (that is, how to modify a specific format within your file).

1. If you've already entered a format (that is, if Sprint has already inserted the BEGIN and END commands around your marked text), and then decide to change how that format affects your text:

   a. Choose Style/Modify. Sprint searches backward (toward the beginning of the file) until it finds a BEGIN command (for example, BEGIN NUMBERED).

      When Sprint finds a BEGIN command, it displays a menu that asks if you want to modify This Format (the format named in the BEGIN command) or a Previous Format.

   b. If you want to modify the currently selected format, choose This Format. If you want Sprint to continue searching for a different format, choose Previous Format until Sprint finds the desired format.

Once you choose This Format, Sprint displays the following prompt in the information line:

```
Modify by adding:
```

c. Type the parameter(s) you want to add. If you want to enter more than one, separate the parameters with a comma. For example,

```
Modify by adding: leftindent 3 picas, spacing .75, spread 1.5
```

The example parameters tell the formatter to indent the text 3 picas from the current left margin, "tighten-up" the text by inserting only .75 of a blank line between lines (rather than the normal single blank line for single-spaced text), and insert 1.5 blank lines between paragraphs (instead of the normal 1 blank line). You only add parameters that will change the format; don't bother typing parameters you want to keep.

Note: You can add parameters only to formats that insert BEGIN and END commands. The parameters will only affect text between the currently selected BEGIN and END format commands.

2. If you haven't yet entered the format, but know how you want to modify it:

You can enter parameters as part of an **O**ther Format command. This eliminates the need to choose Style/**M**odify to add the parameter(s) later.

a. As you would normally do, choose Style/**O**ther Format and type the name of the format you want to use. Before you press *Enter,* type a comma and the parameter you want to add. For example,

```
Format: Description, font Helvetica
```

This specifies the Description format, but also instructs the formatter to print the text in the Helvetica font. If you didn't add the *font Helvetica* parameter, the formatter would print the text in the default font.

b. If you want to add more than one parameter, type a comma after each parameter. For example,

```
Format: Description, font Helvetica, spacing 2
```

c. When Sprint prompts for either *R* or *C*, press *R*.

Note: You can only modify a format that affects a *region* (for example, a format that, when chosen, inserts a BEGIN and END command in the file). If you try to modify a format command that doesn't insert BEGIN and END commands (like the Kern command), you'll get an error message when you try to print your file.

Note too that to modify a format "on the fly" like this, you must use the **Other Format** command—even if the format has its own menu command (as Description does).

Remember, when you use the modify techniques described in this section, you are only affecting the current instance of the format. For example, if you modify a Description format as explained above, you are only affecting text in *this particular Description format.* Any other time you choose Description, the formatter will format the text with the normal Description format parameters.

If you want to modify all instances of a format within a file, use the Modify command as described in the next section.

If you want to *permanently* change the effect of a format (for example, modify the Description format so that in *every* Sprint file the Description format always prints text in the Helvetica font and double-spaces it), you should edit a copy of the STANDARD.FMT file and add to the Description format's definition in that file. See page 144 for instructions.

Note: If you can't easily create the "look" you want by modifying a format (for example, you want text indented 3 inches from the right margin, triple-spaced, printed in different font, and paragraphs numbered with Roman numerals), you can define a format of your own. See the "Defining a Unique Format" section beginning on page 150 for instructions.

Practical Example: Modifying the Description Format

This section provides a practical example of how and why you might modify a format. We take a look at the Style/Lists/Description format and change the indent value to get a wider gap between columns.

Note: The Description format automatically moves the left margin in (to the right) by .25 line. The first line of each paragraph, however, is *outdented* from this new left margin by .25 line and prints in bold type. This means that the first line of each paragraph begins printing at the *old* left margin, and the rest of the paragraph prints .25 line to the right. For example:

One The first number in a series.

Two The second number in a series. Two follows One, and precedes Three; Three is described in the following paragraph.

Three The third and final number in this series.

This format works well in many instances, but not always, as a look at the example that follows reveals. This example uses nested Description formats to format the body of a resume. In the first invocation of Description, the outdent looks fine. But in the second case (the nested Description format), we need to create a wider gap between the two columns.

Here's how the original output looks:

Objective To land a flexible, well-paying, undemanding job close to home.

Education Graduated with honors from Ima Flayke University.

Work History

1985 to present
Unemployed

December, 1984
Sam's Toy Shop
Santa's Helper

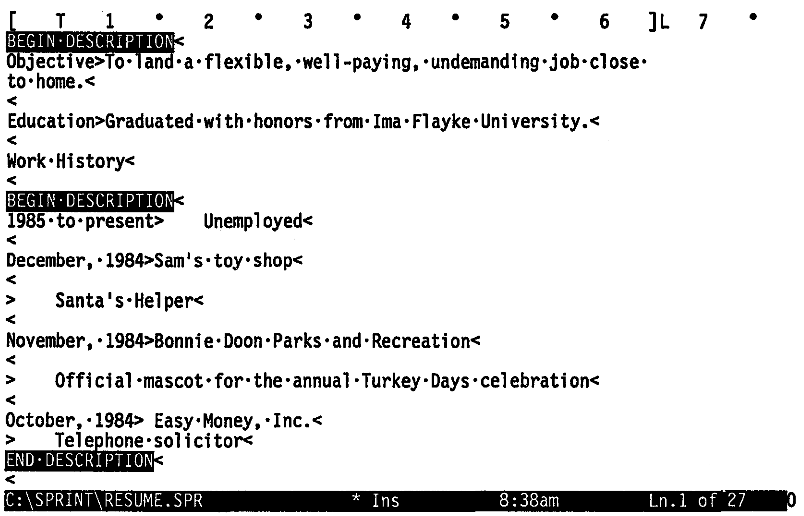
November, 1984
Bonnie Doon Parks and Recreation Department

Official mascot for the annual Turkey Days celebration

October, 1984 Easy Money, Inc.
Telephone solicitor

Personal Single, attractive, good sense of humor, always welcome at social functions. Willing to relocate to southern California.

Here's how the original file looks onscreen:



```
[   T   1   •   2   •   3   •   4   •   5   •   6   ]L  7   •
BEGIN·DESCRIPTION<
Objective>To·land·a·flexible,·well-paying,·undemanding·job·close·
to·home.<
<
Education>Graduated·with·honors·from·Ima·Flayke·University.<
<
Work·History<
<
BEGIN·DESCRIPTION<
1985·to·present>    Unemployed<
<
December,·1984>Sam's·toy·shop<
<
>    Santa's·Helper<
<
November,·1984>Bonnie·Doon·Parks·and·Recreation<
<
>    Official·mascot·for·the·annual·Turkey·Days·celebration<
<
October,·1984> Easy·Money,·Inc.<
>    Telephone·solicitor<
END·DESCRIPTION<
<
C:\SPRINT\RESUME.SPR             * Ins          8:38am          Ln.1 of 27     0
```

Figure 3.1: The Original Resume

After viewing the printed resume, you might decide that the "Work History" section needs a wider gap between the date of employment and the company/position columns. You don't want to change the Description format permanently, you want to change it only in this particular instance.

Before you modify a format, you should know—at least in general terms—how it's already defined in the STANDARD.FMT file. For example, STANDARD.FMT defines the Description format like this (unless you or someone else modified the STANDARD.FMT definition):

```
@Define(Description, indent -.25 line, WithEach "@b(@eval) @\",above 1,below 1)
```

Don't be alarmed at all the at-signs (@) and commands you don't know. @-signs are the ASCII alternative to menu-selected formats. They are covered in Chapter 3 of the *Reference Guide,* "Using @-Commands."

For now, all you have to notice is the *indent* parameter.

You can change the indent value for Description to be just about anything you like, since it's always relative to the current left margin. If you don't leave enough space to produce a gutter between the left and right sides of a Description format, the formatter prints the left side as usual and then begins the descriptive text on the next line. The first example of the formatted, printed resume shows this "stair-step" effect.

To *temporarily* change the *indent* value of the Description format, move the cursor to a line in your document (*not* in STANDARD.FMT) within this format, press *F10*, and then choose Style/Modify.

Sprint searches for the last format, and then displays the Modify By menu. Choose This Format, and Sprint displays the following prompt:

```
Modify by adding:
```

To change the *indent* parameter so that the left column is outdented by half a line (instead of .25 line), type the following response:

```
Modify by adding: indent -.5 line
```

Sprint adds this information to the BEGIN DESCRIPTION command line, and then returns the cursor to its location before you entered the Style/Modify format command. You don't have to modify the END DESCRIPTION command; the formatter automatically ends the Description format and any format parameters you may have changed or added to the BEGIN DESCRIPTION command.

Here's how the now-modified format prints:

Objective To land a flexible, well-paying, undemanding job close to home.

Education Graduated with honors from Ima Flayke University.

Work History

1985 to present Unemployed

December, 1984 Sam's Toy Shop
Santa's Helper

November, 1984 Bonnie Doon Parks and Recreation Department

Official mascot for the annual Turkey Days celebration

October, 1984 Easy Money, Inc.
Telephone solicitor

Personal Single, attractive, good sense of humor, always welcome at social functions. Willing to relocate to southern California.

Document-Wide Format Changes

You can modify a format so that it formats text differently throughout a document. Rather than modify each instance of the format (for example, using the **Style/Modify** command), you can insert the **Other** Format command *Modify* in the file, specify the format you want to modify, and add the parameters that create the effect you want. In essence, the Modify command tells the formatter how, in this document only, a particular format should affect your text.

Note: Unlike the **Style/Modify** method of modifying commands, which is used to change a single occurrence of a format, the Modify command can be used to change all subsequent occurrences of a format.

Note: Some commands are built-in, and cannot be modified. For a list of these commands, see Table B.1 on page 362.

The example in this section modifies the Quotation format, which normally indents the text .5 inch from the left and right margins, inserts a blank line above and below the quotation text, and single-spaces the text. We'll show you how to modify this format so that throughout a document, the Quotation format will double-space the text and print it in Helvetica.

Note: Before you insert a Modify command in an actual file, you should be familiar with the default effect of the format you want to modify (the effect as defined in STANDARD.FMT), and with the format parameters listed in Table D.6 (page 429).

There are two ways to create a document-wide format change:

1. You can copy the original format definition in the STANDARD.FMT file, paste the definition into your text file, edit the definition as necessary, and then change the text @Define (the command that begins a format definition) to @Modify.
2. You can choose **Style/Other** Format and type `Modify`, the name of the format you want to modify, and the parameters required to change the format.

We explain both methods in the following sections.

Copying the Definition from STANDARD.FMT

This method reduces the risk of typos in a Modify command, and starts you off with a command that you know works already. If you copy a definition from STANDARD.FMT, you'll end up with an @-sign command in your file, but don't let that concern you. The @-sign version of Sprint commands is usually only slightly different from the menu version. (For more on @-sign commands, refer to Chapter 3 of the *Reference Guide*, "Using @-Commands."

1. Open a copy of the STANDARD.FMT file and find the command that defines the format you want to modify. For example, search for

   ```
   @Define(Quotation,
   ```

2. Select the text of the entire definition.
3. Copy the definition to Sprint's Clipboard, switch back to your document file, and then paste the definition near the top of the file. Be sure to paste the text *before* any command that uses the command you're about to modify. For example, paste the `@Define(Quotation,` text before any BEGIN QUOTATION commands in your file.

   Note: You *cannot* use Modify to change how a format works halfway through a document. If you want to do this, either modify each occurrence of the format with the **Style/Modify** command, or define a new format (see the "Defining a Unique Format" section beginning on page 150). Remember, the Modify command must appear before the first occurrence of the format.
4. Change the word *Define* to *Modify*. For example,

   ```
   @Modify(Quotation, margins +.5 in, above 1, below 1, spacing 1)
   ```

5. Change the desired parameter(s). For example, change `spacing 1` to `spacing 2`. This tells the formatter to double-space text within the Quotation format.
6. Before the closing parenthesis, add any desired parameters and make sure you insert a comma to separate parameters. For example, after `spacing 2`, type a comma and add the parameter `font Helvetica`. The sample Quotation format definition now looks like this:

   ```
        @Modify(Quotation, margins +.5 in, above 1, below 1,
                spacing 2, font Helvetica)
   ```

   When you print the file, all text within the modified format will print as you specified with the Modify command.

Using Style/Other Format

If you don't want to copy a definition from STANDARD.FMT, you can use the **Style/Other Format** command to insert a Modify command in your file. If you choose this method, we recommend that you open a window to display STANDARD.FMT while you insert the Modify command; that way, you won't forget what is or isn't in the default format definition.

1. Open a window and then open STANDARD.FMT in that window.
2. Search for the command that defines the format you want to modify. For example, search for *Quotation,*.
3. Switch to the window containing your document file and move the cursor near the top of the file. The Modify command that you insert must appear before any use of the command you're modifying.
4. Determine whether you'll need to change any of the parameters listed in the definition, or whether you just need to add some additional parameters.
5. Choose **Style/Other** Format. At the prompt, type `Modify`, the name of the format you want to modify, and the parameters you want to change or add. Use the definition displayed in the other window for reference. Using the Quotation example explained earlier, you would type

   ```
   Modify Quotation, spacing 2, font Helvetica
   ```

   You don't have to retype the entire format definition. You only add or change the parameter(s) required to create a special effect. The formatter changes the format only as specified in the Modify command. Using the example above, you wanted to change the spacing, and tell the formatter to print Quotation text in a Helvetica font. The margins are fine and so is the amount of blank space inserted above and below the format; so you don't have to retype those parameters.
6. Separate parameters with a comma, and then press *Enter* when you complete the Modify command.
7. When prompted, press *C* for command. Sprint will insert the Modify command in the file. When you print the file, all text within the modified format will print as specified in your Modify command. For example, all text within the Quotation format will be double-spaced and in the Helvetica font.
8. Close the STANDARD.FMT file when you complete your Modify command(s). If you accidentally changed anything in STANDARD.FMT, Sprint will ask if you want to save the changes. Answer No.

Format Changes to All Sprint Files

To make permanent changes to a format (that is, to tell the formatter to change a format's effect in *all Sprint files*), copy the STANDARD.FMT file, rename it, change the desired format(s) in the copy of STANDARD.FMT, and then use the **Layout/Document-Wide/Style** Sheet command in your files and specify the name of the new format file. For example, you could

- Open STANDARD.FMT.

- Choose **Write As** and type `MYFORMAT.FMT`.
- Edit MYFORMAT.FMT and change the desired format(s).
- Whenever you want a file affected by the changes to MYFORMAT.FMT, choose **Layout/Document-Wide/Style Sheet** and type `MYFORMAT.FMT`. This tells the formatter to use MYFORMAT.FMT instead of STANDARD.FMT when it formats the file.

Warning: Don't edit STANDARD.FMT; edit only *copies* of this file! You must have a working copy of STANDARD.FMT in order to print a Sprint file! You should also refer to Table D.6 (page 429) for a list of format parameters.

Let's say that you want to force Sprint to automatically place Figure formats at the top of a page. The default definition of Figure doesn't say anything about *placement*; when you choose the **Figure** format, Sprint immediately inserts the figure text and the prompt for the figure caption. If you want all Figure formats *in all Sprint files* to appear at the top of a page:

1. Copy or write the STANDARD.FMT file to another file (the file name must have the .FMT extension). For example, write the file as MYFORMAT.FMT.
2. Open MYFORMAT.FMT and search for the line that begins with *@Define(Figure,*.
3. At the current cursor position (immediately after *@Define(Figure,)*, type

   ```
   above page
   ```

   followed by a comma (for example,

   ```
   @Define(Figure, above page,
   ```

4. Save the MYFORMAT.FMT file.
5. Choose the **Layout/Document-Wide/Style Sheet** command and type `MYFORMAT.FMT` in any file that should print figures at the top of the page.

That's all there is to it. If, after changing this file, you find that some people prefer figures in-line with the text while others like figures at the top, use STANDARD.FMT for files with in-line figures and MYFORMAT.FMT for files with top-of-page figures.

Style Sheets

In Sprint terms, *style sheet* refers to the file the formatter will use to interpret the formatting commands you've chosen. The default style sheet is STANDARD.FMT, which defines all the formatting commands you see listed on the menus and a variety of others (see Table A.1 on page 354 for a complete list).

If you want Sprint to use a file other than STANDARD.FMT when formatting your files, you need to choose **Layout/Document-Wide/Style Sheet** and type the name of the desired file. Sprint inserts the Format FILENAME.FMT command on the first line of the file, above the ruler line. Do not put anything above this command line! The formatter will ignore any Format command that is not at the top of the file (you'll see a warning message to this effect).

Why have more than one style sheet? Perhaps you create several types of documents, and each type has its own format requirements. For example, let's say that you produce internal specifications as well as reports that are distributed outside the company.

The following scenario shows why you might have more than one style sheet.

- Page, figure, and table numbers for internal specifications should be numbered sequentially. In reports, however, you want page, figure, and table numbers to include the number of the current chapter.
- Tables in internal specifications should always appear at the top of the page. Tables within a report, however, should print in-line with the preceding text.
- Section titles (Level1n heads) in internal specifications should print in the printer's default type size, not in a large one. For reports, however, you want section titles to be big. You also want the section number and title to print in a Times font, rather than the default font.

In this case, it would be convenient to have two style sheets: one for printing spec sheets, another for reports.

Here are suggestions to accompany this scenario:

1. Make two copies of the STANDARD.FMT file. Name one of the copies INTERNAL.FMT and name the other copy REPORTS.FMT.
2. Edit INTERNAL.FMT as follows:

   a. Delete the following lines:

   ```
   @parent(figure = Chapter)
   @parent(table = Chapter)
   ```

   These lines tell the formatter to precede figure and table numbers with the current chapter number.

b. Search for the Table definition and add the `above page` parameter to this definition. This forces all Table formats to the top of the next page.

c. Search for the Level1n macro and change *MedLeftHead* to *FlushLeft.*

3. When you create an internal specification, choose **Layout/Document-Wide/Style Sheet**. When prompted for the name of the style sheet to use, type `INTERNAL.FMT`. Be sure to do this at the very top of the file.
4. Edit REPORTS.FMT and make the following changes:

a. Search for the line *@Parent(table = Chapter)*. On the following line, type the following command:

```
@Parent{page = Chapter}
```

b. Search for the *BigLeftHead* definition. (The Level1n command uses this command to format section titles.) Following the *Big* parameter, add `font Times,` (be sure to add the comma to separate this parameter from the next).

c. When you create a report file, choose **Layout/Document-Wide/Style Sheet**. When prompted for the name of the style sheet to use, type `REPORTS.FMT`. Be sure to do this at the very top of the file.

If you follow these guidelines, you won't have to modify formats all the time. You have a style sheet for each of your different document types. You also have the STANDARD.FMT file in its original state. You may want the formatter to use this file to format your memos, letters, and so on, in which case you do not have to choose the Style Sheet command. If the formatter doesn't see a Style Sheet command at the top of a file, it automatically uses STANDARD.FMT.

For a detailed example of creating a unique .FMT file, refer to Chapter 4.

Creating Your Own Formats

Sprint's **Style** and **Layout** menus provide enough document style and format options to satisfy most word-processing applications. You can

- set left, right, top, and bottom margins; tabs; and paragraph indentation
- determine how text will be aligned
- design page headings and footings
- create divisions such as chapters, sections, and appendixes
- automatically generate a table of contents
- change typestyles, fonts, and type sizes

- index a document
- insert footnotes and endnotes
- cross-reference text

At some point, however, you may see the need to format *part* of a document but can't find a "predefined" format to create the desired effect. In this situation, you can use the special Text format on just part of your document and, by adding the appropriate format parameters, create a format to produce the desired effect. The following section explains how to use the Text command.

The Text command is handy for occasional customization of a format. But if you need to use a changed format a lot, you'll want to use the Define command to create a brand-new format that you can then use by name, like any other. You do this by using the Define command, give the format a unique name (that is, unique to the definitions listed in STANDARD.FMT), and specify the parameters that create the look you want. The "Defining a Unique Format" section beginning on page 150 explains how to define new commands.

Custom Formats for Part of a Document: The Text Command

The Text command lets you create a customized format for a particular area of text. By itself (without any format parameters), the Text command won't do anything to your text. In essence, Text is a "do-nothing" command that is available exclusively to be modified. You can enter any number of valid format parameters (as listed in Table D.6 on page 429) to create the desired effect.

As an example, let's say you want to indent all text in a region (that is, temporarily widen the left or right margin). There are a couple of commands that affect the margins, such as Display and Quotation. Display doesn't fully justify lines, though, and Quotation affects both the left and right margins. What if you want to move the left margin in (to the right) 7 picas (or 2 inches, one-half line, or whatever), double-space the text, and justify the lines? You'd use the Text command, and then modify it to include the *LeftIndent*, *Justify*, and *Spacing* parameters.

The following example shows how the Text command creates a left margin 2 inches from the current left margin and double-spaces the printed text.

> This text automatically appears 2 inches from the current left margin, and is double-spaced. You don't have to know the column number of the left margin; you can just tell the formatter to add 2 inches to this setting before printing the marked text. You don't have to insert an extra ruler, either. Just add parameters to the Text command.

To create this printed example:

1. Type the text to be affected. Don't indent the text yourself or vary the line spacing.
2. Mark the text.
3. Choose **S**tyle/**O**ther Format.
4. When prompted, type `Text`, press *Enter*, and then press *R*.
5. Once Sprint inserts the BEGIN and END TEXT commands around the marked block, choose **S**tyle/**M**odify.
6. When Sprint displays the `Modify by adding:` prompt, type:

```
LeftIndent +2 inches, Spacing 2, Justify Yes
```

Note: If you prefer, you can choose **Style/Other** Format, type `Text`, and follow this format name with the parameter(s) you want to add to the format. This eliminates the need to choose **Style/Modify** once you've inserted the Text format.

Inserting the Text format and modifying it to format text a certain way is similar to creating your own formats. Modifying the Text format, however, is the easier of the two ways to do what you want and is best suited for occasional use. If you find yourself using the same modified Text format often, you should consider defining the modified format as a brand-new format in the STANDARD.FMT file. The following section explains how to do this.

Defining a Unique Format

You can create your own format style with the Define command. One way to do this is to find a format command in the STANDARD.FMT file that has at least one formatting function in common with the functions you want to perform. Once you find such a command, copy its definition and then alter it to suit your needs. Table D.6 on page 429 lists all possible parameters that can be included in a format definition.

Each new format must have a unique name; that is, the name of the new format must be different from all other Sprint commands and variables—either built-in or defined in STANDARD.FMT. The new name can have up to 24 characters and contain any sequence of letters, digits, and underscores; upper- and lowercase letters are treated equally. The name can also contain any single ASCII symbolic character, such as % or # or +. Here are some valid command names:

NewItemize
Form23B
UserEntry
*
Report
Newsletter
!
2Column
Acct_Invoice

The rest of this discussion will make more sense if there's an example to reference. Let's say you're writing a user's guide and want to create a format for information displayed by a software program. You want the software messages and displays to stand out from the rest of the text; they should be separated from surrounding text by a blank line and appear in bold type, printed with a fixed-width font, indented from both the left and right margins, and single-spaced. The first thing to do is look in the STANDARD.FMT file for a format that comes close to the format you want to create. The Quotation and Example format definitions share functions common to the desired format. Quotation indents text from both margins; Example indents text from the left margin and prints text in a fixed-width font; both format definitions specify single spacing, and insert a blank line above and below the text of the format. Let's work from the Quotation definition and call the new format *Computer*.

Note: You don't have to start with a predefined command. It may be that there is no predefined command that comes close to the effect you're trying to achieve. In that case, you can start from scratch by skipping the first two steps outlined next.

1. Open (a *renamed copy* of) the STANDARD.FMT file, search for *Quotation*, and then select the definition:

```
@Define(Quotation, margins +.5 in, above 1, below 1, spacing 1)
```

2. Copy the definition to the Clipboard, and then paste it at the end of your file. When you've done this, you'll have two Quotation definitions. You're going to convert the second one to the new format.
3. Now give the format a name. Move the cursor to the word Quotation in the second Quotation definition and change it to Computer. The definition now looks like this:

   ```
   @Define(Computer, margins +.5 in, above 1, below 1, spacing 1)
   ```

   Note: If you're starting to define a format from scratch, you should just go to the bottom of the .FMT file and enter the @Define command, followed by a name, followed by parameters, as shown.
4. Using the format parameters listed in Table D.6 on page 429, you can add the missing functions: a bold typeface and a fixed-width font. Both of these formatting functions are accomplished with the *Font* parameter. If your printer has a font that prints text in bold fixed-width characters (for example, courier.bold), you're in great shape. All you need to do is add the *font* parameter to the Computer definition, and specify the name of the printer font that prints bold, fixed-width characters. Be sure to separate the font parameter from the others with a comma. For example,

   ```
   @Define(Computer, margins +.5 in, above 1, below 1,
           spacing 1, font CourierBold)
   ```

   If your printer *doesn't* have a bold, fixed-width font, you can't specify both functions with a single *font* parameter. As noted in the Table D.6, you can enter *one* font parameter per format; if you enter more than one, Sprint accepts the first one and ignores the others. If you enter `font bold pica`, for example, the printer prints the text in bold, but in the default font, not a pica font.

   In such cases, you can use the *Font* parameter to specify the desired printer font, and the *OverStruck* parameter when you want text printed in a bold typeface. *OverStruck* double-strikes characters in the format, which makes them darker than the surrounding text. If your printer definition includes a command to offset overstrikes, characters may be slightly wider. Modify the Computer definition to include the *Font* and *Overstruck* parameters:

   ```
   @Define(Computer, margins +.5 in, above 1, below 1, spacing 1,
           font pica, overstruck)
   ```

   The Computer format definition now includes all the desired parameters. Once you save the renamed copy of STANDARD.FMT, you can use Computer like any other format: Choose **S**tyle/**O**ther Format, type `Computer`, and then press *R* to tell the formatter that this applies to a region of text. If you haven't already selected text, Sprint will prompt you whether to insert a BEGIN or END command. If you've preselected the text, Sprint automatically inserts the BEGIN and END commands as soon as you enter the command name `Computer`.

5. To test the Computer format, create a new file called COMPUTER.TST. If you try out new commands in a test file first, testing usually goes faster and isolating problems becomes an easier task.

   a. Enter the following text in the COMPUTER.TST file:

      ```
      Once you type something, the program displays the
      following message on your screen:
      ```

   b. Choose **Style/Other** Format, type `Computer`, and then press *R* for region.

   c. Press *B* to signal that your cursor is at the beginning of the format reigons. Sprint inserts

      ```
      BEGIN COMPUTER
      ```

   d. Type

      ```
      Do you really want to do this? Answer Y or N __
      ```

   e. Now choose **Style/Other** Format again. Type `Computer`, press *R*, and press *E* to signal the end of the region. Sprint inserts

      ```
      END COMPUTER
      ```

   f. Finish the section by typing:

      ```
      Decide whether or not you want to, and then enter your response.
      ```

6. Print the COMPUTER.TST file. Your printed text looks similar to this:

   Once you type something, the program displays the following message on your screen:

   ```
       Do you really want to do this?
       Answer Y or N __
   ```

   Decide whether or not you want to, and then enter your response.

   If Sprint displays an error message instead of printing your file, note the line number on which the error occurred and check your entry against the one shown in Step 4. Correct the error and repeat this step.

   Be careful that your printer supports the font you specified as part of the *Font* parameter. The example in Step 4 specifies *pica*, but your printer might have an *elite* or *courier* font instead. (Or maybe it *only* has fixed-width fonts.) Check your printer manual to find out if it has a fixed-width font, and modify your Computer command so that it includes the correct font name.

7. Once you verify that the new format does what you want it to, you can use it to format text in any Sprint file.

CHAPTER

4

Custom Document Design

As illustrated by this manual, Sprint can format and produce large, complex documents as easily as small documents. This manual was created with the same version of Sprint you have; as you can see, Sprint's formatting capabilities are powerful. This power stems from the Sprint formatting language.

In the previous two chapters, you were introduced to many of Sprint's formatting commands. Using this book as a running example, this chapter will show you how to build your own custom macros from the formatting commands. You will gain further experience with STANDARD.FMT, the text file that contains the definitions of most standard formatting commands and macros. (Some commands are built-in and cannot be altered.) The Sprint formatter uses the STANDARD.FMT file (or whichever .FMT file you specify) when formatting your files. The .FMT file Sprint uses is called a *style sheet*. You can change style sheets by choosing Layout/ Document-Wide/ Style Sheet.

Although you needn't be a programmer to work through this chapter, you should be prepared for more technical language and concepts. We do assume that you have read and understood the material in the "Advanced Formatting" chapter.

Sprint's formatting macros produced the special chapter heading, "running" footer that varies on odd and even pages, special table of contents format, and other custom design formats you see in this book. To get Sprint to produce this (and most other Borland manuals), we created custom commands and put them in a special .FMT file. We then loaded the new .FMT file with the Layout/Document-Wide/Style Sheet command.

The formatting language consists of Sprint commands that allow you to create *other* Sprint formatting commands. All of these commands are listed in alphabetical order, in Appendix C. In this chapter, we will be using and discussing the following commands:

- Style
- Define
- Macro
- Modify
- Eval
- Case
- Value
- String
- Incr
- If
- TocB

In .FMT files, you enter Sprint commands with @-signs, as in all pure ASCII files. That is, instead of entering commands via the Sprint menus, you'll enter an @-sign, followed by the name of the command.

Note: Since we used a PostScript typesetter to produce this manual, we had access to many fonts and other capabilities. If you are using a different printer or typesetter, you may not be able to do everything we discuss in this chapter. However, you should still find it useful to read through the chapter and learn about the concepts of custom format definitions.

There are four major Sprint commands you'll use in this chapter: @Style, @Define, @Macro, and @Modify. @Style is used to describe formatting styles for the entire document, @Define is used to describe a new format to affect blocks of text, @Macro is used to define a formatting macro, and @Modify is used to alter an already-defined format.

As you work through this chapter, you may want to open the STANDARD.FMT file (on your Program Disk) to see examples of what we are discussing.

Overall Document Style

Every Sprint document must have a document-wide *style;* the default settings are those given in the @Style command at the top of STANDARD.FMT. If you don't specify a style (with the @Style command), Sprint will use the default values for the style parameters. These default values and the use of the @Style command are described in Appendixes B and C.

For this manual, we wanted a different style, so we used a different @Style command:

```
@Style(counter SectionNumber,   justify yes,
       LeftMargin 11.5 picas,   Rightmargin 11.5 picas,
       BottomMargin 1.5 inches, TopMargin 9 picas,
       Size 10 points,          Spacing 1.2,
       Spread .6,               fill no,
       widowprevent on,         Font Palatino)
```

We could have specified values for every one of the parameters, but Sprint's default values for the others were correct for our application.

There are three places you can put an @Style command:

1. **In your document.** You can place as many @Style commands as you want in your document, but since the command specifies a *document-wide* style, it makes sense to use only one @Style command at the very *beginning* of your document (before the ruler line). Some of the @Style parameters have a different effect if the @Style command is not placed at the start of the document (before any ruler lines or printed text). To be safe, you should use only one @Style in your document, and you should position it at the start of your document. You should use individual format commands for any other style and format changes.
2. **In STANDARD.FMT.** If most of the documents you create will use the same style, put the @Style command in STANDARD.FMT. That way you'll have to define a style only once.
3. **In a custom .FMT file.** If a group of your documents will have one style and another group another style, you may want to create a custom .FMT file for each of the groups. Then, specify the appropriate style in each .FMT file.

Note: To subsequently use a different .FMT file when you use Sprint, choose Layout/Document-Wide/Style Sheet, then choose the .FMT file you want.

Caution: Before making any changes to STANDARD.FMT, you should make a copy of the file so you can retrieve it in case you make changes you didn't mean to.

Custom Formats

You'll use custom formats to create custom headings, examples, lists, and any other chunks of text that should look different from the normal style of the document (determined by the @Style command). All the formats in

STANDARD.FMT can be customized to your liking, such as @Foot, @Description, and heading formats. Before we explain how to create your own custom formats, let's dissect one of the heading commands in STANDARD.FMT. @Section uses the generic format *SubHeading*. The definition (determined by the @Define command) of the *SubHeading* format is:

```
@Define(Subheading = Large, above 2, below 2, fill no, group)
```

The equals sign in this definition tells the Sprint formatter to interpret *SubHeading* as if it were *Large*, except for the changes specified by the parameters *above, below, fill,* and *group*. (**Note:** A complete list of all possible format parameters can be found in Appendix D.) The meanings of the parameters used in the *SubHeading* format are as follows:

Above tells the formatter to leave space above the format (in this case, two line spaces).

Below tells the formatter to leave space below the format (in this case, two line spaces).

Fill no (or **fill off**) tells the formatter to do no *filling* (joining of lines) in this format. (if filling is requested (with `fill yes` or `fill on`) the formatter will fill up a line with text from the following line.

Group tells the formatter to keep the text in the format together.

The definition for the format *SubHeading* was built on the definition for the format *Large*. *Large* is defined like this:

```
@Define(Large=B, font large dwidth, ifnotfound, size 1.414,
                 font bold, ifnotfound, overstruck,
                 afterexit "@NoHinge")
```

The format *B* is defined this way:

```
@Define(B, font bold, ifnotfound overstruck)
```

Note: *B* is an example of a typeface format. Typeface formats *do not* force the formatter to automatically start the text on a new line. All other formats will automatically start a new line.

Font is used to specify the font for this format (in this case, bold).

Ifnotfound is followed by either *script, size, overstruck, underline, strikeout,* or *invisible* and specifies what font to use if the preceding font is unavailable on the printer. For example, `font bold, ifnotfound, overstruck` tells the formatter to overstrike the text in this format if the printer cannot print in bold. **Note:** It's a good idea to use *ifnotfound* in your format definitions if your formats will be shared by people who use different printers with Sprint.

Afterexit is an example of a *command parameter*. A command parameter is quite different from the other format parameters. It is a command (or group of commands) that is executed at a certain time in relation to the other text in the format. In this example, the @NoHinge command is executed immediately after the *Large* format is left, but immediately before returning to the parent format (the format that encloses the *Large* format).

There are six other command parameters: *Divider, Initialize, AfterEntry, BeforeEach, WithEach,* and *BeforeExit*. These are fully described in Appendix D.

Defining a Custom Format with @Define

All formats are defined in essentially the same way. The standard format definition is:

```
@Define(newname = oldname, parameter = value, parameter = value, ... )
```

The `= oldname` part is optional. If you include it, the format *newname* (a name you make up) will be a copy of the existing format *oldname, except* for the changes to the listed parameters. (See the *SubHeading* example in the previous section.) If you want to create a brand new format, leave off the = `oldname` part (see the *B* example in the previous section).

You can specify as many parameters (elements to be included as part of the format, such as font) as you want. The values for the parameters that you *don't* specify will be inherited from the format's *parent format*. (The parent format is the format enclosing the format you invoke.)

Note: A complete list of all possible parameters for the @Define command can be found in Appendix D.

Example: A Section Heading

As an example, look at the heading for the main sections of this chapter—for example, "Overall Document Style" on page 154. This head (*Ahead*) is 18-point bold and underlined. In our custom .FMT file, we defined the format for this heading as follows:

```
@Define(Ahead, font Palatino.bold,  size 18 point,
               above 2,             below 1,
               fill no,             group,
               beforeExit "@PROM[@UX(@>)]@*" )
```

Compare the heading for this section with this definition. You've seen examples of all the parameters except *Size* and *beforeExit.*

Size specifies the size of the type. If no unit is specified, the value is multiplied times the size of the parent format's type. For example, if the parent's type size is 10 point, *size .9* would produce 9 point type. To specify a point size explicitly, follow the value with *points*: `size 9 point`.

beforeExit is another example of a command parameter (this concept was introduced under the explanation for the *B* format). The string of characters following *beforeExit* will be executed as commands at the end of the format, *just before* returning to standard text. In this example, the commands in the string ask for two other formats: *PROM* and *UX*. Both of these formats were defined earlier in our .FMT file. *PROM* was defined to be 10-point Palatino and *UX* was defined to mean underlined. The @UX(@>) command tells the formatter to underline the wide break. In this case, the wide break is an entire (blank) line, so Sprint will print a line as long as the *linelength* defined in the @Style command. That is, the heading will be *underlined* with a 10-point Palatino line.

***Note:** Formats must be defined before they can be used.*

Example: Program Listing

Let's look at another example of a format definition. The text of the definitions of the formats—such as the one below—are formatted for this manual by the format *Program*. It's in a typewriter (or monospaced), 8-point type. @Program is defined like this:

```
@Define(Program, font mono, size 8 point,
        spacing 1, justify no,
        fill no, spread 1,  notct,
        leadingspaces kept, blanklines kept,
        initialize "@*",    beforeExit "@*")
```

Spacing specifies the line spacing (or leading). If no unit is specified after the value, *line* is assumed. *spacing 1* specifies single-spacing, *spacing 2* specifies double-spacing, etc. To explicitly specify leading, use points (e.g., `spacing 12 points`).

Justify specifies the type of justification. *Left, no,* or *off* tell the formatter to leave the right edge ragged (filling is still done). *Both, yes,* and *on* tell the formatter to justify both edges. *Right* tells the formatter to justify the right edge and leave the left edge ragged. *Centered* tells it to center the text. *Right* and *center* turn off filling. *Justification* means the same as *justify*. *Centered* means the same as *justify center*. *Flushright* means the same as *justify right*. *Flushleft* means the same as *justify left*.

Spread is the same thing as paragraph spacing and means the extra distance between paragraphs. Adding extra space between paragraphs is an alternative to indenting paragraphs, and is used to visually separate paragraphs.

Notct is the same as **tct no**. It tells the formatter to stop all character translation in this format. (Character translation is further explained on page 170.) All @Tct commands will be ignored. (If you specify the parameter **tct yes**, character translation will be turned on. In this example, we turned tct off for program listings because we wanted them to be printed *exactly* as we typed them.

Leadingspaces can be *ignored* or *kept*. If you specify *ignored*, white space at the start of a line is ignored (this is the default value). If you specify *kept*, tabs or spaces at the start of a paragraph are left as is.

Blanklines can be *kept*, *break*, or *hinge*. If you specify *kept*, blank lines are retained (this is the default value). If you specify *break*, multiple blank lines will be ignored—only one blank line will be printed. If you specify *hinge*, the formatter performs an automatic HINGE command at each blank line. You'll probably want to also specify *group* in the format to make this useful.

Initialize is another example of a command parameter. The commands following *initialize* will be executed at the start of the format, before the text is formatted by the command. In this example, we start off each program listing with a blank line (specified by @*).

Example: An Index

As a final example, we'll look at a more complicated format: the index for this book. Before looking at the index (on page 169), try to imagine how it should look, based on the following format definition:

```
@Define{TheIndex,
        index, columns 2, gutter 3 picas, justify no,
        size 9 points, spacing 11 points, spread .8, indent -2,
        initialize
         "@String(sectiontitle"Index")
          @String(chaptertitle=sectiontitle)
          @If(page&1,y "@blankpage")@NewPage
          @TocB(@*@PBold<Index@$>@Word[@>@eval(Page)]@nohinge@*)
          @HeadingMajor(Index)@NewPage
          @Tabset(2,4,6,8)",
        BeforeEach
         "@case[counter,0 "", else
         "@*@Mediumb[@*@char(counter)]@nohinge@*"]"
        }
```

Some of the parameters are new:

Index is a special parameter. Like the *After* parameter, *index* format text is saved until the end of document is reached, and then processed by the formatter. The index is discussed further on page 169.

Columns tells the formatter to divide the page into columns. You can divide a page into up to 6 columns.

Gutter tells the formatter how much space to leave between the columns, in this case, 3 picas.

Indent tells the formatter where the indent margin is, relative to the left margin (this is different from the indent field of a ruler, where the indent is absolute). If you don't specify a unit for the value, *character* will be assumed.

If the value for *indent* is positive, you're specifying an amount to indent the first line of a paragraph. If the value for *indent* is negative, you will get an "outdented" format (a hanging indent). In this case, the wrap margin is set to the given amount. The *wrap margin* is the margin at which all lines of a paragraph, *except the first line,* will start (the first line of a paragraph will be printed flush against the left margin). If a paragraph is indented in the input file (or starts with a tab), the first line will also start at the wrap margin. (You can also set and move the wrap margin by using the `@$` command.)

Where You Should Keep Your Definitions

Definitions for all formats must be located either in the .FMT file you plan to use for your document, or at the top of any file you print. If you might use the formats in another document, go ahead and put the definitions in STANDARD.FMT (or your standard .FMT file). **Caution:** As mentioned earlier, you should make a copy of the original STANDARD.FMT in case you make a lot of mistakes and want to retrieve the original file.

You can also put format definitions in your document file, but you should define only formats that are unique to that document. For clarity, enter any format definitions at the beginning of your document file.

Before studying this section further, acquaint yourself with the STANDARD.FMT file. It contains many format definitions, and might define all the formats you'll ever need.

Changing a Format with @Modify

You can make changes to any existing format (generic or custom) with the @Modify command. Its structure is similar to @Define:

```
@Modify(format, parameter = value, ...)
```

All parameters that you don't specify will be left unchanged. For example, if you decided that in one special report you wanted all of the @Quotations to be italicized, you would enter the following command in your document:

```
@Modify(Quotation, font = italic)
```

If you're going to modify a format, you must use @Modify *before* using the format. You can't @Modify a format you've already used. If you need to use a format that you want to modify later, you should instead use @Define to describe a *new* format.

Example—Modifying @Numbered

The generic format @Numbered is used to create lists, and uses hyphens to mark the beginning of each list item. Nested paragraphs are marked with asterisks. For this manual, we modified @Numbered so that it would use boxes and bullets (characters available on PostScript devices) to mark the paragraphs, and we wanted extra space between the items. This is how our @Modify statement looks:

```
@Modify(Numbered, numbered "%<@Ding[n]%;@Ding[z]%]",
                  above 6 points,
                  below 6 points,
                  indent -2 )
```

Numbered (the second one; the first one is the format name you're modifying) means that the paragraphs in this format will be numbered automatically by the formatter. The string following the *numbered* is called the *template.* It determines how the *number* of each paragraph will be printed. The template in this example is a *parent template.*

The parent template prints different text for each *parent* of the variable. We use a parent template for our itemized lists because we often have lists imbedded in other lists (we call them sublists).

Parent templates start with `%<` and end with `%]`. In between is the different text to print for each level of parent. The part of the template in the middle specifies what character to print for the main list (`@ding[n]`). A semicolon separates it from what character to print for the sublist (`@ding[z]`). The characters are separated by a semicolon. (`@DING` is simply a typeface format that says we want the PostScript Dingbats font.)

For example, if you have a dot matrix printer that can print all of the ASCII characters and want to print solid boxes for the top level of lists, asterisks for the second level, and hyphens for a third level, you would use this template:

```
numbered "%< ;*;-%]"
```

There are a number of other types of templates; they are described in the *Reference Manual.*

Command Macros

You've just learned how to define style formats. Now you will learn how to define command macros.

A *command macro* is a collection of formatting commands. They are similar to the command parameters you learned about in the first part of this chapter (if you've forgotten what command parameters are, review the explanation on page 157).

The commands in a command parameter are executed only when entering or leaving the command format. The commands in a command *macro* are executed as soon as the formatter encounters them. Here's an example of a commonly-used macro, the macro to insert an item into the index:

```
@macro(Index() = "@`@TheIndex(e=text,v="@, @eval(page)@,")@'")
```

Notice that no style formatting is done to the text. The text is simply sent, along with the page number, to another macro or format (called *TheIndex,* which is described on page 169). To use the macro in your document, you would type @Index() around the word or phrase you want indexed. For example, in our document file, the paragraph above starts:

```
A command macro@Index(command macros) is a collection of...
```

Naming Command Macros

Command macros must be assigned unique names. The name can be made up of any sequence of letters, digits, and ASCII symbols, up to 24 characters, either upper or lowercase letters can be used. The following are examples of legal macro names:

```
@new     @old_list   @-    @Ahead     @List2
@_       @Table      @{    @Macro2a   @Salutation
```

Note: If you want to use an open delimiter such as {, (, [, or <, you should put it in quotes when you define the macro.

Equate Macros

The simplest use of the @Macro command is to make a copy of (*equate*) an existing format macro or format. For example, if you plan to use the @Flushright format a lot, you might want to make a copy with a shorter name:

```
@Macro(FR = FlushRight)
```

This @Macro command copies the definition of @Flushright into a new format, @FR. Now, you can type the command @FR instead of @Flushright. You can equate as many other commands as you like at the same time by adding equations to a single @Macro command:

```
@Macro(FR = FlushRight, FL = FlushLeft, FC = Center)
```

You can also use @Macro to redefine existing commands. For example, if you typed a document and used @U to emphasize words with underlining, but later had access to a different printer that could print in italics, you could redefine @U to mean @E:

```
@Macro(U = E)
```

That way you wouldn't have to change all the @U's in your document; you would just include the equate macro at the top of that document (*not* in the STANDARD.FMT file, since you don't want to permanently get rid of the ability to underline).

Note: You *cannot* redefine a command after using it.

If you redefine an *existing* command with @Macro, you can use the original definition *inside* the @Macro. For example, if you want the @Include command to display a message on the screen telling you when it's including a file, you could use the following command:

```
@Macro[Include() = "@Message(@* INCLUDING @eval)@Include(@eval)"]
```

Substitution Macros

If you have a long word that you have to type often in your document, you can create a *substitution macro*. For example, if you're tired of typing Scotts Valley, you could create a new command called @SV:

```
@Macro(SV = "Scotts Valley")
```

To invoke the macro, simply type @SV wherever you want Scotts Valley to appear. When the formatter encounters @SV, it will expand the macro into the text between the quotation marks.

```
If you're ever in the @SV area, please stop by Borland. We'll
gladly give you a tour of our facilities and introduce you to your
sales rep. Our address is: 4585 @SV Drive, @SV, CA 95066.
```

The formatter will expand each occurrence of @SV into Scotts Valley:

> *If you're ever in the Scotts Valley area, please stop by Borland. We'll gladly give you a tour of our facilities and introduce you to your sales rep. Our address is: 4585 Scotts Valley Drive, Scotts Valley, CA 95066.*

You can put other commands inside the definition of the command. Using the same example, we could save some more time by creating @SVC:

```
@Macro(SVC = "@SV, CA 95066")
```

If you always want your company name to be printed in boldface but your address in italic, you could create a new command:

```
@Macro(BI = "@B<Borland International>@*@~
             @I<4585 @SV Drive>@*@~
             @I<@SVC>@*")
```

@* tells the formatter to do a hard carriage return, and @~ tells the formatter to ignore all whitespace (tabs, spaces) up to the next printing character. (We could have typed all three lines of the address on one line, but broke it up for clarity. The @~ makes sure that the formatter doesn't insert extra whitespace in front of the second and third lines of the address.) Complete definitions of these commands can be found in the *Reference Guide*.

As you can see, substitution macros can get quite complex. You can include any valid Sprint command or any of your own commands between the quotation marks.

Macros with Arguments

Macros can also take arguments. An *argument* is a string of text that you supply when you "call" (invoke) the macro. When the formatter processes the macro, it inserts the argument into specified parts of the macro definitions. For example, the definition of the macro *UnNumbered* is:

```
@Macro(UnNumbered() = "@newpage@HeadingMajor(@*@* @eval)")
```

The parentheses after *UnNumbered* serve as placeholders for the argument you supply. At print time, the formatter replaces every occurrence of `@eval` in the macro definition with the argument you supplied. If you invoke *UnNumbered* with the argument *Section Title,* its definition is first expanded to: `"@newpage @HeadingMajor(@*@*Section Title)"`, and then processed.

Here's another example: To make a macro @Indx that prints a piece of text and includes that text in the index, you might use the following:

```
@Macro(indx() = "@eval@Index(@eval)")
```

Now, typing `@indx(word)` is the same as typing `word` (except that word will be listed in the index).

To indicate that a macro takes an argument, add a pair of delimiters such as () after the macro name. To access the argument *inside* the macro definition, refer to it with the command @eval or @eval(text).

You can invoke a macro (@Mac below) that takes an argument in three ways:

Call	**Result**
@mac	No argument; *text* is undefined
@mac()	*text* is set to a null string, a string of zero length
@mac(argument)	*text* is set to "argument"

`@eval` is the command that expands a variable. It is normally used with delimiters and a variable name. When it is used inside a macro definition, you can use it without a variable name. Then `@eval` will refer to the argument given the macro, which is called *text* (unless otherwise specified—read about multiple arguments on page 168).

Example

For an example, let's dissect the macro that defines a *SubSection,* a command that creates a numbered subheading.

```
@Macro(SubSection() = "
@Incr(SubSection)
@String(SectionTitle = text)
@String(SubSectionTitle = text)
@Flushleft(@b(@*@value(SubSection)  @eval@*@*@NoHinge))
@TOC(@\@\@eval(SubSection)  @$@eval@word[@>(.)@eval(Page)]@*)")
```

`@incr(SubSection)`
increments (adds 1) to the value associated with the variable *SubSection.* If the variable *SubSection* hasn't been used yet, the formatter will add 1 to 0, and the result of `incr(SubSection)` will be 1.

`@String(SectionTitle = text)`
`@String(SubSectionTitle = text)`
make the two string variables, *SectionTitle* and *SubsectionTitle* equal to the specified argument. Notice that when you equate string variables, you don't need to refer to the argument as `@Eval(text)`, just as `text`.

`@flushleft(@b(@*@value(SubSection)  @eval@*@*@NoHinge))`
describes how the argument should be printed. The `@b` means boldface, `@*` means print a blank line before the heading, `@value(Subsection)` means print the number of the section, the two spaces will print as two spaces, `@Eval` is expanded into the argument given the macro (the title of the subsection) and then printed, the first `@*` means go to the next line, the second means print a blank line, and the `@NoHinge` prevents the section head from being separated from the text following it.

`@TOC(@\@\@Eval(Subsection)  @$@Eval@Word[@>(.)@Eval(Page)]@*)")`
describes how the table of contents entry should look for this section. The interpretation is straightforward: Move to the second tab, print the section number and two spaces, set the wrap margin to here, print the argument, some leader dots, then the page number and a forced return. @WORD means to keep together the items included in delimiters (the leader dots and the page number).

Since the table of contents will be printed later, you have to use @eval(subsection) instead of @value(subsection). @value(subsection) would return the value of *SubSection* at the time the table of contents is printed. This would be wrong: we want the current value of *SubSection.*

Example

Dissection of the formatting macro that produces this book's chapter heads and page footers reveals some interesting techniques. The following are the definitions of the macro and some included formats:

```
@macro(Chapter() = "
       @makeodd()
       @incr(Chapter)
       @string(pho = " ")@string(phe = " ")
       @string(pfe="@value<Page>@>@i{@value<BookTitle>}")
       @string(pfo="@i[Chapter
       @value<Chapter>,@title[Chapter]]@>@value<Page>")
       @string(SectionTitle = text)@string(ChapterTitle =text)
       @ChapterStart(C@>H@>A@>P@>T@>E@>R )
       @AvGa7[@ux{@>}]@*@blankspace(4 points)
       @HeadingMajor(@value(Chapter))@*
       @*@*
       @HeadingMinor(@eval)
       @TOCB(@*@PBold<Chapter @eval(Chapter)  @~
       @$@eval>@Word[@>@eval(Page)]@nohinge@*)")

@Define(AvGa7, font AvantGarde, size 7 point)

@Define(HeadingMajor, font AvantGarde.bold, size 26 point,
       columns 1, FlushRight, group)

@Define(HeadingMinor, font AvantGarde.bold, size 20 point,
                      columns 1, FlushLeft, below 2, group)

@Define(ChapterStart, columns 1, above 1 inch, below 0, size 8 point,
                      font AvantGarde, FlushLeft, group, spacing 0,)
```

Compare these definitions to the first page of this chapter. The formats are simple and easy to understand, so let's go straight to interpreting the macro:

```
@incr(Chapter)
```

Adds 1 to the variable *Chapter*.

```
@string(pfe="@value<page>@>@i{value<BookTitle>}")
```

Sets up the string variable *pfe,* which is the string that's printed on the bottom of each *even* page. The value of the variable *page* will be printed flush against the left margin; @> inserts white space to fill up the center of the line; the remainder of the string prints the title of the book in italic type. Compare it to the footer printed on the bottom of one of the odd-numbered pages in this manual.

```
@string(pfo="@i{value<ChapterTitle>@>@value<page>")
```

This is similar to the assignment to *pfe* above, except that it refers to the footer on each odd page. Compare it to the footer on one of the even-numbered pages in this manual.

```
@string(SectionTitle = text)
```

This assigns a name to the string *SectionTitle.*

```
@string(ChapterTitle = text)
```

This assigns a name to the string *ChapterTitle.*

```
@makeodd()
```

We want all chapters to start on new, odd pages. The *makeodd* command makes sure they do.

```
@ChapterStart(C@>H@>A@>P@>T@>E@>R)@*
```

This spreads out (letterspaces) the word CHAPTER across the top of the page. The last parameter of the @ChapterStart definition, spacing 0, sets the line spacing to zero, so that no vertical advancement is made at the end of the line (zero leading). (The `@*` moves the formatter back to the start of this line.)

```
@AvGa7[@UX{@>}]@*
```

This prints a rule.

```
@HeadingMajor(@value(Chapter))@*
```

This prints the number of the chapter.

```
@*@*
```

Two blank lines.

```
@HeadingMinor(@eval)
```

This prints the title of the chapter.

```
@TocB(@*@PBold<Chapter @eval(Chapter)@$@eval>@Word[@>@eval(page)]
```

This is similar to the previous example, except the the chapter title is printed in bold type, flush against the left margin.

Multiple-Argument Macros

Macros can also be passed more than one argument. To do this, you give each argument a name, and you include the names in the beginning of the macro definition. For example:

```
@macro[job(dept,after,rep) =
  "Thank you for you interest you've shown in working for
   Zendex Corporation. We have forwarded your resume to the
    department. If you don't hear from us before
   , please call ."
```

defines a macro to make form letters easier. When you use a multiple-argument macro you must supply a definition for each of the arguments you listed in the macro definition. This is how you would use the example above:

```
Dear Mr. Jones,

@job(dept="Advertising",after="October 1st",rep="Ms. Rannice")

Sincerely,


Mark Thomas
```

You can supply the arguments in any order. If you don't supply one of the arguments, that argument is undefined and won't print anything. You can use the @IfDef command to check whether an argument is defined.

The Index and the Table of Contents

The Index

The index is formatted in a specialized *After* format. (If you don't remember *After*, read about it in Appendix D.) You cannot insert the index format with the @Place command, and you call the format using a special command form.

To put a word into our format *TheIndex* (which later became the index), we used this command:

```
@macro(Index()=(@`@TheIndex(e=text, v="@, @eval(page) @,")@'")
```

The *entries* (words) are alphabetized and the *values* for each entry (page number) are placed into the special index pool, where they are appended together to make a string like

```
entryvaluevaluevaluevalue . . .
```

for each unique entry. The formatter ignores case and formatting commands when comparing entries.

When the formatter reaches the end of the document, it formats the index pool. The entries are printed in the Index format (see page 159).

In the index format, each time the initial letter changes, the variable *Counter* is set to that letter (A is 65) and the BeginEach command is executed.

Commas in the *value* strings are used to make multi-level indexes. The text before the comma locates the primary entry, and the rest of the text describes an entry in a sub-index which is printed after that entry. The subindex formats just like the main index, except each line is printed with a tab command (@\) in front of it. Subindexes may be nested any number of times.

Commas inside commands in the entry string are not used for this, so you can use @word(text,text) to put commas in an index entry.

Although it is useful to understand how the index is created, you need not learn how the process works: just use the @Index formatting macro provided in STANDARD.FMT. You can change the appearance of the index by editing or modifying the index format.

The Table of Contents

Once you've figured out how the index works, the table of contents will seem simple. Here is the definition of the format:

```
@Define(TocB, before, justify no, size 10 point, spacing 1.2,initialize "
@String(pfo = "@=@ref(page,template'%i')")@string(pfe = pfo)
@String(pho = "")@string(phe = pho)
@String(sectiontitle "Table of Contents")
@String(chaptertitle=sectiontitle)
@HeadingMinor(Table of Contents)
@TabSet(1 en, 2 en, 3 en, 4 en, 5 en, 6 en)@*@*")
```

The TOC introduces two new features: use of *template* to change the appearance of the page number; and the *before* parameter.

Like the index, the table of contents is created as your document is formatted. The entries are kept in the TOC format in the same order they are inserted (they aren't alphabetized). The format is processed at the end of the document, after the index.

The page numbers will start at 1, and will appear as lowercase roman numerals (see the description of @Template).

Compare the table of contents in this manual with the format definition listed above.

Translating Characters

Character translation means that you want Sprint to automatically translate characters (or words) into something else when it formats your document. The format of the translate command is simple:

```
@tct("translate from" "translate to")
```

Let's say your printer can print the ® symbol. Since you can't display the ® symbol on your screen, you decide to use the combination `(R)` every place you want the registered trademark symbol. At the top of your document, or in your .FMT file, insert this command:

```
@tct("(R)" "@char(y)")
```

where *y* is the character code for ® on you printer.

You can also include formatting commands in the *translate to* string. For example, one of the first lines in STANDARD.FMT is

```
@tct("-" "-@|")
```

This command tells Sprint to translate all hyphens into hyphens followed by ok-to-break-here commands. This will let the formatter break a line after any hyphen.

In producing these manuals, we used a lot of @tct commands. They are all listed in the file POSTSCR.TCT on your printer driver disk. This file is @include'd by our .FMT file at formatting time.

Look through POSTSCR.TCT and study some tricks we used. For example, all pair kerning is done in POSTSCR.TCT. Double quotes are changed into open and close quotes with the following @tct commands:

```
@tct(' "' "@set(inquote 1)@char(0AAH)")
@tct('("' "@set(inquote 1)(@char(0AAH)")
@tct('"' "@set(inquote +1)@if(inquote&1,y "@char(0AAH)",n "@char(0BAH)")")
```

The first (and second) lines automatically translate all space quote (and (") combinations into open quote symbols and set the variable *inquote* to 1. The third line adds 1 to *inquote* then checks whether *inquote* is odd. If it is, the double quote is translated into a open quote character; if *inquote* has an even value, the double quote is changed into a close quote.

Note: Leading spaces in the *translate from* string are retained in the *translate to* string. Only the *first* character in a translate from string can be a space.

Designing Your Own Document

As mentioned previously, the easiest way to design your own document is to change, or add to, the design and common formats, and the formatting macros listed in STANDARD.FMT.

After you have an idea of the design of your document, try to implement it using the predefined formats and macros. Print out a version of your document using those formats and then compare the result to the design you want.

If you need to make a lot of changes, you'll probably want to make a copy of STANDARD.FMT. Give the new .FMT file a name similar to its purpose (e.g., if it's for reports, call it REPORT.FMT).

Then change the format definitions and format macros to match your design. Start with the easiest changes first. After changing the first few formats or macros, you'll feel comfortable with making the big changes. Don't be afraid to be creative. Sprint was designed for power and *flexibility*. The more you learn about Sprint and push it to its limits, the better it will perform for you.

PART

2

Programming Editor Macros

CHAPTER

5

Sprint Editor Macros

A computer program that does *macros* provides you with the ability to construct, record, and play back actions. A *macro* is simply a sequence of instructions that tells the program to perform an action. In the context of Sprint, a macro can be one instruction, several instructions, or many lines of instructions. A collection of macros is placed in a file that has a file extension of .SPM.

When you understand how the macros and the .SPM files work, you can even build a whole new editing system by changing the collection of macros stored in a user interface .SPM file, and then using that file as the user interface file. In fact, we encourage you to change the editor if you don't like the way it works. Welcome to the world of the write-your-own word processor!

For example, perhaps your chapter headings are in all uppercase (capital) letters and you want to change them so that they only start with an uppercase letter. In such a situation, you could use the Search menu to find each occurrence of a chapter. However, once you get there, you'd have to manually move the cursor over to the chapter title, change the case of the word, and then begin the search again. Using the macro facility, you can automate the process so that all chapter headings are changed to uppercase while you sit back and watch the process!

To help you learn how to create such macros, this chapter will

- introduce you to the concept of macros
- explain what is contained in an .SPM file
- tell you how to use the Macros menu

- teach you how to use macros, for which you don't need any programming knowledge or experience (although we assume you know Sprint reasonably well)
- lead you gently into making your own custom macros out of the macros we supply (you don't need any previous programming experience to make custom macros that allow quick access to some functions in the editor)
- build an example of a semi-elaborate macro, so that you can begin to see how to write useful macros
- provide a complete reference to the macro language itself, so that you can continue to expand and improve your macros

Note: If you're already an experienced programmer, you may be tempted to skip the introductory sections and dig right into the description of the programming language. We don't recommend that, however, since the short time it takes to read the introductory sections will save you a lot of time later.

When used effectively, macros increase productivity and reduce the tedium of repetitive typing. Sprint takes the macro concept and expands it as far as possible; in fact, the entire user interface to the editor is the result of a collection of macros.

There are already several macro-making programs on the market today, including Borland's own SuperKey. Such products work independently of the application in which they're being used, typically recording a sequence of keystrokes and often allowing the user to edit the sequence. The macro-makers then allow the user to easily reproduce the sequence with one or more keystrokes. (In fact, Sprint itself has this ability; choose Utilities/ Glossary/Keyboard Record.)

In itself, this "keyboard-stuffing" is a valuable commodity. In fact, many users, once they are familiar with one of those macro-makers, neglect to learn an individual program's macro language. That way, they do not have to learn a new interface, nor learn about the idiosyncrasies of the language. However, such macro-makers, by their independent nature, can't allow access to the inner workings of the program with which they're dealing.

In contrast, Sprint's macros let you dig down into the structure of the macro language itself. We think that Sprint's macros are so powerful that you will not only use them, but soon wonder how you got along without them! Sprint's macros allow you to access the commands that control the editor in much the same way that the STANDARD.FMT file allows you to control the behavior of the formatter.

Before we begin our discussion of how to use Sprint's macros, we'll talk about the files in which Sprint defines macros.

A Note on Typography

Throughout this section, we use **boldface** type to indicate that a word is one of the primitive (built-in) editor macros and variables. This convention is preserved even in program listings, as a learning aid. When you enter these macros, however, you should *not* make them boldface.

What Is an .SPM File?

A file with an .SPM extension is an ASCII text file containing one or more macros written in Sprint's macro language. .SPM files are very important to Sprint; in effect, they tell Sprint how the editor is supposed to respond to keystrokes, in much the same way that .FMT files tell the formatter how to respond to formatting commands. SP.SPM, for example, is the file that causes the Sprint standard interface to act the way it does.

The distinction between .SPM files and .FMT files is an important one in Sprint. In a sense, you are programming the *formatter* when you write or modify .FMT files; but you are programming the *editor* when you write macros or .SPM files. Remember, the macros discussed in this chapter have absolutely no effect on how a document is formatted (the way it prints); they solely concern the way the editor looks, acts, and responds.

You can create, read, modify, and save .SPM files using Sprint itself, as long as you're sure not to include any ruler lines or onscreen formatting. Whenever an .SPM file is loaded and compiled, the file is converted into memory and combined with the existing SP.OVL file (if any).

If you ever want to document your macros, make your own menus, or actually change the functioning of the editor, you'll need to become familiar with how Sprint uses the .SPM files to control the editor. To that end, we start our discussion by explaining the interaction of the Macros menu with the .SPM files.

Using the Macros Menu

The first thing you need to know about the macros is the gateway into them. Like the other functions of the Sprint interface, the macros are implemented as a menu option. In the default Sprint interface, you reach the macros menu by taking the following steps:

1. Press *F10* to display the main Sprint menu.
2. Press *U* or use the arrow keys to display the **Utilities** menu.
3. Press *M* or use the arrow keys to display the **Macros** menu.

You're then presented with three commands:

Load — Loads and compiles a macro file, adding it to the current overlay (.OVL file) that Sprint is using for its editor instructions.

Enter — Allows you to enter the name of a macro (more about macro names later) that can be executed directly, or assigned to a key for later use.

Run — A shortcut that allows you to save, load, and execute the current .SPM file (that is, the one onscreen at the moment).

Each of these commands is discussed more completely in the sections that follow.

Loading a Small .SPM File

It's possible that your interest in macro files is limited to using those constructed by someone else. For example, Borland supplies an .SPM file called MATCH.SPM that is very useful when writing macros. Its function is to show matching delimiters (like braces) around macro commands.

To load the file, take the following steps:

1. Press *F10* to display the Sprint main menu.
2. Choose the **Utilities** menu.
3. Choose the **Macros** menu.
4. Choose the **Load** command. Sprint then displays a list of all .SPM files in the current directory. In this case, you want to load the file named MATCH.
5. Choose MATCH in the list of files and press *Enter.*

   Note: If MATCH is not in the list, change directories to the Sprint directory and try again. If it is still not there, find the appropriate

Borland distribution disk and copy MATCH.SPM into the current directory.

6. Sprint then displays the message

   ```
   Compiling <drivename> <pathname> \MATCH.SPM
   ```

   on the status line as it loads and compiles the macro definition (that is, as it translates the macro from Sprint macro language to computer language).

7. When the compiling process finishes, the menus will disappear. This means that the process was successful and all macros contained in the MATCH.SPM file are ready to execute.

Loading an .SPM file adds any completely new macro definitions in that file, and replaces any existing macros that conflict with the new macros. In other words, if the new .SPM file contains a new key assignment for *Ctrl-S,* that new assignment will replace the old assignment for *Ctrl-S* when you load the new macro file.

In addition, if the new .SPM file defines a macro with the same name as an already existing macro, the new macro replaces the old one. Keys used to execute the original macro will now execute its replacement.

Note that, however, most macro files only replace conflicting macros or add new ones; they leave intact those that do not conflict. Thus, they don't replace any existing keys or shortcuts unless they conflict with macros in the new file. For example, the MATCH.SPM file you just loaded only *adds* a new macro, and leaves your existing assignments intact, with the single exception of *Alt-M,* which it redefines to the *MatchPair* macro.

Warning: An important exception to this "overwrite only conflicting macros" principle occurs if a **#clear** macro is in the new macro file. (*Directives* are instructions to the compiler.) The **#clear** directive clears *all* macros and starts over with only the key assignments and macros in the new file. Generally, **#clear** is only used in user interface files, such as WORDPERF.SPM. (Any macro file that includes the **#clear** directive *must* have a macro called *Main.*)

Executing the Macro

After you load the .SPM file containing the selected macro, you have to execute the macro. To execute the *MatchPair* macro, using MATCH.SPM as a sample file, take the following steps:

1. Open MATCH.SPM, and position the cursor on any delimiter, like a parenthesis or brace ("{").

2. Press *F10* to display the Sprint main menu.
3. Choose the **U**tilities menu.
4. Choose the **M**acros menu.
5. Choose **E**nter. Sprint then displays the following prompt in the information line:

   ```
   Enter macro:
   ```

6. Type `MatchPair` and press *Enter*. Sprint then displays the following prompt:

   ```
   Execute (E) or Assign to a key (A):
   ```

7. If you wanted to assign this macro to a shortcut, you would press *A*; however, in this case, you would probably want to execute the macro only once, so press *E*.
8. The *MatchPair* macro then searches forward or backward for the matching parenthesis or brace; when it finds the match, the macro bounces the cursor back and forth between the matching characters until you press a key.

Actually, since the MATCH.SPM file reassigned *Alt-M* to invoke this macro, you could have pressed that key combination and skipped steps 2 through 8. In other instances, though, you'll need to use this process to enter a macro.

You can use any number of macro files that someone else has constructed; you simply follow the directions just described and substitute the correct name of the macro when Sprint asks for it on the information line. Thus, you can easily load and execute any small macros contained in .SPM files.

Note: Remember, when you load a new macro file, the new file overwrites any conflicting macro definitions. Thus, if your new macro file redefines a menu that already exists, your new menu will be used.

Clearing Out This Session's Macros

When you exit Sprint, any macros you load are automatically saved to the SP.OVL file, and thus will be available the next time you want them. However, you will sometimes load a macro that you want to execute only once, and don't want to save as a part of the editor's macros. You can prevent it from being saved by doing one of two things:

1. Use the **R**eset Shortcuts command on the **C**ustomize/**U**ser Interface menu. This discards any macros you added to the original interface.

2. Write the existing assignments to a different file by using the **Save** command on the **User Interface** menu. (You can reload this file by choosing **User Interface/Load**.)
3. Preventing the loaded macros from being kept in SP.OVL saves some memory space, but means that you will have to reload the macros the next time you want to use them. In general, it's better to let Sprint automatically save those macro files that you will be using often, and discard those that you will only be using rarely.
4. Loading and executing someone else's macros is the simplest way to use them. Don't feel restricted to this use, however, for there is yet another layer of macros to be considered; you can write your own small .SPM files! At this point, we think it fair to warn you: writing small macro files provides so much power with so little effort that you may find it addictive. In the next section, we introduce you to simple ways to do such things as redefine the way almost all of the control and function keys work.

Creating Your Own Small .SPM Files

As discussed in the *User's Guide,* you can assign shortcuts to menus and menu options simply by using the menu system and pressing *Ctrl-Enter.* You can then save *all* your shortcuts by choosing **C**ustomize/**U**ser Interface/**S**ave and reload them by choosing **U**ser Interface/**L**oad. But you may want to break up your shortcuts so that only a few are reloadable, instead of all of them. This is easily done with a custom .SPM file.

You can change the definition of any key, or even any macro, by writing the changes in a small .SPM file, and then reading the file in so you use the new key or macro definitions. This section presents some simple examples of small .SPM files.

Remember, we're not recommending that you create a file to exactly match these examples. (That would be against the spirit of customization.) We are showing you examples of some changes that could be useful. You may want to make the changes that we indicate, load in the small .SPM files to verify that the examples work, and then throw those small files away by using the **R**eset Shortcuts command.

Redefining the Control and Function Keys

A useful way to customize a small .SPM file is to copy the lines that define what each key does from the SP.SPM file into another file. You can then change the assignments for the keys in the new file. For example, say you're an experienced Macintosh user, and thus are used to word processors where Command-X means Cut and Command-C means Copy. Now, you come back to your PC and find the usual problem; your fingers absolutely insist on looking for those functions under those letters. Therefore, you would like to redefine *Ctrl-X* and *Ctrl-C* as those functions.

Now, you could do so by using the menu-shortcut method, but it might be safer to change those assignments permanently in a small .SPM file.

The best way to begin is to open the SP.SPM file and copy the entire key table (that is, the list of keys defined for your user interface) to a small .SPM file. That way, you can change the key assignments and quickly load them in again (since it's a small file), instead of having to wait for the entire SP.SPM file to be loaded and compiled.

To copy the key table out of the .SPM file, take the following steps:

1. Create a new file called MYKEYS.SPM. (Be sure you use the .SPM extension.)
2. Move to the ruler line and delete it by pressing *Ctrl-Y*. If you don't remove the ruler from your macro file, it will *not* run correctly (Sprint tries to interpret the codes in the ruler as macros—with unfortunate results).
3. Press *F10*, choose **File/Insert**, and provide the name of the .SPM file containing the user interface whose keys you wish to modify. (This file will usually be SP.SPM.)

   Note: We strongly recommend that you do make this copy, and don't use the original file for your experimentation. That way, if you make changes to the file, save those changes, and then find that the changes are destructive, you can easily start over with a new copy of the original .SPM file.
4. When the file appears on your screen, use the search command (press *F7* to search for the words *Control and Function Keys*. Skip the first occurrence of the string, and look for the next one. When you find it, the first few lines of the table will look something like this in SP.SPM:

```
;Control and Function Keys
EscapeKey : if RulerEdit (toeol c) else if record KeyRecordEnd else abort
CopyKey : if (RulerEdit && inruler) CopyLastRuler else BlockCopy
^@ : key + 256 keyexec                   ; handle IBM function key 0 prefix
^A : repeat WordBack
^B : ReFormat
^C : ScreenFwd
^D : Right
^E : Up
^F : repeat WordFwd
^G : DelFwd
```

5. The lines from here to the end of the file define the functions of all of the keys on the keyboard. In the lines above, for example, *Ctrl-A* performs the Wordstar-like *Go back one word* function. You'll notice that the function assigned to the key is fairly easy to understand; we've tried to make the macro language read as much like English as possible.
6. Now, select all the lines preceding the start of this table and delete them (you only want the key assignments in this file).
7. As an example of a key redefinition, let's start by modifying *Ctrl-X*. Find the line that starts with ^X (that's with a caret and an X). In the SP.SPM file, it looks like this:

```
^X : Down
```

8. As defined in this user interface, *Ctrl-X* causes the cursor to drop down a line. However, following our example, you've already decided you'd rather use the key to *cut* text, so you'll have to replace the command after the colon.
9. Next, you need to find the equivalent in the macro language for the function that you want. Assume that you already know that *F7* is the functional equivalent of *cut* in the default interface. Look down a few lines to the lines that define the function keys. In SP.SPM, they look like this:

```
F1 : HelpMenu      ;   F1
F2 : GlossLookUp4  ;   F2
F3 : ToggleSelect  ;   F3
F4 : CopyKey       ;   F4
F5 : BlockDelete   ;   F5
F6 : BlockPaste    ;   F6
F7 : FindFwd       ;   F7
F8 : QueryReplace  ;   F8
F9 : GotoLine      ;   F9
F10 : SprintMenu   ;   F10
F11 : SprintMenu   ;   F11
F12 : SprintMenu   ;   F12
```

10. In the above example, you can see that *F5* performs the *BlockDelete* function. As you remember, *F5* performed the *cut* operation—that is, *F5* moves the marked region to the Clipboard—so the term *BlockDelete* is equivalent to the cut operation.
11. Go back up to the line that begins with ^X, delete the existing command (up to the end of the line), and type the word *BlockDelete*. The line should look like this when you're through:

    ```
    ^X : BlockDelete
    ```

12. That's it! If you save MYKEYS.SPM, and then load it as a new macro definition, the new definition of *Ctrl-X* will take effect, and you can use it to cut a marked region of text. At the same time, you could have redefined *Ctrl-C* to mean *copy*. Of course, you could have made menu shortcuts to those functions on-the-fly, but then you would be subject to the usual problem; the definitions would be lumped together with all your other shortcuts. Now, you can add your Macintosh-like shortcuts independently of any others by loading MYKEYS.SPM.

In the above example, we simply substituted an existing key assignment for another existing key assignment. This provides a permanent, documented, easily accessed home for the new assignments.

Note: We could have also added the meaning to any key that is undefined in SP.SPM, such as *Alt-K* (for "Kut"), and thus retained the original key meaning. The key you choose is up to you, although be careful of conflicts with existing keys.

By now, you may be curious as to what an element like *BlockDelete* really is. It is nothing more than another macro! That is, it is a macro contained within the .SPM file controlling your particular user interface. Using such macros, you can do far more than simply switch key assignments. There are two parts to understanding the macros; finding their definition, which we discuss in the next section, and understanding what the macro does.

Understanding exactly what the macro does, of course, means being able to read the macro language. That information is in the section "Programming the Macro Language." However, we think that you can do some more modifications to .SPM files without understanding the language entirely, as long as you understand that you want a certain operation to perform like an existing operation. You do need to know what you're looking for and what you want to do. Since the macro language has been designed to read a lot like English, it's relatively easy to find the operation you want, as you'll see in the next section.

Exploring the Menus in an .SPM File

Any computer program, including Sprint's .SPM file, is nothing more than an orderly progression of steps. The trick is to be able to figure out the correct order for the steps.

As mentioned at the beginning of this chapter, the SP.SPM file defines the control and function keys and the menu structure of Sprint's default user interface. For your first attempts at programming macros, all you need to do is to look at a copy of SP.SPM and examine the action of that file as it travels its way down through the menus to the operations you want to copy. Thus, you need to learn how Sprint *jumps* from one spot to another in the .SPM file.

At this point, open up a copy of the SP.SPM file you're using, and search for the definition of the Sprint menu (press F7 and enter the word *SprintMenu*). The Sprint menu definition looks like this:

```
SprintMenu :
  menu "Sprint" {
     "File"        FilesMenu,
     "Edit"        EditMenu,
     "_",
     "Insert"      InsertMenu,
     "Typestyle"   TypeStyleMenu,
     "Style"       StyleMenu,
     "Layout"      LayoutMenu,
     "_",
     "Print"       PrintMenu,
     "Window"      WindowsMenu,
     "Utilities"   UtilitiesMenu,
     "Customize"   CustomMenu,
     "_",
     "Quit"        ExitEditor
     }
```

The word *SprintMenu* is the name of the macro. In any .SPM file, the macro name appears before a colon, followed by the macro that is performed when the name is given.

Note that all lines that follow the line with the colon should be indented (it doesn't matter how much).

This macro uses the *menu* command. The structure of the menu command can be represented as follows:

```
WhoAmI : menu "Onscreen name" ("first menu item name" Macro1ToDoWhenChosen,
                               "second menu item name" Macro2ToDoWhenChosen,
                               and so on)
```

The structure of this example can be broken down as follows:

- *WhoAmI* is the macro name for the macro that follows.
- *menu* is the built-in Sprint macro that draws a pop-up menu on the screen.
- The words in quotation marks are the words that appear on the screen when the *WhoAmI* macro command is given.
- *Macro1ToDoWhenChosen* and *Macro2ToDoWhenChosen* are the names of macros that are executed when the appropriate menu selection is made.

Note that the menu items are separated by commas, and that the entire list is enclosed between delimiters (in this case, parentheses). We'll explain the importance of those marks in more detail shortly.

Now, back to the Sprint main menu: If the user chooses the **File** option on the Sprint menu, the *FilesMenu* macro will be performed. The next trick is to find the definitions of the macros to which the menus refer. Sprint's search commands come in handy for this, since you can search the file, for example, for *FilesMenu*. You can then inspect that menu, which looks something like this:

```
FilesMenu :
   menu "File" {
      "New"               NewFile,
      "Open"              OpenFile,
      "Close"             CloseFile,
      "Insert"            InsertFile,
      "_",
      "Save"              Save,
      "Write As"          WriteFile,
      "Revert to Saved"   RevertToSaved,
      "_",
      "Translate"         FileTrans,
      "File Manager"      DiskDirectory,
      "Pick from List"    PickFile
      }
```

As you can see, the *FilesMenu* is composed of more macros. For the moment, and until you understand more about the macro language, you should think about simply copying and combining the lines you need. Using this method, you can find the menus or options that you want and simply copy the lines into another .SPM file, as we'll show you in the next section.

For example, if you wanted to add the function to open a file on a custom menu of your own, you could copy the first few lines of the **Files** menu and modify them, perhaps as follows:

```
MyMenu : menu "Custom Menu" ("Open" OpenFile,
                                 ...more menus or commands to end of menu....)
```

In this example, *MyMenu* is the name of the macro, *Custom Menu* is the text that would appear as the title of the menu on the screen, and *Open* will be the first item on the menu.

You could similarly continue to move around in the .SPM file, copying the functions you need and adding them to your new menu. In the next section, we explore such an example of a custom-made menu.

Adding an "Index" Menu

This section constructs an example short menu out of macros already provided in the default Sprint interface.

For the purposes of the example, assume that you want to make an indexing pass through a document. At the same time, you want to fix some of the inevitable typographical errors and typestyle discrepancies that creep into a document as it is being written. Thus, you want to provide yourself (and any of your other users) with a quicker, more convenient way tailored to this specific use. At the same time, you don't want to cripple Sprint and confuse the issue by replacing any of Sprint's normal interface. What you really need to do in such instances is to provide an entirely *new* menu that you and your users can conveniently call up and use.

One of the most important programming concepts is: learn to write your own code by modifying the code of others! This time-honored method of plagiarism is one of the best ways to learn the language. Thus, for a custom menu, you might want to start by copying the first few lines of a menu that contains some of the operations you want for your new menu. For this example, start by copying and modifying some lines from the *SprintMenu* macro in SP.SPM.

To find this macro, search for the word *SprintMenu* until you find the lines that look like this (you actually already found this in the earlier section):

```
;   ---------- Main Menu ----------

SprintMenu :
  menu "Sprint" {
     "File"         FilesMenu,
     "Edit"         EditMenu,
     "_",
     "Insert"       InsertMenu,
     "Typestyle"    TypestyleMenu,
     "Style"        StyleMenu,
     "Layout"       LayoutMenu,
  "_",
  "Print"       PrintMenu,
  "Window"      WindowsMenu,
  "Utilities"   UtilitiesMenu,
  "Customize"   CustomMenu,
  "_",
  "Quit"        ExitEditor
  }
```

Since we have proposed that this custom menu needs to contain the edit and typestyle menus, you can copy the first seven lines from the menu (down to the line that contains *"Typestyle"*) into a new .SPM file. Copy the lines and open a new file called MYINDEX.SPM.

When the new file is opened, take the following steps:

1. Delete the ruler line that automatically appears at the top of the new file (otherwise an error will be generated when the macro file is loaded).
2. Paste in the seven lines you copied from the SP.SPM file.
3. Change the name of the label in the first line from *SprintMenu* to *MyIndexMenu.*
4. Change the name of the menu (in line 2) from *Sprint* to *My Menu.*
5. Delete the lines you don't need (lines 3, 5, and 6).

When you've finished these steps, the menu looks like this:

```
MyIndexMenu :
  menu "My Menu" {
    "Edit"        EditMenu,
    "Typestyle"   TypestyleMenu,
```

This is almost a complete macro in itself; however, you still need to delete a comma and add a closing brace. You must use commas to separate menu items, and pairs of braces or parentheses to enclose all of the menu items. You'll find that punctuation is very important in Sprint's macro language. This example will demonstrate some of the punctuation principles, and an upcoming example will demonstate more of them. For further help, look at the existing .SPM files and study the examples in the *Macro Encyclopedia,* starting on page 225.

After you've modified the punctuation, your macro looks like this:

```
MyIndexMenu :
  menu "My Menu" {
    "Edit"        EditMenu,
    "Typestyle"   TypestyleMenu
    }
```

Note: The brace could be directly after the word *TypestyleMenu,* but placing the ending brace on a separate line helps to show the underlying structure of the macro.

One of the great things about Sprint's menu processing is that the cursor and first-letter selection techniques come for free in all of the menu processing. Thus, the user using the menu called My Menu above could use the arrow keys and press *Enter* or press *E* to display the Edit menu, and you, as the macro writer, don't have to do anything extra!

The last option we'll place on the menu is the **Index/Word** command. Because this function is on the **Index** menu, which in turn is on the **Style** menu, getting to the macro to copy becomes somewhat complicated. Just remember to follow the bouncing ball of the program flow until it comes to rest on the operation you want.

Look in the *SprintMenu* macro in the SP.SPM file, and you'll see that the **Style** menu performs the *StyleMenu* macro. Since we don't want the whole menu, we need to find that *StyleMenu* macro to choose the function we want. Search for the word *StyleMenu*. When you find it, it will look something like this:

```
StyleMenu :
  menu "Style" {
     "Center Line"     CenterLine,
     "Modify"          ModifyEnv,
     "_",
     "Headings"        HeadingsMenu,
     "Lists"           ListsMenu,
     "_",
     "Table"           InsertTable,
     "Figure"          InsertFigure,
     "PostScript"      PostScriptMenu,
     "Index"           IndexMenu,
     "References"      ReferenceMenu,
     "X-Reference"     XRefMenu,
     "_",
     "Other Formats"   MiscCmdEntry
     }
```

Since the index function is on the Index menu, we need to jump to still another macro; the one called *IndexMenu*. That macro looks like this in SP.SPM:

```
IndexMenu :
  menu "Index" {
     "Word" if !select SelectWord '^D' CharFormat,
     "Reference Word" if !select SelectWord
        set Q8 "IXREF" MakeRegionIntoCmd,
"Master Keyword" if !select SelectWord
   set Q8 "IXMASTER" MakeRegionIntoCmd,
"_",
"See"            0 SeeSeeAlso,
"Also See"       1 SeeSeeAlso,
"Index Under"    IndexUnder,
"Page Range"     IndexRange
}
```

Finally, there it is! Now the task is to copy the appropriate lines and paste them into your new file.

For this example, let's assume that you only want the Word option in your Index menu, so you need to copy only these two lines of the menu to your MYINDEX.SPM file.

Since the example proposed that you wanted to make an indexing pass, it's probably best to place this option at the top of the menu, so that it automatically comes up selected. At the same time, for clarity's sake, it's good to change *Word* to *Index This Word*. After you paste the lines in and change the wording, your menu now looks like this:

```
MyIndexMenu :
   menu "My Menu" {
      "Index This Word" if !select SelectWord '^D' CharFormat,
      "Edit"       EditMenu,
      "Typestyle"  TypestyleMenu
      }
```

This is now a complete, valid Sprint macro statement, but there's something drastically wrong; since you changed the name of the menu to *MyIndexMenu*, no other macro or key calls this menu up. That is, no keystroke will make this menu appear.

You could make the menu appear directly by using the Enter option on the Macro menu and entering *MyIndexMenu* (don't type the file name; be sure to type the *macro* name). However, using a technique you've already learned, it is more convenient to assign a key to call up the menu. Let's assign it to *Alt-A*.

Inspect the MYKEYS.SPM file you made earlier, or the key table at the end of any existing .SPM file. Almost at the very end, you'll see the lines which define the Alt keys. In SP.SPM, the appropriate lines look like this:

```
;   Alt-Letter codes

~A : ChangeRuler
~B : Reselect
~C : CustomMenu
~D :
```

```
~E : EditMenu
~F : FilesMenu
~G : MarkerJump
~H :
~I : InsertMenu
~J :
~K :
~L : LayoutMenu
~M : MarkerSet
~N : NewPage
~O :
~P : PrintMenu
~Q : ExitEditor
~R : NewRuler
~S : StyleMenu
~T : TypestyleMenu
~U : UtilitiesMenu
~V :
~W : WindowsMenu
~X : ExitEditor
~Y :
~Z : ++raw
```

To replace the *Alt-A* assignment, you should place the following line in the appropriate alphabetical order:

```
~A : MyIndexMenu
```

That's it! Save MYINDEX.SPM and the revised version of MYKEYS.SPM to disk, and then load them as macro definitions, and you'll have your very own custom-made indexing menu available at the touch of a key.

Note: Given the above example, MYINDEX.SPM must be loaded first, or the macro facility would give you an *undefined macro* error when you tried to load MYKEYS.SPM. You could have added the key assignment to the end of the MYINDEX.SPM file, and thus would have to load only one macro definition instead of two. But this would split up the definitions of the key table, so it's not to our programming taste. However, it may be to yours, and Sprint supports your right to program it that way!

So far, you've learned how to use an .SPM file to make your own quick menu; this technique provides convenient shortcuts to the existing Sprint interface. However, Sprint does not just provide you with ways to quickly access the existing options. It's the only word processor we know that actually lets you change or add to the way its editing commands work! The next sections begin to show off some of Sprint's flexible nature.

Learning to Program Macros

As you begin to understand how the macros work, we hope you'll begin to stretch and bend Sprint so that it fits your needs, combining existing statements to make new functions. This means you'll have to learn more about the macro language. Like any other programming language, the macro language might seem a bit overwhelming until you get used to it. Thus, we present in this section some more complex examples of macros. You can review these examples and begin to experiment on your own, which will naturally lead you into the programming specifications presented in the last section of this chapter. When you succeed in actually adding something new to Sprint, you're well on the road to successful macro programming.

Macro Conventions

Like any programming language, there are certain conventions you have to follow in order for Sprint to understand your commands. Happily, these rules are few and aren't difficult.

The most important one involves indentation. When writing a macro, only the first line (the line with the colon in it) can start at the left margin. All lines that follow it must be indented (it doesn't matter how much). As you will see in the examples, you can use indentation to your advantage by visually grouping blocks of code.

Another important convention is that you cannot use the names of any of the built-in macros as the names of macros you define. In other words, if you're writing a macro designed to repeatedly go through a file looking for two spaces in a row and changing them to a single space, you cannot call this macro *Repeat.* If you did, the name would conflict with the built-in primitive macro that's called **repeat**. Whenever a primitve macro is mentioned in this manual, it's printed in bold letters to alert you to its special status as a "reserved word."

Two other important restrictions in naming macros are that they cannot contain space characters and must begin with a letter, not a number. You can use numbers in the names, but not as the first letters. Also, upper- and lowercase is irrelevant in macro names (although by convention the primitive macros are printed in all lowercase letters). All the following are legal Sprint macro names:

IsEmpty?
Ringadingy
One4theRoad
z999
TOUPPERCASE

The following are *not* legal names:

2UPPERCASE	(starts with a number)
KeyPressed	(is a reserved word)
empty buffer	(has a space character)

Your best guide to macro conventions is SP.SPM. Browsing through that file will show you what can and cannot be done.

Making Macros That Move the Cursor and Manipulate Text

So far, the discussion has been limited to those macros that are relatively static; that is, they either make key assignments or cause menus to pop up. Some of the macros you copied, however, have actually moved the cursor through the text, although we didn't emphasize that function. The ability to program a macro to move the cursor is one of the most important of the macro language; such abilities allow you to automate many of the repetitive tasks you'll be faced with during day-to-day text editing. By now, you should be ready to learn about such macros.

Perhaps the simplest way to move the cursor is to tell Sprint to move one character at a time. For example, assume the cursor in the current file is on the first character of the word *hello*, as shown in the following line (the underline represents the cursor):

```
hello, world
```

Then, you can tell Sprint to move the cursor one character forward by giving the simple macro command **c** (for *character*).

An excellent way to see exactly what a macro does is to use the menus to reach the Enter command, typing the name of the macro you want, and executing the macro by pressing *Enter*. You could also use the reassignment techniques you've learned to provide a shortcut to the Enter command.

We recommend you execute the macro while the cursor is in a test file that doesn't contain valuable information, to play it safe.

By entering the macro directly, you can test the macro under different circumstances and see exactly what it does. In the preceding example, you can place the cursor on the *h* of *hello,* and then interpret the macro **c**. When you give the macro command **c**, the cursor moves forward one character, as shown here:

```
hello, world
```

You can also direct Sprint to move the cursor backward through the file by changing the direction with an **r** macro command (**r** stands for *reverse*). The **r** command tells Sprint to go backward through the file. How far to go backward depends on the command that follows it. Thus, the command

```
Enter macro: r c
```

would result in the cursor in the preceding example moving back one character to the *h*. If you don't give the **r** macro command, Sprint always assumes that you want the cursor to move forward. Under some circumstances, you may have to change the direction from reverse to forward with the **f** ("forward") macro command.

Another basic way you can move the cursor is by giving a **to** command. The simplest kinds of those are direct commands, such as **toeol**. That command directs Sprint to go to the end of the line. If you look at the definition of the *End* key in the key table of SP.SPM, you'll see that the definition is:

```
F14FH : toeol      ; End
```

By the way, the Sprint macro language is not case-sensitive. By convention the primitive, built-in macros are written in all lowercase letters (and printed in bold in this manual to help you spot them). But if **toeol** looks too odd to you, you could write it **ToEol**, which might improve legibility.

This assigns the to-end-of-line function to the *End* key.

This is an example of a very specific **to** command. Sprint also has a more general **to** case which, when used in conjunction with another class of movement commands called **is** commands, begins to give us the power to move freely through a document.

The **is-** command tests whether the character at the cursor *is* either a specified type of thing, or *is not* a specified type of thing. For example, the Sprint macro command **ispara** tests whether the cursor is on a paragraph mark (hard return) or not on a paragraph mark. For you to understand how to ask Sprint for the results of the test, there's a couple of preliminary concepts you should understand.

The way Sprint indicates that the cursor is or is not a specified type of thing is by *returning a value* of either *True* or *False*. What is meant by *returning a value* is that Sprint maintains and keeps track of a single value (called an *argument*) that your macros can check to decide what to do. You might think of the *argument* as a kind of combination in/out basket that can only contain one thing at a time. The Sprint macro commands can look at the contents of the basket and replace the contents with one of the following:

- a value of True or False (0 = False, Nonzero = True)
- a number in the range –32768 to 32767
- nothing at all; that is remove the value and not place anything new into it
- leave the current value as it is

In a way, such action is like the way you usually use the Clipboard; Sprint retains the last thing entered into the argument for your use.

So, back to the **is-** command, which returns a value of True or False. Such values are called *Boolean* values, and, by checking the condition of the value from a particular **is-** command, you can tell such things as whether the cursor is on a word or a space, or at the end of a sentence or paragraph.

By itself, such information isn't very useful. However, when you combine it with **to** movement commands, you can move the cursor to a specific place in the document.

For example, the macros that move by word units (such as move forward a word or delete this word) need to understand just what is meant by a *word*. Many of those macros reference the *WordBack* macro, which looks like this in CORE.SPM:

```
WordBack : r to istoken r past istoken
```

This somewhat involved chain of movement commands performs the function of moving the cursor either to the start of the current word or one word to the left, depending on whether or not the cursor is already at the beginning of a word. Let's break this macro down to see how that movement is accomplished.

First, **r** command tells sprint to move backwards. Next, we have to tell Sprint how far to back the cursor up. The **to istoken** command means that the cursor is to back up until it is on one of those characters that Sprint considers part of a word. Any letter, number, underscore, dollar sign, percent sign, apostrophe, or hyphen is considered to be part of a word.

Thus, **r to istoken** means to Sprint that, if the cursor is not on a character that is part of a word, Sprint should move the cursor backward until the cursor is on a character that is part of a word; if the cursor is already on such a character, don't move it.

The *WordBack* macro then does an **r past istoken** command. Again, the **r** establishes the direction of the command. The **past** command then tells Sprint to test the character underneath the cursor and move until **istoken** is no longer True. In this case, Sprint moves the cursor backward until the cursor is past the word, that is, until the cursor reaches a character that Sprint does not consider part of a word, such as a space, period, or hard return.

After the command finishes, Sprint moves the cursor forward one character. As a result, this part of the macro ensures that the cursor is left on the very first letter of the word, which completes the *move one word back* function.

This type of combination of movement commands is extremely important; for fun, you might search through SP.SPM for combinations such as **to is** or **past is**. Many macros depend on movement commands for the accurate positioning of the cursor.

The complete specification of each movement command is listed in the *Macro Encyclopedia* which begins on page 225, and you will need to read the SP.SPM file and the individual descriptions in the encyclopedia to understand them completely.

However, before you leave this topic, you should also know about the built-in **current** variable. **Current** contains the ASCII value of the character at the cursor position. When used in conjunction with an **if** statement—and an ASCII table such as the one in SideKick or the appendix in the Sprint *Reference Guide*—this provides you with the ability to check or change the value of the current character. That's getting ahead of ourselves, though; let's defer discussion of that concept until the next section, where you will learn how to tell your macros to make decisions.

Constructing Macros That Make Decisions

To speak anthropomorphically, most of the "brains" of any computer program lies in that program's ability to make tests and then take different actions based on the results. Such tests are very much like the tests we make of the weather when we say, "If it is sunny, I'll go to the beach, otherwise I'll stay home," or, "While it's raining, I'll stay home."

Such tests in programming are usually called *conditionals,* because they check whether a certain condition is true or false. The tests are very much like those involved in the movement commands discussed earlier. With the addition of conditionals, however, you can extend the range of such tests to include any variable.

Sprint uses the words **if** or **while** to indicate that a decision is to be made based on whether a test is true or false.

Using "If" Statements

Using the **if** macro to check whether something is true or false, you can direct the program to make a decision.

For example, as discussed earlier, the macro variable **current** allows your program to find out what the character is to the right of the current point (that is, the character "under" the cursor). Using an **if** statement, you can direct your program to look ahead in the file and take action based upon what **if** finds.

One of the more common instances of this checking ahead in the SP.SPM file is the test used to see if the current character is a hard return character, signifying the end of a paragraph. The ASCII code for that character is *10*, or '^J' (look these up in an ASCII table). Look at the following line:

```
if (current != '^J') ('^J' insert)
```

The exclamation point **!** means *not* to Sprint, so the line can be roughly translated as: "If the current character is not a return character, insert a return character; otherwise, don't do anything." Thus, this macro inserts a hard return character in the text if one is not there already, but avoids inserting two in a row. This is the test Sprint uses to make sure that such things as Begin and End formats, when chosen from the menu, end with a return character, and thus are on a line by themselves.

For another example, when you want to index a single word, the SP.SPM macros will either index the word the cursor is on or index the word already selected. Sprint accomplishes this by using an **if** test and the predefined macro variable **select**; that variable is True if the user has selected something, and false if the user hasn't made a selection.

Here are the lines in the SP.SPM file that do the "index a word" function (only the first two lines are actual macro code; the rest is condensed into English for the purposes of the example):

```
IndexMenu :
   menu "Index" {
      "Word" if !select SelectWord
                    '^D' CharFormat,
      ...other options on the index menu...
```

The **if** statement in combination with the **!select** test means the following: "If nothing has been selected already, do the *SelectWord* routine, then set the argument to '^D' and do the *CharFormat* routine."

The **if** test does not always have to be combined with a *not*, of course; frequently, you'll want to simply check whether something is true or false and then take action based on that test. For example, the first two lines of the macro that makes Beginning and End formats look like this in SP.SPM:

```
MakeBegEnd :
   if select SetEnv
   else {...
      }
```

The **select** predefined variable, if True, means that the user has selected a word or words on the screen; thus, the **if select** *SetEnv* part of the statement means: "If the user has selected something, do the *SetEnv* macro; otherwise do the statements enclosed in braces." (We have left out the rest of the else statement.)

The list of **if** examples could be continued forever, but by now you should have some idea of just how powerful such decision-making can be. There is at least one more important example of a type of implied **if** statement, which is discussed in the next section.

Using "While" Statements

One last explanation of a programming concept, and we'll turn you loose! This concept is the one of repeating a function until something is True. In Sprint, you can accomplish this with a **while** or **do...while** statement.

A use of a **while** test is illustrated in the ASCII export option. In order to make a Sprint file into pure ASCII, the soft returns (represented by '^_') have to be replaced with hard returns (represented by '^J'). Thus, the logic of replacing soft returns with hard returns can be stated as: *Replace all '^_' characters with '^J'*. Using a *while* test, this statement is written as follows in the SP.SPM file:

```
while ('^_' csearch) ('^J' -> current)
```

Using a **while** test allows you to check the state of something until that something reaches a predefined condition; in this case, until there are no more soft returns in the file to be exported.

Building a Macro Step-by-Step

Up to this point, you've only seen isolated examples of how some of the major concepts behind Sprint are implemented. In this section, you're going to build a useful macro step-by-step so that you can better understand some of those concepts.

The example uses a macro to implement one of those typical changes that happens to a document when it's being edited. Imagine that you produced a document which used the convention of double quote marks to set off words with special meaning, such as in the following phrase: the "is" commands.

Now, your boss wants you to replace that kind of convention with the one where special words are marked in italic, as in the following phrase: the *is* commands. A quick examination of the problem indicates that you shouldn't do a global, since you undoubtedly don't want to replace all of the quote marks in the file with italic commands.

You could also do two search-and-replaces to accomplish the effect; that is, you could search for a quote mark and replace all appropriate beginning quote marks with *^E* (Sprint's control code that starts italics), and then go back through the file to search for the appropriate end quote marks and replace them with *^N* (Sprint's control code that ends a format). However, you obviously wouldn't want to go through the file twice if you can find a way to go through it once.

Note: By the way, this set of circumstances really occurred during the production of this manual, and the macros presented here were designed to solve this problem.

When you start to analyze a problem for possible macro solutions, you may want to write out the steps necessary to solve the problem. This preliminary step of constructing the problem-solving steps (known as the *algorithm*) in logical English can save you from a lot of mistakes.

For the problem at hand, you need to define what *beginning* and *end* quotes are, since they are usually the same character when entered from the PC keyboard. However, think about the position of the quotes in relationship to words in the file, and you'll see that a beginning quote is followed immediately by a word, while an ending quote is not followed immediately by a word, but more likely by a space.

Also, for the moment, assume that you have placed the cursor on a quote mark to be changed, and now need to simply replace the quote mark with the appropriate character (we'll improve this part of the algorithm later). Thus, the steps might be written:

1. If the quote mark is immediately before a word, replace the quote mark with *^E*.
2. If the quote mark is not immediately before a word, replace the quote mark with *^N*.

This algorithm clearly states the problem, but we need to define our terms a little more carefully (computers have an unnerving tendency to do exactly what we tell them). In this case, *immediately before a word* can be interpreted to mean *the character to the right of the quote must be a character that is part of a word.*

Also, the term *replace* hides two functions; that is, the existing quote mark must be deleted and the new character inserted. Given this revised version of the meaning of our terms, we can now present the steps as:

1. Beginning at the current cursor position, if the character to the right of the quote mark is part of a word, delete the quote mark and insert *^E.* (The *^E* is the character that starts italics.)
2. Beginning at the current cursor position, if the character to the right of the quote mark is not part of a word, replace the quote mark with *^N.* (The *^N* is the character that ends a format.)

You now begin to interpret these steps in Sprint's macro language.

As you may remember from the preceding discussion of the *WordBack* macro, Sprint provides commands to move forward or backward through the file and the **istoken** command that tests whether a character is part of a word. You may also remember the **if** conditionals and the **current** variable that was discussed briefly. When you combine those concepts with the algorithm presented earlier, you wind up with the following Sprint macro statements:

```
QuoteToItal :
   c if istoken (r c '^E' -> current)
   else (r c '^N' -> current)
```

Here's a breakdown of the code:

QuoteToItal:
The name of the command; this is the name you would enter at the `Enter macro:` prompt when you choose **M**acros/**R**un from the menus.

c
Move the cursor one position forward, so that we can check its value.

if istoken
If the character at the cursor position is part of a word, do the statements that follow in parentheses; otherwise, go to the **else** statement.

(r c '^E' -> **current**)
Reverse the direction, move one character in that direction, and replace that character with *^E.* Note that parentheses enclose the commands that are performed only if the character at the cursor position is part of a word.

Sprint uses parentheses and braces to group commands together into one command; you can think of them as joining macro commands into a single command, or as enclosing the commands between them. You'll see a lot more of them as you continue to build this example.

else
If the character at the cursor position is not part of the word, do the statements in parentheses.

(r c '^N' -> current)
Reverse the direction, move one character in that direction, and replace that character with *^N*. Again, note that parentheses are used to enclose the commands that are done only if the character at the cursor position is not part of a word.

This routine will work only if the cursor is already on a quote mark. However, it's more convenient to automatically move the cursor to the next quote mark in the file and then perform the routine. We can do that by using the **csearch** macro command as shown in the following macro:

```
QuoteToItal :
   '"'csearch
   c
   if istoken (r c '^E' -> current)
   else (r c '^N' -> current)
```

Here's a translation of the new line:

'"' csearch
Search for a quote character. Note that the character is enclosed in single quote marks, which tells **csearch** to search for the ASCII equivalent to that character. If you wanted to search for the quote character by decimal value, you could give the command *34* **csearch.**

The rest of the macro functions in the same fashion as it did before; the addition of the **csearch** command simply saves the user from placing the cursor on the quote mark. This helps, but the user still has to invoke the macro each time. It would obviously be better if the macro would continue through the file, changing quote marks appropriately as it goes.

Such a task is perfect for a **while** macro command, which continues to do a task until a specified condition is no longer true. In this case, assuming you want to change all quote marks in a file, you simply tell Sprint to continue to change quote marks until there are no more quote marks. You modify the existng macro as follows:

```
QuoteToItal :
   while ('"'csearch) {
      c
      if istoken (r c '^E' -> current)
      else (r c '^N' -> current)
      }
```

Here's what the added **while** macro means:

while
This command says that, while **csearch** can still find a quote character, continue doing the group of commands enclosed in braces. When **csearch** cannot find any more quote characters, either because there are no more or because the command has reached the end of the file, the commands in braces are skipped, thus ending the macro.

To help you understand the way Sprint uses parentheses and braces to group commands, you can think of the preceding example as taking the following form (with the commands replaced by *pN* labels and dots for emphasis—in this context, the commands can be anything):

```
QuoteToItal :
   while (p1.........) {
      c
      if .......
         (p2.................)
      else
         (p3.................)
      }
```

The commands within the pair of braces are those that will be performed while the commands between the first pair of parentheses (*p1*) set the argument to True (in this case, while **csearch** can find a quote character).

The macro will then do one command (the **c**) and then move on to the **if...else** command. If the **if** test is True, the macro will perform the commands in *p2;* otherwise the macro will perform the commands in *p3.* The **while** condition in *p1* is then evaluated again. If that condition still returns a True argument, the commands in braces are performed again. If the condition returns a False argument, the macro ends.

Note: As yet, you haven't given any commands to be performed if the condition is False (in this example, if a quote character is not found). You'll add that capability in the next example. As it stands, a False value just brings the macro to a shuddering halt.

By now, you should be able to see that the placement of parentheses and braces are very important when you're grouping commands to be executed as a single command. You'll see more of them as you go along in this example, but feel free to experiment with parentheses on your own. You'll often find that, when one of your macros isn't working as it should, there's a misplaced parenthesis or brace.

Remember, you can use the *MatchPair* macro in MATCH.SPM to help you find a missing closing parenthesis or brace.

So far, you've told Sprint what to do if it finds a quote mark in the file, but neglected to tell it what to do if it can't. This isn't particularly serious in this case; this particular macro goes to the end of the file and quits. However, it's better policy to tell the user what's happening if the character isn't found.

To add this capability, you'll add another command so that, if the quote character is not found, Sprint will sound the system speaker and produce a message on the status line. One added line will do that:

```
QuoteToItal :
   while ('"'csearch) {
   c
      if istoken
         (r c '^E' -> current)
      else
         (r c '^N' -> current)
      }
   Bell message "\nFinished; no more quotes"
```

The new line translates as follows:

bell
Sound the system speaker.

message "\nFinished; no more quotes"
Put the indicated message on the status line. The message will remain on the line until the user presses a key. The \n inserts a carriage return in the string, which effectively removes any messages that might be left over from other macros. (You'll see the importance of that later in this example.)

As an automatic quotes-changer, the macro is now complete; when the user enters the macro command *QuotesToItal*, the macro will, from the cursor to the end of the file, automatically change text enclosed in quotes into italics, and will notify the user when there are no more quotes in the file.

But is the macro really done? Could it be improved? Up to now the assumption has been that the user wants to change all pairs of quotes in the file; that is, the commands simply change the quotes to italics without asking. This is useful, since it makes a one-pass process out of what would have been two, but is not much better than a global change, since it will replace all of the quote marks in the file regardless of their function. When you think about it, this is a perfect example of a macro that should ask the user for confirmation before taking action, in much the same way that search-and-replace works in the default Sprint interface.

The logic for the task at hand becomes as follows:

1. After a quote is found, highlight the quote on the screen. (Since you're going to ask the user about replacing some quotes, you have to show which ones you're talking about.)
2. Ask if the quote should be replaced.
3. If the user answers *Yes,* replace the pair of quotes and continue the macro.
4. If the user answers *No,* skip the pair of quotes and continue the macro.

The expanded macro looks like this:

```
QuoteToItal :
   while ('"'csearch) {
      set themark 1 -> select c draw 0->select
      ask "\nReplace pair of quotes?" ? {
         draw                                               ; turn off highlighting
         if istoken {
            r c '^E' -> current
            ('"' csearch) ('^N' -> current)
            }
         }
         : {                                                     ; don't replace...
            draw                                            ; turn off highlighting
            '"' csearch  c                                    ; skip matching quote
            }
      }
   Bell message "\nFinished: no more quotes"
```

Here's a breakdown of the new commands:

set themark
Sets the **mark** to be equal to the current cursor position. Marks are very important when you're moving the cursor through the file and acting upon various places in the text. You can think of marks as *placeholders* or *signposts;* thus, setting a **mark** marks the current position as a place you want to return to when a command or group of commands finishes. In this particular instance, you're telling Sprint that this is the place where text will begin to be highlighted.

1 -> **select**
Turns on onscreen highlighting. The term **select** is one of Sprint's predefined variables; setting it to 1 turns on the highlighting. The highlighting is actually accomplished the next time the screen is drawn.

c
Move the cursor to the other side of the quote. As a result, since highlighting is turned on, the quote character will be highlighted the next time the screen is drawn.

draw
Draw the screen so that the highlighting is displayed.

0 -> **select**
Turns off onscreen highlighting at the current cursor position by setting **select** to zero. Be careful not to think of this as turning off highlighting in general; instead, think of it as saying "We've highlighted the section we're working with; now stop the highlighting at this position, and get rid of the highlighting the next time the screen is drawn."

ask "Replace pair of quotes?"
Puts into the status line the text that appears within the quote marks and waits for a yes/no response from the user. *Y* or *y* means *Yes*; *N* or *n* means *No*.

? {...} : {...}
Instructs Sprint to test the value of the argument and perform one of the commands or groups of commands. If the argument is True, Sprint performs the first command or commands up to the colon; if the argument is False, Sprint skips to the colon and performs the command or groups of commands after the colon.

In this example, if the argument is True, the user has answered *Yes* to the replace question, so Sprint will perform the commands that replace the quote marks. If the argument is False, Sprint skips those commands and moves to the next group of commands, as detailed in the following description.

draw
Redraws the screen and so gets rid of the highlighting.

if istoken {...}
Note that the commands in this statement have changed, as detailed in the next description. There is now no **else** command for the **if**; that's valid in Sprint's macro language. We'll add an **else** in an upcoming example.

('"' **csearch** ('^N' -> **current**)
This searches for the next quote character and replaces it with a ^*N*.

: {....}
The commands after the colon and between the braces are performed only if the argument is False. In this case, that means that the user doesn't want the quotes replaced.

draw
Gets rid of the highlighting.

; turn off highlighting
Sprint treats everything to the right of a semicolon up to the end of a line as a comment; that is, the terms included after the semicolon will not be executed. The comments can thus be used to document and explain what's happening in the macro. You'll find that the more comments you include in your macros, the easier they'll be to understand three months after you've written them.

'"' csearch c
This searches for the next quote character and then moves the cursor past the character.

Added to the commands we already had in the macro, these commands have turned the macro into a routine with a reasonable human interface; the user can go through the file answering the prompt and replacing the appropriate pairs of quotes. There are still some more tricks we can play to improve the interface part of the macro, but first let's make some allowances for user error.

Whenever you write a macro that will be used by other people, you should try and anticipate what kind of mistakes they might make. In this example, it would be very easy for the original file to have only one quote mark where there should be two. This would get in the way of changing pairs of appropriate quote characters. The following macro traps for ending quote characters that don't have a beginning character:

```
QuoteToItal :
   while ('"'csearch) {
      set themark 1 -> select c draw 0->select
      ask "\nReplace pair of quotes?" ? {
         draw
         if istoken {
            r c '^E' -> current
            ('"' csearch) ('^N' -> current)
            }
         else {
            bell
            message "\nUnmatched quotes - ESC to abort, "
            message "any other key to continue"
            getkey
            }
         }
         : {                                                    ; don't replace...
            draw                                           ; turn off highlighting
            '"' csearch  c                                         ; skip matching
            }
    }
    bell message "\nFinished: no more quotes"
```

The translation of the new lines:

else {...}
This reappearance of the **else** part of the **if** command indicates that the commands in braces will be performed only when the character to the right of the quote mark is not part of a word (that is, when **if istoken** is False). Thus, this test ensures that the first quote of a matching pair must be a starting quote.

bell
Sound the system speaker.

message "\nUnmatched quotes – ESC to abort, "
This message clears out the status line (the \n does that) and prints the first part of the error message to the user. Note that there is an extra space after the comma, placed there so that the first word of the next message doesn't get squished together with this message.

message "any other key to continue"
This message finishes the message to the user. Note that there is no *n* to begin this message; in this case, we want the message to start immediately after the last one. (In fact, the text of the message could be placed in one **message** command. However, because a carriage return can't be placed in the message itself, the text would run off the editing screen on the right-hand side and be difficult to edit.

getkey
This command will wait for the user to press a key.

The example is now pretty well finished, at least in the functional sense. It does everything it needs to do. Now, you can turn your efforts to improving the user interface a little more, and to making it into a more general purpose routine.

First, you'll add some messages that will display on the status line while the search for a quote character goes on. Such messages reassure the user that something is indeed happening.

Then, you'll add some variables at the beginning of the routine and use them to hold the characters to be searched for and to be inserted. You'll find it's good practice to make your macros as general-purpose as possible; that way, you'll begin to build a *library* of routines that can be stored and reused for other tasks. Since the macro will be made more general-purpose, you'll also change some of the display messages to be more general.

Finally, you'll also move the group of commands that result in highlighting the quote character into another macro. This improves the readability of the original macro and makes the function of highlighting a character found by **csearch** easily available to other macros.

Here's the macro with these improvements:

```
HiliteChar : set themark 1 -> select c draw 0->select

QuoteToItal :
   '"'  -> int SearchChar                                         ; quote character
   '^E' -> int TypestyleChar                             ; begin italics character
   '^N' -> int CloseChar                               ; close typestyle character
   status "\nBeginning search..."
   while (SearchChar csearch) {
      HiliteChar
      ask "\nReplace pair of quotes?" ? {
         draw
         if istoken {
            r c TypestyleChar -> current
            (SearchChar csearch) (CloseChar -> current)
            }
         else {
            Bell
            message "Unmatched quotes - ESC to abort, "
            message "any other key to continue"
            GetKey
            }
         }
         : {                                                     ; don't replace...
            draw                                            ; turn off highlighting
            SearchChar csearch  c                                 ;   skip matching
            }
         status "\nContinuing search..."
         }
   Bell message "\nFinished"
```

HiliteChar : ...

The commands in this macro are the same as they have been in preceding examples. This example simply moves them out of the original macro and places them in a separate macro. This improves the logical flow of the original macro and makes the *HiliteChar* macro easily available to other macros.

'"' -> **int** SearchChar

Initializes a local variable called *SearchChar* and assigns the quote characters to it. This statement moves the assignment of the character to be searched for to the front of the macro, where it can be easily changed if necessary. Note also the comment that explains the function of the command.

'^E' -> **int** TypestyleChar
Initializes a local variable called *TypestyleChar* and assigns *^E* to it. This statement moves the assignment of the characters to be used as the beginning type style character to the front of the macro, where it can be easily changed if necessary. For example, note that the single change of *^E* to *^B* makes the macro change characters between quotes to boldfaced text.

'^N' -> **int** CloseChar
Initializes a local variable called *CloseChar* and assigns *^N* to it. Like the preceding commands, this command assigns the close typestyle character in a convenient place.

status "\nBeginning search..."
This command shows a status message to the user while the search for the initial quote character is taking place.

while (SearchChar **csearch**)
This is the first use of the *SearchChar* variable. Sprint uses it exactly as if " had been placed directly in the macro. The rest of the variables in this routine are referenced in the same fashion.

HiliteChar
Sprint now calls the *HiliteChar* macro to do its work of highlighting the character found by **csearch**.

status "\nContinuing search..."
This command shows a status message to the user while the search for the next quote character is taking place.

The user interface to the macro could still be improved, since it won't, for example, find two mismatched beginning characters in a row. Try to do that if you like, or you could add prompts to ask the user what characters were to be replaced with what.

This example has introduced you to some of the major concepts of the macro language and has shown you a few examples of how those concepts are used. By now you should get an idea of how to write your own macros, and you'll need to start writing them to continue learning. Armed with the background we've given you, and the reference material in the *Macro Encyclopedia* section on page 219, we predict you'll soon be happily extending and modifying the way Sprint works. To that end, we offer a challenge.

A Challenge: Build Your Own Interface

Like any good programming language, Sprint's macro language lends itself to learning by trial-and-error and experimentation. This chapter has so far presented concepts and examples to help you get started.

During these long explanations of some macro examples, we've postponed discussion of the actual macro language, hoping you'd get a feel for what was happening before you had to look at the specifics. Now is the time for you to review them. Before you go much further along the path to programming Sprint macros, you'll need to know how the language works.

Here's a challenge: After you read about the inner workings of the macro language, use your newfound knowledge to modify, extend, chop, swap, hack, and customize the appropriate .SPM file to suit your own needs. Perhaps you'll entirely rewrite the interface, and wind up with the editor you always wanted!

The Macro Programming Language

This section defines the basic constructs of the macro programming language, and as such is a fairly detailed account of how the macro language works. If you aren't used to programming languages, and haven't yet read the preceding sections of this chapter, we suggest you do so before continuing.

If you are an experienced programmer, or you have gone through the former sections, this section will provide you with background knowledge that you'll find helpful as you program your macros. The concepts should pose no problem, and you'll only have to get used to the syntax and elements of the macro language (the syntax is similar to C but is not case-sensitive, as C is).

Structure of the Language

When you are in the editor, everything you type runs a *macro*. Macros are behind-the-scenes sequences of commands executed from left to right. A macro executes one *word* at a time. As the macro commands are executed, Sprint keeps track of a single *argument*—the number that is passed to each word as it is executed. That word then returns the next value for the argument, which is then passed to the next word, and so on.

Your macros can check the argument to decide what to do. You might think of the *argument* as a kind of combination in/out basket that can only contain one thing at a time. The Sprint macro commands can look at the contents of the basket and replace the contents with one of the following:

Boolean value	A value of True (nonzero) or False (0)
numeric value	A number in the range –32768 to 32767
null value	Nothing at all; that is, remove the value and not place anything new into it
not do anything	That is, leave the current value as it is

Once the current command returns a value, Sprint passes the value to the next command, and then executes that command. In a way, such action is like the way you usually use the Clipboard; Sprint retains the last thing entered into the argument for your use.

Most commands will use the argument, if it exists. For example, sometimes the argument is used as a repeat count, and the command is done that many times (if the argument is zero, the command is still done once). If the argument doesn't exist, these commands will usually act as though the argument is 1, but some commands act differently when there is no argument. Many commands will ignore the argument, in which case it doesn't matter whether it exists or not.

Most commands return a Boolean value indicating success as 1 (or nonzero) or failure as 0 (zero); this value can then be checked by the conditional commands. A movement command fails if it hits either end of the file without fulfilling its conditions for completion.

For instance, **c** (move right one character) will succeed if the point is anywhere other than at the end of the file. The **toeol** command, however, will fail if the point is in the last line of the file and that line does not end with a return character (when any movement command fails, the point is left at the end of file).

For example, the command **search** returns a value of 1 if the item is found, 0 if not; the macro **if search** *"whatever"* **bell** will cause the bell to ring when *whatever* is found.

Some macro commands need to be followed by another macro command. Instead of executing on its own, the first command controls the second's execution; for instance, the **repeat** macro repeats the next macro *N* times, and the **if** macro can skip the following macro if some condition is false.

A macro can move the pointer that indicates the next macro to be executed, to allow for conditionals. For instance, the question-mark (?) command moves the pointer past the next word if the argument is zero, thus skipping the next command.

Note: Several macros can be combined into one word by enclosing the commands in the brace characters, { }.

Many macro commands require more than one word. Often they are followed by optional strings; these are used to provide values such as file names or search strings, and these strings are considered part of that macro command.

Strings can only be given in two ways, either as a quoted constant (like "hello") to be inserted into the text, or as the name of one of the available string registers, called *Q0* thru *QP*. (In addition, some commands, such as **flist**, act as if they return strings if placed in the right spot in a macro.)

If a command takes a string, it will usually do some obvious operation if the string is missing. For instance **open** *"foo"* will always open the file "foo," but just **open** will prompt the user to enter a file name, and then open the specified file. Note, however, that a quoted string by itself is also a macro command (it is inserted into the text).

The macros are written in text files with the extension .SPM; SP.SPM contains the basic Sprint commands. You can read these files to see examples of macros, and can change them as described below. The editor, however, does not refer to a text file each time it executes a command; instead it compiles a .SPM file into a binary file with the extension .OVL. This reduces each command word to just a few bytes. The code can then be executed very efficiently, so the standard editing macros generally run fast enough to keep up with the fastest typists. It is this binary .OVL file that actually stores the commands the editor uses; once it exists, you can remove the .SPM files from the disk.

The Main Loop

Normally, as Sprint sits around waiting for the user to type a character, it first updates the screen (see the **draw** macro). If no key has been pressed when that is done, it then sits around for a few seconds in a loop (see the **swapdelay** variable). If still no key has been pressed, it starts writing any modified in-memory pages to the swap (backup) file, to preserve them in case of a crash. If it gets done with this and still no key has been pressed, it calls DOS directly and tells it to wait for a key to be pressed.

When a key is pressed, the editor takes the single 8-bit code and executes the macro assigned to that code.

Note: An IBM PC (and many other PC-compatibles) sends more than one 8-bit code for the function keys. The macro for the first code must read the next code and correctly branch to the correct function key.

If there is no macro assigned to the code, the code is inserted into the text. After the macro completes, the main loop continues; the first thing it does is draw the screen.

Input to the editor can be redirected with the <*filename*> switch on the command line. The editor detects this (by checking if the *isdev* bit is zero on *stdin*), and checks for end-of-file. When the editor encounters the end of the input file, it will exit back to the operating system.

Operator Precedence

Some macros (such as +) take more than one argument. If so, the remaining arguments are supplied by macro words after the one being executed. The macro interpreter recursively calls itself to execute these "post" arguments and to evaluate things in order of precedence. For instance, the macro *1+2* first executes *1* (returning an argument of 1), then it executes +. The + macro saves the 1 argument, then executes the next word, *2*, which returns an argument of 2. The + macro then checks the command after the *2* to see if it has higher precedence (if it does, it will be executed next). Then + adds the returned argument to the saved one, and returns a new argument of 3.

The precedence of these operators is as follows:

Unary – and ~
*, /, %, & and ^
<< and >>
+, –, and |
<, <=, = or ==, != or <>, >=, and >
Unary !
&&
| | and ->
All others, such as **if, while, attribute**

Note: This precedence is *similar* to that of the C language, but the bitwise logical operators (&, ~, ^, and |) have been moved to a higher precedence level.

Terms of the Macro Language

There are several terms you should know when working with Sprint editor macros.

DOS Devices

Reference is occasionally made to the DOS terms for standard input device, *stdin*, and for standard output device, *stdout*. Normally, the keyboard is the input device and the screen is the output device, but both of these can be changed, or redirected.

A related term is the register called *isdev*, which is the DOS flag indicating the device or file.

Mark

The *mark* is an invisible indicator in a text buffer. It can be set at a particular position using the **set <mark>** command. Like the *point*, a mark rests between two characters. There exists 16 numbered marks (**mark0** through **mark9** and **markA** through **markF**), as well as a gloabl mark (**gmark**). If you are writing your own macros, you can also use a "stack" of marks, whose top mark is called **themark**. Although there are many marks available, there is only one mark usable at any one time per text buffer.

Point

A *point* is the position in the text where editing occurs. The point is always between two characters, before the first character in the file, or after the last character. If you type a letter, it is inserted into the text at the point. Each buffer has its own point.

Note that the point is not the same as the cursor position. On the screen, the cursor is always on the character to the *right* of the point.

printf % Commands

C programmers will recognize *printf* as an indespensible function for printing formatted text. Sprint adopts the *printf* conventions for specifying formats in its macro language.

Formats are specified using the percent sign followed by a letter. For example, the `%d` format specification says that data should be printed as an integer. Here are the Sprint format specifications allowed:

- %d (number is printed as signed decimal integer)
- %u (number is printed as unsigned integer)
- %c (number is converted to character and printed)
- %x (number is printed as integer in hex format)

You can set the field width by placing a number between the % and the letter; for example, a decimal field of width 4 would be `%4d`.

If you need to print a percent sign, enter `%%`.

Format specifications can also include the backslash followed by letter to represent special characters being inserted into the string. This backslash-plus-letter is known (for historical reasons) as an *escape sequence*. For example, an `\n` at the start of the quoted string after the **message** macro inserts a hard return character, which effectively clears the status line for the message that follows. See the entry for "Strings" in this section for a complete list of escape sequences.

Push and Pop

Placing new data in the stack is called *pushing*; retrieving data (usually a mark) is called *popping*.

Region

A *region* is the piece of text that's spanned by a macro or a series of macros in their execution.

Stack

A buffer (used mostly for storing marks) constructed to be last-in, first-out. That is, data is retrieved in the reverse order as it was stored. As a new mark is added to the stack, it takes the topmost position, pushing all marks already in the stack one spot lower.

Strings

Strings of text in Sprint macros are written between quotation marks. There are also 26 predefined string variables, Q0 through Q9 plus QA through QP, which you can use just like quoted strings.

Any characters within the quotation marks are taken literally except for the following special "escape sequences":

- \a bell (^G)
- \b backspace (^H)
- \f form feed (^L)
- \n hard return (^J)
- \r carriage return (^M)
- \t tab (^I)
- \v vertical tab (ruler indicator, ^K)
- \> wide space (spring, ^F)
- \^ caret
- \\ backslash
- \' single quote
- \" double quote
- \NNN octal constant
- \xNN hex constant
- ^X control character (X can be A-Z, @, [, \,] ^, _, or ?)

Classifying Macro Names

The terms of Sprint's macro language can be divided into the following classes:

- Built-in macros
- Built-in macro variables
- Macro directives
- Automatically called macros

Macros in the first three categories appear in **bold** type in this chapter.

In this section, we give four tables that list all of the terms of the Sprint macro language divided into these categories, and listed alphabetically within each category.

Table 5.1: Built-In Macros

abort
action
after themark
again
ask
at mark
atoi
before merk
break
buffind
bufnum
bufswitch
call
case
cd
cdstrip
clear
cleartab
close
copy
csearch
datecheck
debug
del
delay
delete
do
do...while
dokey
draw
else
engine
erase
error
exist
exit
exitmenus
exitmessage
f
f1...f12
false
fchange
fcopy
fdelete
field
files
flags
flist
fmove
fname
forced
found
get
gmark
hardware
if...else
imenu
inbuff
index
infobox
inruler
insert
insert "string"
insertruler
isascii
isclose
isend
isgray
isin
ismarkset
isnl
isopen
ispara
issent
istab
istoken
isvisible
iswhite
jamount
key
keyexec
keyhelp
keypressed
keypushback
length
lines
macro
mark
marknumber
markregion
match
menu
message
mode
move
mread
nexttab
offset
open
ovlread
ovlwrite
pageread
past
pickcolor
pickfile
pickfont
prevmark
printer
put
qmenu
qnumber
qswitch
r
rangeget
rawout
read
readpage
readruler
redraw
refill
regionfwd
repeat
replace
return
runengine
scroll
search
set
set mark
settab
showkeys
sound
sread
status
stopped
subchar
swap mark
swrite
themark
time
to
to mark
toend
toeol
togmark
toruler
tosol
true
undelete
version
wait
while
winswitch
write
writeregion
writeruler

Table 5.2: Built-In Macro Variables

abortkey	**flag6**	**mousecursor**	**scrollborder**
append	**fontcpi**	**overwrite**	**select**
attribute (=tct)	**inagain**	**ovlmodf**	**smodf**
column	**indent**	**peek**	**sounddur**
cpi	**ioport**	**peekseg**	**soundfreq**
curatt	**isibm**	**previous**	**statline**
current	**justify**	**raw**	**swapdelay**
dcolumn	**killswap**	**record**	**tabsize**
direction	**leftedge**	**rightmargin**	**tct (=attribute)**
dline	**leftmargin**	**ruleredit**	**windows**
flag3	**line**	**rulermod**	**wlines**
flag4	**menudelay**	**rwtrans**	**wtop**
flag5	**modf**	**scancode**	**zoom**

Table 5.3: Macro Directives

#clear	**#include**
#define	**int**

Table 5.4: Automatically Called Macros

Bell	InitArg
DoHelp	Main
EditKey	MenuKey
GetKey	Restart
Init	

CHAPTER

6

Macro Encyclopedia

As you're making your custom key assignments, menu shortcuts, or custom .SPM files, you'll eventually want to know about all of the built-in entities in the macro language. The rest of this chapter provides a complete reference to the language and will be your ongoing guide to building your own macros.

The first section in this chapter lists the commands that don't fit into alphabetical order; the second section lists all macro commands and variables that can be alphabetized.

In the list of macro commands and variables in this chapter, the following conventions are used:

command Any single macro command (including sets of macros with parentheses or braces).

commands A string of zero or more commands. You can group any commands so they are treated as a single one (usually for "post" arguments). Parentheses or braces around a group of commands preserve the current argument, and restore it if the grouped expression does not return anything or ends with a dollar sign ($).

region A string of one or more commands that move the point. the area spanned by the macros is considered the "region."

→ result Commands that return a result (don't confuse this with the -> operator). The result is defined in each individual command.

X — A single macro command (or set of commands in parentheses or braces) that returns an argument. For instance, the name of a variable can be used here.

N or # — The value (argument) returned from the previous command. It is never required that the previous macro return a value; often, a default value of 1 is assumed. *N* and # are just used so that a command supplies a reasonable argument.

M — Any numeric value.

variable — The name of a variable, either a built-in one or one defined by a **#define** or **int** command. (This is more restrictive than *X*.)

"string" — An optional quoted string, or the name of a Q register (Q registers store text that can be used by in various ways; see below), or a macro that returns a string. In all cases this string does not *need* to be placed in the macro. If it is missing, the command will either ask the user for the string, or supply a default string.

For instance, the **open** command needs a file name. You can follow it with a string, as **open** "FILE.MSS". Or you can follow it with a Q register, as **open** Q3. Or, if you don't give a string after the **open** macro, the user will be asked for a file name each time the macro is executed.

The Nonalphabetical Constructs

Some of Sprint's commands and variables can't be organized into alphabetical order. Such things as math commands have only variable names in them, and placing them under the symbol for that variable doesn't seem logical. For this reason, you'll find them in this section.

macro name → result of macro
A macro can include the name of any predefined macro. The current argument is passed to the first command in that macro, and the argument that remains after the macro completes is passed to the next word in this macro.

variable → #
The name of any predefined or built-in variable returns the value of that variable (this is a number, even if a *Boolean* was assigned to it earlier).

number → #
You can embed a constant number in the macro; the number, however, must start with a digit. If it contains only digits, it is taken in decimal. To

get hex, octal, or binary numbers, you can end *number* with *H, O,* or *B,* respectively.

'X' → #
A character in quotes returns the ASCII value of that character.

'^X' → #
A control character (a caret followed by a character) within quotes returns the value of that control character.

Q0-P "string"
A Q register. Possible registers are Q0 to Q9 and QA to QP (26 total).

*commands1 **else** commands2 → result of commands1*
else is normally used for **ifs**. If encountered unexpectedly, **else** skips over the next command, and leaves the current argument alone. Don't rely on this effect, since the skipping action might be changed in the future.

commands1, commands2 → result of commands1
A comma (,) separates cases in a menu or case statement. If encountered unexpectedly, it skips to after the parenthesis. This effect is here for **again** processing; don't rely on it, since it might change in the future.

^char → result of the key's macro
The caret (^) by a letter executes the macro bound to that control key (*char* xor 64).

~char → result of the key's macro
A tilde (~) followed by a character executes the macro bound to that "meta" key (that is, *char* with the high-bit set, or *char* + 128).

~^char → result of the key's macro
A tilde (~) followed by the caret (^) and a letter executes the macro bound to that "meta-control" key (*char* xor 64 + 128).

Fnumber → result of the key's macro
F and a number (usually 1-10) executes the macro given to a "hyper" or function key (*number* + 256). See the key table at the end of SP.SPM for the definition of keys.

N <> M → T/F
Returns True if *N* is not equal to *M*.

N != M → T/F
Same as <> for the convenience of C programmers.

$
Returns 0, or selects the default in **case** macros, or assigns the argument to null before a closing parenthesis.

N % *M* → #
Returns *N* modulus *M*.

N & *M* → #
Returns the bitwise AND of *N* and *M*. (Remember, Booleans are treated as 1 or 0.)

Returns the bitwise OR of *N* and *M*.

! *Boolean* → *T/F*
An exclamation point (!) complements the True/False state of the argument that follows it.

Boolean1 && *Boolean2* → *T/F*
Boolean2 is executed only if *Boolean1* is True. Returns True only if both return True.

Boolean1 || *Boolean2* → *T/F*
Boolean2 is executed only if *Boolean1* is False. Returns False only if both return False.

Boolean1 ^^ *Boolean2* → *T/F*
Both Booleans are always executed. Returns True if only one of the two return True.

Boolean ? *commands1* {:*commands2*} → *result of commands1 or commands2*
If *Boolean* is True, *commands1* is executed (if specified). If *Boolean* is False, *commands2* is executed (if specified).

N * *M* → #
Returns *N* multiplied by *M*.

N + *M* → #
Returns the sum of *N* and *M*. If *N* is null, returns *M*.

N – *M* → #
Returns *N* minus *M*. If *N* is null, returns negative *M*.

++*variable* → #
Increments the contents of the variable, and returns the result. (For the built-in variables that are Booleans, this complements the variable and returns 1 or 0.) There is no postfix ++.

--*variable* → #
Decrements the contents of the variable, and returns the result. (Like ++, this complements built-in Boolean variables.)

N -> *variable* → *N* (modulus what can go in variable)
Sets the variable to *N*. If the variable is a built-in Boolean, it is set to 1 if *N* is not zero.

N / M → #
Returns *N* divided by *M,* rounded down to the nearest integer. Returns *N* if *M* is zero.

N \ M → #
Returns *N* divided by *M,* unsigned and rounded down to the nearest integer. Returns *N* if *M* is zero.

N < M → T/F
Returns True if *N* is less than *M* (a signed comparison).

N <= M → T/F
Returns True if *N* is less or equal to *M.*

N << M → #
Shifts *N* left by *M* bits. (This multiplies the number by two, except for numbers larger than 16383.)

N = M → T/F
Returns True if *N* equals *M.*

N = (at end of macro only)
Echoes the number on the status line; equivalent to **message** "%d". This is so you can quickly type macros such as 2 + 2 = and see the answer 4 on the status line.

N = "string"
Echoes and formats the string on the status line. For example, you can enter 2 + 2 = "%x" and see the answer in hex.

N == M → T/F
Same as =.

N > M → T/F
Returns True if *N* is greater than *M* (a signed comparison).

N >= M → T/F
Returns True is *N* is greater or equal to *M.*

N >> M → #
Shifts *N* right by *M* bits (unsigned shift). (This divides a positive number by two.)

N ^ M → #
Returns the bitwise XOR of *N* and *M.*

:
As a macro file is read by an **mread** function, any line that starts with just a colon is executed immediately. The compiled macro is then thrown away, so it does not take any space. This allows you to write long "batch" macro files, such as for the conversion of one word processor format to another,

without modifying the current overlay. Note that the colon is also used as part of the **?** command, mentioned earlier in this list.

Alphabetical Listing of Macros and Variables

The rest of this chapter contains detailed descriptions of every macro and variable that lends itself to being listed in alphabetical order. Words in **bold** indicate a built-in macro, built-in macro variable, or a macro directive.

abort

Syntax

```
abort
```

Function

Exits either to the closest enclosing **stopped** macro or to the closest enclosing **menu** macro. The **abort** macro will exit to the closest enclosing menu only if an **exitmenus** macro command was not given; when **abort** exits to the menu, the menu will be redrawn and the user can select another entry.

If this command is part of a macro assigned to a key, that key acts as an abort key (like *Esc* in the default interface) and cancels out of menus, cancels string input, and breaks infinite loops.

Example

```
GetKey -> x if (x = '^[') abort
```

This example reads a key from the user and places it in *x*. If *x* is the Escape key (^[), **GetKey** aborts the macro.

See Also

abortkey, break, exitmenus

abortkey (Variable)

Syntax

```
abortkey
```

Returns

#

Function

This variable is the code of the key that aborts loops and is saved in the overlay file. In the SP.SPM interface, this is *Esc*. You can set the key to any control character. You can also set it to function key codes (that is, codes greater than 256). Setting **abortkey** to 0 makes all function keys abort loops on a PC because every function key sends a 0 followed by its code.

The key represented by **abortkey** also acts like *Esc* in all prompts and menus.

Example

```
'^U' -> abortkey
```

This example assigns *Ctrl-U* as the key that will abort loops.

See Also

abort

action

Syntax

```
action
```

Returns

T/F

Function

This flag is True if there is an enclosing region-action command (such as **delete** or **copy**) This is useful if you want to set the current mark to somewhere other than where the command started, but don't want to touch the global mark.

Example

```
Up : (if action (tosol set themark) $)
repeat (tosol r c)
if action tosol
else (dcolumn -> dcolumn)
```

This example defines the *Up* macro to move to the previous line in various fashions, depending on whether a region-action command is currently in force, or whether it was executed by pressing *Up arrow.*

See Also

mark, set (mark), to (mark)

after (mark)

Syntax

```
after mark
```

Returns

T/F

Function

Returns True if the point is after (to the right of) the specified mark; otherwise, this returns False.

Example

```
CtrlQDispatch :
   ...
GetKey CharToAlpha case {
   'A' QueryReplace,
   'B' if (after themark && select) swap themark,
   ... }
```

This macro, which is like the Wordstar *Ctrl-Q B* command, checks if the point is to the right of the current mark and, if something is currently highlighted, to swap the point and the mark. (That is, it moves the cursor to the start of the selection if it's not already there.)

See Also

before (mark), mark, set (mark), swap (mark)

again

Syntax

```
again
```

Function

Reexecutes the last macro executed with a **dokey** command; that is, **again** executes the macro saved for "again" processing. The argument to this command, and any prefix macros, are preserved with the saved keystroke, so they are done as well. We recommend that you not combine this macro with other macros.

Example

```
~A : again
```

This example causes *Alt-A* to reexecute the last macro.

See Also

dokey, inagain

append (Variable)

Syntax

```
append
```

Function

This flag controls whether **delete** or **copy** commands will append the text to whatever is already on the Clipboard, or replace the text already in the Clipboard. Normally, **delete** and **copy** commands append material only if the new deletion is adjacent to the last one.

When **append** is set to 1, text will be added to the Clipboard without replacing the text already there. Where the text to be added is placed depends on the direction in effect for the **delete** or **copy** command; if the direction is forward, the text is placed at the beginning of the Clipboard; if the direction is reverse, the text is placed at the end.

The **append** variable is reset to 0 after each **delete** or **copy**.

Example

```
1 -> append DeleteRegion
```

This example causes the highlighted text to be deleted and added to whatever already exists on the Clipboard.

See Also

copy, delete, erase

ask

Syntax **ask** "*string*"

Returns T/F

Function Asks the user a yes/no question on the status line (as in "Exit without saving text?"). The **menukey** macro is used to parse the next keystroke. *Y, Ctrl-M,* and the "accept" key result in True. *N* and the "cancel" key result in False. *Esc* or the key defined as **abortkey** causes an abort. Any other key causes Sprint to beep and repeat the question.

Example

```
message "Name of file to write block to:" set Q0
if (! (exist Q0) || ask "Overwrite existing file?")
```

This example allows the user to type in a file name and then checks to see if the file exists. If it does, the macro asks whether the user wants to overwrite the file.

See Also **abortkey, menukey, message, status**

at (mark)

Syntax **at** *mark*

Returns T/F

Function Returns True if the point at the specified *mark;* otherwise, returns False. For more details on what *mark* can be, see **set (mark)** in this chapter.

Example

```
mousetrack :
   ...
   140H set themark set gmark 1 -> select,          ; left press
   143H if (at themark) 0 -> select set themark 0 -> x break,
                                                      ; release
   144H tosol set themark set gmark 1 -> select toeol,
                                                  ; left double
   15CH tosol r c dcolumn -> dcolumn,               ; movements
   ...
```

See Also **before (mark), mark, set (mark), swap (mark)**

atoi

Syntax **atoi** "*string*"

Returns #

Function Converts the string (usually a Q register) to a number. A leading minus sign will make the number negative, and a trailing *H, O,* or *B* will make the number hex, octal, or binary, respectively. Any other characters are illegal and will cause errors.

Example

```
message "ENTER a number to repeat:"
set Q0
atoi Q0 -> x
message "Repeating "Q0" times."
```

attribute

See **tct**.

before (mark)

Syntax **before** *mark*

Returns T/F

Function Returns True if the point is before the specified *mark*. For more details on what *mark* can be, see **set (mark)** in this chapter.

Example

```
RegionUpper :
   markregion while before themark ToUpper
```

This example changes characters to uppercase until it reaches the current mark.

See Also **after (mark), at (mark), mark**

Bell

Syntax Bell

Function This macro is automatically called whenever the user needs to be alerted. The argument passed to *Bell* indicates the severity of the error; 0 means that the user mistyped a key, 1 means that the error should produce a status line error message. Currently, other numbers are not defined, although you are free to define them for your own purposes.

The easiest way to sound the terminal bell is to send a ^G to stdout with `Bell : rawout "^G"`. If *Bell* is undefined (or can't be called due to an error such as stack overflow), nothing is done.

Example

```
Bell :
   if isibm sounddur sound
   else rawout "^G"
```

This is the *Bell* macro from CORE.SPM. It checks if the current machine fits Sprint's definition of a "true" IBM, and then either sounds the speaker for a specified time or rings the terminal bell.

See Also **message, prompt, sound, sounddur, soundfreq**

break

Syntax `(commands1) break (commands2)`

Returns Result of *commands1*

Function Exits the closest enclosing loop. The macro returns the current argument to the first command after the loop.

Example

```
SetCols :
   do {
      1 Get "Number of columns to format text in" -> x
      if (0 < x && x < 7) break
      else stopped error
         "Only numbers between 1 and 6 are allowed."
      }
```

See Also **abort, abortkey**

buffind

Syntax `buffind "filename"`

Returns T/F

Function

Searches through all the open buffers for a file whose name matches the specified *filename*. A *filename* matches if, after the name is expanded to a complete path name, the name is the same as the **fname** of the buffer, ignoring case and matching / to \. A *filename* also matches if it contains no disk or directory name and matches the name and extension of **fname**. A null *filename* does not match any buffers, including other unnamed buffers.

If the buffer is found, **buffind** returns True, and the editor switches to that buffer. If the buffer is not found, **buffind** returns False, and nothing else happens.

Example

```
if buffind Q0 close
```

This example closes a specified buffer (the name would have been collected in an earlier macro) if that buffer is currently open.

See Also

bufnum

bufnum (Variable)

Syntax

bufnum

Returns

#

Function

This is the current buffer number and is a read-only value from 1 to 24. One buffer is used for each open file. This variable is useful for making macros that visit every buffer and need to detect when they have gone all the way around the ring.

Example

```
ExitEditor :
  bufnum -> x
  do {
    if modf {
      draw while keypressed (key draw)
      message "\nThe file"
      message fname
      if (ask "Has not been saved, save it?") Save
      }
    else if (IsUnnamed && IsOnlyRuler) close
    } while (bufswitch && bufnum != x)
  eraseswap || !files -> killswap
  GlossSave                          ; save glossary if in use
  exit
```

The **bufnum** in this example supplies the number of the current buffer so that Sprint can check if every open file has been saved before the user exits the editor.

See Also **buffind, bufswitch**

bufswitch

Syntax `bufswitch`

Returns T/F

Function Switches the window to display the next or previous file in the buffer ring. If the **direction** flag is forward, **bufswitch** goes to the next file; otherwise, it goes to the previous one. Repeatedly doing this will cycle all the way around the ring, visiting each file in turn.

The **bufswitch** command returns False if there is only one buffer in the ring.

Example

```
ExitEditor :
   bufnum -> x
   do {
      if modf {
         draw while keypressed (key draw)
         if length fname {
            message "\nThe file"
            message fname
            }
         else message "\nThis Unnamed file"
         if (ask "Has not been saved;
             save it (Y,N,ESC)?") Save
         }
      else if (IsUnnamed && IsOnlyRuler) close
      } while (bufswitch && bufnum != x)
   eraseswap || !files -> killswap
   GlossSave                          ; save glossary if in use
   exit
```

The **bufswitch** command in this example causes the macro to switch to the next buffer until all files that have been modified and not saved have been visited.

See Also **bufnum**

c

Syntax c

Returns T/F

Function Moves one character to the right (or to the left, if direction is *reverse*).

Example `Right : repeat c`

See Also **f, r, move**

call

Syntax # **call** "*string*"

Returns T/F

Function Does a DOS exec-call of the specified program. The returned value is that returned by the program on exit (0 usually means it worked). # refers to the bits in the argument that determine how the call is done:

1 Do "Press any key to continue" after the called program exits.

2 Append the program switches "-p=xxx –s=xxx" to pass the current printer and screen to the formatter.

4 Reserved.

8 Run the "restart" macro on reload (only works if bit 16 is on).

16 Overlay the editor with the called program (uses less memory). The editor then "executes itself" with either –w or –r (depending on bits 1 and 8). Note, however, that this call aborts the current macro!

32 Don't do the "reset" from the screen definition (for example, don't clear the screen under the normal IBM setup). The cursor will be placed in the lower left corner.

If the program name does not contain a directory name, the macro searches the path. If the program name does not have an extension, .EXE, .BAT, and .COM are tried. In addition, if the program name is COMMAND, it is replaced with the DOS COMSPEC environment variable (if you really want to run COMMAND.COM, despite the COMSPEC, use COMMAND.COM with the extension).

Multiple strings can follow the **call** macro, in which case, they are concantenated to make the full command line to be used.

Example

```
SystemCommand :
   message "\nDOS command:" set Q5
   mark {
      to Q5 delete past isgray
      }                              ; get rid of leading spaces
   if (0 subchar Q5) (1 call "command /c" Q5)
   else {
      exitmessage "--Type EXIT to return to Sprint--\r\n"
      0 call "command"
      }
```

The first **call** command in this example is the one that calls COMMAND.COM, passing it the command line entered by the user. The second **call** also executes COMMAND.COM but does not pass any command to it; instead, it presents the DOS prompt.

case

Syntax

`N case (N1 commands, N2 commands,...)`

Returns

Result of one of the commands

Function

N selects the case to be executed. Each of *N1*, *N2*, and so on is executed in turn; if they return the same number as *N*, then the command after them is executed, and the rest of the case is ignored. If no other command is executed, and a **$** command is included, the commands after the **$** are taken as a default case and executed.

Example

```
FindCharFwd :
   status " Find -> "
   mark {
      CharFind case {
         1 c set themark,
         2 return,
        $ message "Not found"
        }
    }
```

The **case** command in this example selects one of the functions, depending on the value returned by the **CharFind** macro.

Multiple values can be used for each *N1*, *N2*, and so on. For example,

```
case {
   1, 2, ...
   3 ...
   }
```

See Also **if, do, do...while**

cd (Variable)

Syntax **cd**

Function Contains the operating system's current disk and directory. You can change the current disk and directory with **set cd** *"string"*. Note that the directory will remain set to the specified string when you exit the editor. However, a change in the directory does *not* relocate the files that are already open; they will still be saved to the directories from which they were read.

Example

```
NewDirectory :
   message "\nChange directory to:" set cd
```

This example in CORE.SPM allows the user to change the directory without leaving Sprint.

See Also **cdstrip**

cdstrip

Syntax **cdstrip** "*filename*"

Function Modifies the specified *filename* by removing the directory from the front of it, if it's the same as the current directory. This makes the file name more useful for many purposes. For example, calling the formatter with **cdstrip fname** instead of just **fname** will make the formatter produce shorter error messages without the complete file name. Similarly, **cdstrip** could be useful for fixing file names in messages before reporting them to the user.

Example

```
DiskDirectory :
   set Q0 cdstrip fname 1 -> x
   menu "File Manager" {
      "Duplicate-Copy" CopyFile,
      ...
```

The **cdstrip** command in this example from CORE.SPM strips the path name from the file name so that the File Manager menu can display just the file name.

See Also **cd**

#clear

Syntax **`#clear`**

Function This is a compiler directive that erases all defined macros, key bindings, and global variables, thus clearing the macro memory. This is the only way to make an .OVL file smaller.

If you use **#clear**, the macro will exit immediately unless you also include a *Main* macro. If you do define a *Main* macro that loops to interpret keys, make sure that you have a keystroke that exits the editor.

Example

```
#clear
```

This command is found in CORE.SPM and erases all current macros so that a fresh start can be made.

clear

Syntax **`clear`**

Function Erases everything in the current buffer (unrecoverable). This is much faster but has the same effect as **r toend erase toend**.

Example

```
ReReadFile :
   if (exist fname) {
      line -> x
      dline -> y
      clear
      $ read fname
      x -> line
      y redraw                         ; force back to same line
      0 -> select                      ; make sure select is off
      }
   else (error "File not yet saved.")
```

See Also **erase**

cleartab

Syntax **# cleartab**

Function Removes a tab stop (if there is one) at # in the ruler line. This modifies only the "cached" ruler line copy; see **readruler** and **writeruler.**

Example

```
RulerFromText :          ; use the current line to set a ruler
   mark {
      tosol
      if isnl return
      mark insertruler 0 -> x            ; clear all old tabs
      while ((x nexttab -> x) < rightmargin) (x cleartab)
      ...
```

The **cleartab** command in this example erases any tabs already existing in the ruler line that are before the right margin on the ruler.

See Also **readruler, settab, writeruler**

close

Syntax **close**

Returns T/F

Function Closes the current buffer and leaves the point in the previous buffer. This command returns False if there is no previous buffer (that is, the user is in the last open buffer) and then creates a new, unnamed buffer.

Example

```
if (!modf && IsUnnamed && IsOnlyRuler) close
```

This example checks to see if the file has not been modified, is not named, and only contains a ruler; if all of those conditions are true, the example closes the file.

See Also **open**

column (Variable)

Syntax

```
column
```

Function This read-only variable is the distance in screen characters from the point to the first column of the screen. The start of the line is column 0. Setting this moves the point to be as close to the given column as possible; for example, 20 moves the point to column 20.

Example

```
IsBlankLine : ispara && column = 0
```

This example checks whether the current character is a carriage return, and whether the column number is equal to zero, and returns True if both are true.

copy

Syntax

```
commands copy region
commands copy region Q0-QP
commands copy region mark
```

Returns Result of region

Function Copies a specified region to the Clipboard, to a specified Q register, or to a specified mark.

Description The **copy** command pushes a new local mark, executes the *region* command (as with all such mark commands, the argument is passed to *region*). The area between the mark and the point is copied to either the Clipboard, or the specified Q register, or the specified mark. In any case, the point stays where it is (it does not move back to the mark), the mark is popped, and the argument is set to the value returned (if any) from *region*.

The **copy** *region Q0-QP* macro functions the same as the **copy** *region* macro, except that the copy goes into the specified Q register rather than the Clipboard.

The **copy** *region mark* copies *region* and inserts it at the mark (which can be in this or any other buffer). The mark can't be within or immediately after the region.

The **copy** *region* command uses the current setting of the **append** macro variable to decide whether to replace or add to the contents of the Clipboard or Q register.

Examples

```
CopyRegion :
   if select {
      FixRegionNoMod
      copy togmark
      Unselect
      1 -> AppendNext
      }

SeeSeeAlso :                   ; 0 = See command, 1 = SeeAlso
   -> int seeflag
   if !select SelectWord
   copy to themark Q0                        ; get word(s) to use
   0 -> select            ; turn off select in case user aborts
   ...
```

The first example copies the selected region into the Clipboard. The second example copies the selected word into the Q0 register.

See Also

append, delete, erase, undelete

cpi (Variable)

Syntax

cpi

Function

The cpi entry from the current cached ruler line.

Example

```
SettheFont :
   mark {
      r toruler -> x
      set QD field "font"
      if !x DefaultRuler
      r toruler set field "font" pickfont QD
      }
   if (fontcpi != 0) fontcpi -> cpi
```

This example moves the value of **fontcpi** (if it is greater than zero) into **cpi**. It can be used to adjust the margins as displayed on a ruler.

See Also

fontcpi

csearch

Syntax *N* **csearch**

Returns T/F

Function Moves forward or backward (depending on the current direction) until the point reaches a character with the ASCII code *N*. The search is literal (that is, uppercase or lowercase matters). This is much faster than the **search** macro.

Example

```
while ('^B' csearch) (^S -> current)
```

This example uses **csearch** to go to each occurrence of a ^B (which is the control character to turn bold on) in the file and replace it with ^S (which turns underlining on).

See Also **search**

curatt (Variable)

Syntax **curatt**

Returns #

Function Returns a number indicating the current attribute combination of the location of the point. This number is 0 if the point is not inside any pairs of control characters that define an attribute (like ^B...^N for bold, or ^E...^N for italics). If nonzero, it is the internal "cache number" of the current attribute combination. Larger numbers usually indicate deeper nesting, and two equal numbers mean exactly the same attribute nesting is around the two points. **Warning:** These numbers can change between runs of the editor!

You can also assign **curatt**. If you do, the editor will insert open and close delimiters around the current point, or move the point past a few delimiters to cause the display to not change, but the point to have the given attribute.

Example

```
ReportType :
   curatt -> x
   message "\nCursor is in"
   if !x (message "plain text." return)
```

```
mark {
   do {
      r to isopen
      r c
      curatt -> y
      if (y < x) {
         current case {
            '^B' message "bold",
            ...
            }
         if (y -> x) (message "inside")
         }
      } while x
   message "text."
   }
```

The **curatt** variable in this example is used to indicate the typeface of the current character.

current (Variable)

Syntax `current`

Function This is the ASCII value of the character to the right of the point (that is, at the cursor location). Setting the variable causes the point to move one character to the right.

Example

```
if (current = '^J') c
```

This example checks the current character and, if it is a carriage return, moves the cursor right one character. Hard returns are '^J' or 10, and soft returns are '^_' or 31.

datecheck

Syntax `datecheck "filename"`

Returns #

Function Compares the date of the specified file with the current buffer's date and returns –1 if the specified file is older than the buffer, 0 if they are equal, or 1 if the file is newer than the buffer. The buffer date is set to the current time when the editor is started, and every time a file is written. Reading a file sets the buffer date of that file only if the date of the file is older than the buffer.

If **datecheck fname** > 0 when you visit an already-open file, you should probably read the new version of the file from disk with a **clear read fname.** If **datecheck fname** > 0 when you want to write a file, the version on disk is newer than the open file, and you probably shouldn't write over the disk file without alerting the user.

The **datecheck** macro returns invalid values if the **fname** is a device or if the current buffer is a Q register. The macro returns 0 for any nonexistent file or if either date is before 1987.

Example

```
do {
   if (!modf && (datecheck fname > 0)) {
      clear
      read fname
      r toend
      }
   }
```

This example checks the date of a prespecified file and reads in the file from disk if the disk file is newer than the current buffer.

dcolumn (Variable)

Syntax

```
dcolumn
```

Function

The "display column." This is used to make up and down arrow functions work. Normally this is whatever column the cursor was in when the user pressed the keystroke starting the current macro. Usually, an automatic **column -> dcolumn** is done each time the screen is drawn. If a macro sets **dcolumn** directly, however, the assignment is reversed during the next draw, doing **dcolumn -> column** instead.

The *Up arrow* and *Down arrow* keys do **dcolumn -> dcolumn**. You should be able to see how this makes them move straight up and down, even when going through lines that are shorter than the starting column.

Example

```
Down :
   (if action (tosol set themark) $)
   repeat (toeol c)
   if !action (dcolumn -> dcolumn)
```

See Also **action, column**

#define

Syntax `#define name = value`

Function Declares a global variable called *name* and initializes it to *value.* If *value* is specified, it must be a single integer or character expression.

If *value* is not specified, *name* is initialized to 0.

Example

```
#define MyVar 4
```

This example creates a new global variable called *MyVar* and gives it an initial value of 4.

See Also **int (local), int (global)**

del

Syntax `del`

Returns T/F

Function Deletes one character (which can't be recovered) in the current direction. Returns False if at the end of a file. This is not the same function as the **delete** macro.

Example

```
BackSpace :
   if raw (r del return)
   do {
      if (!r c) return
      if (isopen) mark { c if isclose (del r del) }
      else if !isclose break
      }
   ...
```

See Also **delete**

delay

Syntax

```
N delay
```

Function

Waits *N* milliseconds and doesn't do anything during that time. The interval is accurate to the MS-DOS system clock. Use this macro only if you need an interval smaller than 1/2 second; if you want longer delays, use the **wait** macro command.

Example

```
Sound :                                ; IBM PC specific
   10 delay                     ; sync with the msec clock
   ...
```

See Also

wait

delete

Syntax

```
delete region
delete region Q0-P
delete region mark
```

Returns

Result of region

Function

Copies the specified region to the Clipboard, or to a specified Q register, or to a specified mark, then erases the region. Refer to the **copy** entry for more information on the process.

The **delete** command uses the current setting of the **append** macro variable to decide whether to replace or add to the contents of the Clipboard or Q register.

Example

```
DeleteToChar :
   status "Delete to:"
   mark {
      if (CharFind = 1) delete to themark
      }
```

This example deletes characters up to a character specified by the user.

See Also

append, copy, erase, undelete

direction

Syntax **direction**

Returns T/F

Function Returns True if the current direction is forward, False if reverse. You can use this to make a macro have "direction sensibility" when enclosed in another macro.

Example

```
CharFind :                                      ; case insensitive
                                   ; move to char, maps ^M to NL
          ; returns 1 if found, 2 if illegal key, else False
  key -> x
  if (x < ' ') {
     x case {
        '^M' '^J' -> x,                         ; map return to NL
        ...
        }
     }
  x CharToUpper -> x
  direction ? {                                  ; forward search
     ...
     }
  : {                                           ; backward search
  ...
  }
```

The **direction** command in this example causes the *CharFind* macro to take different action based on whether the direction is currently forward or reverse.

See Also **c, f, r**

dline (Variable)

Syntax **dline**

Function The "display line." This is the number of lines between the cursor and the top of the window. Setting it moves the point to the start of that screen line.

Example

```
0 -> dline
```

This example moves the cursor to the top left corner of the window.

See Also **dcolumn, line, lines, wlines**

do

Syntax

```
do command
```

Function

This repeats *command* indefinitely. The only way to get out is to execute a **break** or to have the user abort the loop by pressing the key assigned as **abortkey**.

Example

```
BegEndInsert :
   status "Press (B) for Begin command,
           (E) for End command, or ESC to cancel:"
   do {
      GetKey CharToAlpha case {
         'B' InsertBegin break,
         'E' InsertEnd break,
         '^[', abortkey abort,
         $ 0 Bell
         }
      }
```

This example loops until the user presses *B, E, Esc,* or the **abortkey** (assigned as *Ctrl-U* in the default Sprint interface).

See Also

abort, abortkey, break

do...while

Syntax

```
do command while Boolean
```

Function

Repeatedly executes *command,* stopping only when *Boolean* returns False. No argument is returned unless a break was executed inside the loop. This is similar to the **while** macro, but *Boolean* is evaluated after *command* is done. Therefore, *command* is always done at least once.

Example

```
Main :
   if !files DefaultRuler          ; make sure files have rulers
   else {
      bufnum -> x
      do {                   ; read any files with newer versions
         if (!modf && (datecheck fname > 0)) {
            clear
            read fname
            r toend
            }
         bufswitch
         } while bufnum != x
```

```
...
```

This example checks that a file has not been modified and whether a newer version exists on disk; if so, it reads in the newer version. Then the example checks whether the buffer number is not equal to a prespecified number and, if the number is not, cycles through the loop again.

See Also **while**

DoHelp

Syntax

```
DoHelp:
```

Function

This macro is automatically called when the user requests help by typing a key that *MenuKey* or *EditKey* evaluates as the 101H help code. Q0 contains a macro name that is either pulled from the most recently called macro or, in the case of a menu, pulled from the macro that will be done if the current item is chosen. The value in Q0 can be used to locate a keyword in the help text.

DoHelp can do any actions desired in order to display and allow the user to move around in the help text. Whatever *DoHelp* does, if it changes the screen, it should restore things to the way they were and execute a **draw** before returning.

Example

```
HelpMenu :
   1 -> Inhelp
   menu "Help on..." {
      "_Press F1 for Template",
      "Subject" 0 -> InHelp message
         "Enter subject:" set Q0 DoHelp,
      "Key" 0 -> InHelp status
         "\nPress key:" GetKey keyhelp DoHelp,
      "Last Command" 0 -> InHelp $ keyhelp DoHelp
      }
   0 -> InHelp
```

This example sets up a possible help menu.

dokey

Syntax

```
# dokey
```

Returns

Result of the macro for the key done

Function

This macro effectively does **draw key keyexec.** The only difference is that the argument passed to the macro called for the key is #. (Doing **key keyexec** would not pass a usable argument to the key's macro because the key would be passed as the argument.)

dokey is usually placed in the *Main* macro loop.

Example

```
do {
   if stopped dokey
   else {
      AppendNext -> append
      0 -> AppendNext
      }
   }
```

This example shows the **dokey** part of the *Main* macros loop.

See Also

Main

draw

Syntax

```
draw
```

Function

Does an incremental redisplay to show the current state of the buffer on the screen. Be sure to do this before you ask the user a question of some type; otherwise, the screen will not contain current information. The redisplay will halt if the user presses a key before **draw** completes, and the screen will be left in a partially updated state.

Example

```
HiLiteFound :
   mark (found 1 -> select
   draw
   0 wait
   Unselect swap themark)
```

The **draw** command in this example causes the screen to be updated after a string has been found.

See Also

redraw

EditKey

Syntax `EditKey :`

Function The editor has a built-in "prompt editor" that accepts some commands. The *EditKey* macro is automatically called and works like *MenuKey* for this prompt editor; it should read a keystroke and return a number indicating what should be done. The return values are as follows:

0	: throw away keystroke
147H	: start of line (Home)
14BH	: left
14DH	: right
14FH	: end of line (End)
152H	: ins key (not used)
153H	: del key
16DH	: word left (Ctrl-Left)
16FH	: word right (Ctrl-Right)
101H	: help (F1)
'^H'	: delete left one character
'^?'	: delete left one character
'^M'	: confirm (CR)
'^['	: cancel (ESC)
'^J'	: insert a hard return (NL) into string
abortkey	: exit/cancel

All other codes less than 100H or greater than 180H are inserted into the string.

Although a generic *EditKey* that returns IBM function codes would work, you probably want to create an editing style that matches the control characters and function keys in your editor. For instance, if *Ctrl-F* moves forward a character, you could translate it to 14DH (the *Right arrow*).

For another example, to simulate EMACS string entry, where *Esc* confirms entry, and an *Enter* inserts a NL, translate ESC (^[) into ^M and ^M into ^J. You would also probably want to set **abortkey** to ^G.

To make a code such as ^H or ESC insert themselves, add 180H to the code. To make a quote prefix, make ^Q do *GetKey*, add 180H to the key, and return it.

If *EditKey* is undefined, the editor uses *MenuKey*.

EditKey is also used by the ruler-line editor. If the ruler line editor gets a number from *EditKey* that it can't use (such as the up-arrow), it executes the macro for that key at the start of the ruler line (with the point right before the ^K).

Example

```
EditKey :
   GetKey -> int ktmp case {
      '^A'        16bh,                            ; Ctrl-Left
      '^D'        14dh,                                ; Right
      '^F'        16dh,                           ; Ctrl-Right
      '^G'        153h,                                  ; Del
      '^Q'        GetKey + 180h,                       ; Quote
      '^S'        14bh,                                 ; Left
      1aah        '*',                                ; PrtScr
      1adh        '-',                                 ; Grey-
      1abh        '+',                                 ; Grey+
      101h        101h,                             ; F1: Help
      10Ah        '^M',                          ; F10: Accept
      140h,144h   '^M',             ; mouse left key is Accept
      141h,145h   '^[',            ; mouse right key is Cancel
      14ch        150h,   ; if code comes, make '5' Down Arrow
      19bh        exitmenus '^[',               ; Ctrl/Alt-Esc
      abortkey    exitmenus '^U',
      $           ktmp
      }
```

This is the *EditKey* routine from SP.SPM.

See Also

GetKey, MenuKey

engine (Variable)

Syntax

engine

Function

Contains the name of the current Borland word "engine" that will be run when the **runengine** macro is called. Setting this variable actually loads and initializes the engine (if it isn't already loaded).

The string that identifies the engine can be the name of a Q register. The first word in the string is the name of the engine (an .ENG extension will be added if there is none) that is loaded into memory. The rest of the string consists of arguments to the engine, words that usually identify the names of the dictionary and the files it needs (.LEX extensions added by default), and words starting with dashes that serve as switches to control the engine. For example, the command *speller english user* identifies the files SPELLER.ENG, ENGLISH.LEX, and USER.LEX.

Currently, all engines take one or two dictionary files and ignore all switches.

If you set the string to null, nothing happens when **runengine** is called (and it is not considered an error).

Example

```
LoadSpeller :                   ; force the speller into memory
   if ((0 subchar engine) != 's')
      { status "Loading speller..." }
   set engine QJ
   swap themark)
```

See Also

runengine

erase

Syntax

```
erase region
```

Returns

Result of region

Function

Deletes the area without copying it anywhere (and it can't be retrieved). This is faster than **delete** and doesn't change the Clipboard.

Example

```
EraseRegion :
   if select {
      FixRegion
      erase togmark
      curatt -> DelAtt
      Unselect
      0 -> AppendNext
      }
   else NoBlock
```

The **erase** command in this example causes the entire region up to the global mark to be erased.

See Also

append, clear, copy, del, delete, undelete

error

Syntax

`error "message"`

Function

Prints the string as an error message, waits for the user to press *Esc*, and then does an abort. If you want, you can print the current argument with a single % command in the message.

Example

```
AssignError :
   stopped error "That key cannot be reassigned."
```

See Also

abort

exist

Syntax

`# exist "filename"`

Returns

T/F

Function

Returns True if *filename* exists on disk and is readable (if a file has been created in the editor but not yet written out, this will return False).

The value given in # controls where Sprint looks for the file, as follows:

2 Searches the path for any matching files.

4 Forces the directory menu. A menu is drawn of all matches (usually 1) even if there is no wildcard (* or ?) in the name. In addition, names ending in :, /, or \ have an asterisk (*) wildcard added to them, so they list the contents of a disk or directory.

8 Hides the extensions in the directory listing and returns the file name with the extension removed.

Example

```
CopyFile :
   message "File to copy:" set Q0
   set Q1 Q0
   set Q0 flist Q1
   if !exist Q0 {          ; if spec'd file mask had no matches
      set QD Q1
      mark { to QD "\nNo files match ' "toend" '." }
      error QD
      }
   set Q1 "" message "Copy" message Q0 message "to:" set Q1
   status "\nCopying..."
   fcopy Q0 Q1
   message "\nCopy complete."
```

The **exist** command in this example checks whether a file given as the file to be copied exists on disk.

See Also **flist**

exit

Syntax **exit**

Function Exits the editor back to DOS.

If the **killswap** variable is True, and the swap (backup) file is a temporary one not created by SPRECOVE, the swap file is deleted. If the **ovlmodf** variable is True, the overlay file is written before exiting. Similary, if the **smodf** variable is True, the screen description is written before exiting.

Example

```
ExitEditor :
   bufnum -> x
   do {
      if modf {
         draw while keypressed (key draw)
         message "\nThe file"
         message fname
         if (ask "has not been saved, save it?") Save
         }
      else if (IsUnnamed && IsOnlyRuler) close
      } while (bufswitch && bufnum != x)
   eraseswap || !files -> killswap
   GlossSave                          ; save glossary if in use
   exit
```

This example exits the editor in the default Sprint version. Note that the routine before the **exit** command checks whether a file has been modified and prompts the user about its unsaved condition before allowing the user to exit.

See Also **killswap, ovlmodf, smodf**

exitmenus

Syntax

exitmenus

Function

Prevents menus from continuing. Doing **exitmenus** causes all menus to disappear, even if the macros being done are aborted. Normally, if a macro in a menu is aborted, Sprint redraws the menu and lets the user select another operation. The **exitmenus** state stays on until the last menu disappears from the screen.

Example

```
MenuKey :
   GetKey -> int ktmp case {
      '^C' 151h,                                   ; page down
      ...
      '^U' exitmenus '^[',
      ...
      }
```

The **exitmenus** command on line 5 in this example is done if the user presses *Ctrl-U.*

See Also

abort, abortkey, break

exitmessage

Syntax

exitmessage "*string*"

Function

This macro specifies the string to be sent to standard output the next time the screen is reset by either exiting the editor or by calling another program. The *string* will be sent after the screen is reset. Thus, you can display a message even if resetting the screen will clear that message.

Example

```
SystemCommand :
   message "\nDOS command:" set Q5
   if (0 subchar Q5) (1 call "command /c" Q5)
   else {
      exitmessage "--Type EXIT to return to Sprint--\r\n"
      0 call "command"
      }
```

This example invokes the DOS command in CORE.SPM and produces the message that instructs the user to type `exit` to return to Sprint.

See Also

call, exit

f

Syntax **f** *command*

Returns Result of command

Function Executes the command with the direction forward and is used to override an enclosing **r** command.

Example

```
WindowFwd : f winswitch
```

This example ensures that the window being switched to will be the "next" window instead of the "previous" window.

See Also **c**, **r**

false

Syntax **false**

Returns False

Function Returns False. You can use this to directly set a Boolean variable.

See Also True

fchange

Syntax **fchange** "*filename1*" "*filename2*"

Function Changes the file name specified in *filename1* by replacing any %'s in *filename1* with the appropriate part specified in *filename2*. You can use this command to modify a part of a file name.

Example

```
SetStyleSheet :
   mark {
      set QD ""
      message "Name of style sheet to use: "
      set QD
      set QD fchange "%.FMT" QD
      ...
```

This example makes sure that the style sheet named by the user (and placed in the QD register) is given an extension of .FMT.

See Also

fcopy, fdelete, flist, fmove

fcopy

Syntax

fcopy *"filename1" "filename2"*

Returns

T/F

Function

Copies disk *filename1* to *filename2*. This is independent of the swap file's contents; it does not matter if either source or destination are opened or modified. The file is copied in 32K chunks, the raw ioctl bit is set on the destination, and the editor attempts to duplicate the source's data onto the destination.

Returns True if the copy is successful, aborts if the copy fails for any reason (for example, the source does not exist).

If the destination already exists, the user will be asked if it should be replaced. To prevent this question, the macro should delete the destination first. If the user answers no, an **abort** is done, so further commands in the macro are not done.

Example

```
CopyFile :
   message "File to copy: " set Q0
   set Q1 Q0
   set Q0 flist Q1
   0 AllCaps
   if !exist Q0 {          ; if spec'd file mask had no matches
      set QD Q1
      if length QD {
         mark { to QD "No files match'" toend "'." }
         }
      else set QD "Can't copy unnamed files."
      error QD
      }
   set Q1 "" message "Copy" message Q0
      message " to: " set Q1
   status "\nCopying..."
   fcopy Q0 Q1
   message "\nCopy complete."
```

This example is the default file copy routine from CORE.SPM.

See Also **fchange, fdelete, flist, fmove**

fdelete

Syntax `fdelete "filename"`

Function Deletes the specified file from the disk (but does not close any buffer containing that file).

Example

```
DeleteFile :
   message "File to erase:" set Q0
   set Q1 Q0
   set Q0 flist Q1
   0 AllCaps
   if !exist Q0 {          ; if spec'd file mask had no matches
      set QD Q1
      mark { to QD "No files match'" toend "'." }
      error QD
      }
   message "\nAre you sure you want to erase" message Q0
   if (ask "?") fdelete Q0
```

This example is the *DeleteFile* routine from CORE.SPM.

See Also **fchange, fcopy, fmove**

field

Syntax `field "fieldname"`

Function Edits ruler lines and any command that starts with ^O and ends with ^N. The macro simplifies reading and writing fields in formatter commands.

To use the *field* macro, position the point on the ^K in a ruler line or on the ^O at the start of any other command. The editor splits the command into fields with commas. The *fieldname* is the first word or symbol after the comma, and the value of the field is everything after the first word or symbol up to the next comma, hard return, or ^N, with any leading spaces stripped. Notice that, if you have *^OBegin text...*, the editor classifies the elements as a field called *Begin* with a value of *text*.

If **field** is placed anywhere a string is expected, it will return the contents of that field. For example,

```
set Q0 field "linelength"
```

puts the contents of the *linelength* field into Q0. If the field does not exist, Q0 is set to empty. (A field never has a null value.)

You can set a field using the **set** macro command. For example,

```
set field "linelength" Q0
```

puts Q0 into the *linelength* field. If the field does not exist, it is added to the end of the command. If you set a field and don't give a source string, the user is asked for one.

Whenever a field command is used, the point is moved to the end of the field value or to the end of the command if the field does not exist. You will usually want to embed fields in **mark** commands so that the point does not move.

Example

```
SetLeftIndent :
   mark {
      r toruler -> x
      set QD field "leftindent"
      message "\nLeft indent:" set QD
      if !x DefaultRuler
      r toruler set field "leftindent" QD
      }
```

See Also **toruler**

files

Syntax **files**

Returns #

Function Returns the number of open files/buffers (a number between 1 and 24). The editor also supports the idea of "no files." If the only open file is unnamed, has no characters in it, and **modf** is False, then **files** returns 0.

Example

```
CloseFile :
   if modf {
      message "\nThe file"
      message fname
      if (ask "has not been saved, save it?") Save
      }
   if (inbuff themark) (0 -> select)         ; turn off select
   close
   if !files DefaultRuler
```

See Also **flist**

flag3, flag4, flag5, flag6 (Variables)

Syntax **flag***N*

Function These are scratch variables that you can use to control the status line.

Example

```
CtrlKDispatch :
   (0 -> x $) -> x
   if (!menudelay || !(menudelay wait)) (status "\n^K")
   GetKey CharToAlpha case {
      ...
      'N' if ColMode (0 -> ColMode -> flag3)
      else (1 -> ColMode -> flag3),
      ...
      $ 1 Bell
      }
```

The **Flag3** command in this example is used to show the status of column mode in the status line in the default interface.

See Also **flag, statline**

flags

Syntax **flags** "*string*"

Function Prints the contents of several variables on the status line. The specified *string* should be a printf string indicating how to print the flags. The flags are passed to printf in the following order:

> Select, Append, OverWrite, Flag3, Flag4, Flag5, Flag6, Hour, Minute, Hour >= 12

Example

```
NormalMode :
   flags "%[   %:Sel%]%2g%[Ins%:Ovr%]
         %[   %:Col%]%7g%11+12#+2u:%02u%[a%:p%]m"
   mode ""
```

This is the macro used to define the status line flags (that is, the middle section of the status line).

See Also **flag3, flag4, flag5, flag6, mode, statline**

Fn

Syntax

```
f1...f12
```

Function

Names a function key. See the key table in Appendix E for the definition of the keys.

Example

```
F1 : HelpMenu        ; F1
```

This example assigns the *HelpMenu* macro to *F1*.

flist

Syntax

```
# flist "string"
```

Function

If encountered by itself, the macro inserts all the matching file names into the buffer, with a hard return after each.

You will normally place this macro anywhere a string is expected; the macro allows the user to enter or construct a file name.

If the string contains a wildcard (* or ?), a menu is drawn of all matching names, and the user is expected to pick one.

If the string contains a %, it is constructed out of the current **fname** and the supplied name:

% turns directly into **fname**

%.xxx turns into **fname** with the extension changed to *.xxx*

xxx% tacks the base part of **fname** onto *xxx*, which is usually a directory.

A string ending with :, /, or \ automatically has a % added, so that *x*: is the same as *x*:% and /dir/ is the same as /dir/%.

Strings that don't contain %, :, /, and \ are returned exactly.

The # is a combination of the following argument switches:

2 Searches the path for any matching files. If the user enters a * or if the 4 bit is set, **flist** draws a menu listing all matches found in each directory on the path. When the user picks one, **flist** returns the file name without its directory name.

4 Forces the directory menu. A menu is drawn of all matches (usually 1) even if there is no * or ? in the name. In addition, names ending in :, /, or \ have an * wildcard added to them (rather than a %), so they list the contents of a disk or directory.

8 Hides the extensions in the directory listing and returns the file name with the extension removed.

If you place **flist** where a string is not expected, it automatically uses 4 as the argument and inserts a list of matching files into the buffer. This capability can be useful for constructing batch files.

Example

```
MacroLoad : 10 set Q0 flist "*.spm" mread Q0
```

This example is the one used in the default Sprint interface to provide the user with a list of .SPM files from which to choose. It searches the path for all files with the .SPM extension and displays them in a menu *without* their extension.

See Also **fcopy, fdelete, fmove, fname, pickfile**

fmove

Syntax `fmove "filename1" "filename2"`

Returns T/F

Function

Moves (renames) a file. This does precisely the same thing as **fcopy**, except the original *(filename1)* disappears. This is different from a DOS REN command in that it can move files between directories (note that renaming / a/b to c: will move from directory /a to the current directory) and will even move between disks (by copying and then deleting the original).

Returns if successful, aborts if unsuccessful for any reason (usually that the source does not exist).

If the destination exists, the user will be asked if it should be replaced. To prevent this question, the macro should delete the destination first. If the user answers no, an **abort** is done, so further commands in the macro are not done.

Example

```
RenameFile :
   message "File to rename or move: " set Q0
   set Q1 Q0
   set Q0 flist Q1
   if !exist Q0 {          ; if spec'd file mask had no matches
      set QD Q1
      mark { to QD "No files match'" toend "'." }
      error QD
      }
   set Q1 "" message "Rename/move" message Q0
      message " to: " set Q1
   0 AllCaps
   1 AllCaps
   fmove Q0 Q1
   message "\nRename complete."
```

The **fmove** command in this example renames the file from the name specified in the Q0 register to the name specified in the Q1 register.

See Also

fcopy, fdelete, flist

fname

Syntax

fname

Function

You can use **fname** anywhere a string is expected to get the current file name (complete with full disk and directory name). You can use the **set** macro to change the **fname**.

If you just want the name of the file without the disk and directory name, use the **cdstrip** macro.

Example

```
IsUnnamed : length fname = 0
```

This example macro returns True if the file is unnamed (that is, if **fname** is an empty string).

See Also **cdstrip, fchange, fcopy, fdelete, flist, fmove**

fontcpi (Variable)

Syntax **fontcpi**

Returns #

Function A recommended CPI (characters per inch) setting for the font selected by the last **pickfont** macro (see **pickfont**). For instance, the font "pica" would probably return 10, and "elite" would probably return 12 (the value is determined by the printer description).

Proportional-spaced fonts would have higher values, like 15. By setting **cpi** equal to this, and then setting the margins and indents to the correct number of "inches" for that **cpi**, the onscreen text will approximate the printer output (about 90% accuracy, good enough to do manual hyphenation). Unfortunately, the text will generally be wider than the screen, making it hard to read and edit.

Example

```
SettheFont :
   mark {
      r toruler -> x
      set QD field "font"
      if !x DefaultRuler
      r toruler set field "font" pickfont QD
      }
   if (fontcpi != 0) fontcpi -> cpi
```

This example moves the value of **fontcpi** into **cpi**.

See Also **cpi, pickfont**

forced

Syntax

```
forced command
```

Returns

Result of command

Function

Executes a command but forces **keypressed** to be False while doing so. This has the effect of not letting any work be interrupted by keystrokes, although keyboard input (such as *GetKey*) will still work.

For example, **forced draw** is the easiest way to make sure that the screen is fully up-to-date. In normal circumstances, you don't need to use **forced**, since the editor is designed to work without it. In particular, the screen redisplay will normally be incrementally updated at the correct time.

Example

```
Init : NormalMode forced draw ...
```

found

Syntax

```
found
```

Returns

T/F

Function

After a search or a match, this command moves the point to the other end of the located string. Repeatedly executing **found** alternates the point between the two ends of the string. Usually this is used to identify a string to be highlighted or replaced.

Example

```
HiLiteFound :
   mark (found 1 -> select draw 0 wait Unselect swap themark)
```

The **found** command in this example is used to move the cursor to the end of the string to be highlighted.

See Also

draw, mark, search, select

get

Syntax

```
# get "prompt"
```

Returns

#

Function

Asks the user to enter a number; when the user enters a valid number, it appears on the status line and can be edited with normal *EditKey* editing. The user can edit the number and confirm the response by pressing *Enter.* If a numeric argument is provided, it is used as a default reply. The number can also be given with a terminating *H, O,* or *B* to indicate hex, octal, or binary, respectively. The **get** macro automatically adds a colon and a space (:) to the end of the specified *prompt.*

Example

```
GoToLine :
   if !select set themark
   LastLine get "Line number" -> LastLine -> line
   if (Line != LastLine)
      message "Line was out of range. At end of file"
```

This example provides the "Get line number" function in the default Sprint interface.

See Also

ask, message, prompt, rangeget

GetKey

Syntax

```
GetKey
```

Function

When the editor's internal typeahead queue is empty and needs a keystroke, *GetKey* is automatically called, and the result is assumed to be the necessary keystroke. *GetKey* should combine any intermediate results into a single keystroke; for instance, it should combine the IBM null prefix code (generated by Function keys) with the following code (Function key scan code) to get a number greater than 255.

GetKey can also be programmed to do any other desired input parsing. For example, it could generate the keystrokes by reading a string. *GetKey* can also use hardware to determine in non-DOS ways (such as doing the BIOS call) what keystroke is next.

Technical Note: The editor checks the keyboard status (using the DOS "ioctl" call) to determine when the input is ready for purposes of minimal screen or swap file update, and the editor expects that if and only if this is True, *GetKey* will return immediately. If input appears to be ready all the time, the screen will not change. If it appears to never be ready, the editor will completely update the screen and swap file after each keystroke.

Make sure *GetKey* does not do anything that might cause it to be called recursively; that is, don't do any user prompts, key, and any loops (which will recall it for abort checks).

Example

```
GetKey : key || key + 256
```

This is the definition of *GetKey* in the default Sprint interface.

See Also

EditKey, MenuKey

gmark

Syntax

```
gmark
```

Function

This is the global mark (that is, the one on the top of the stack if no others are pushed). If there are no enclosing **mark** commands, this is the same as **themark.**

Example

```
mousetrack :
   ...
   x case {
      140H set themark set gmark 1 -> select,
      143H if (at themark) 0 ->
         select set themark 0 -> x break,
      144H tosol set themark set gmark 1 -> select toeol,
    ...
```

See Also

mark, the (mark), tog (mark)

hardware

Syntax

```
# hardware "string"
```

Returns

Result of last hardware operation

Function

Issues commands directly to the hardware using the contents of the string.

You use hardware *control strings* with the **hardware** macro. These strings allow three things to be done: writing memory locations, writing I/O ports, and doing random interrupts to call ROM utilities.

The string is read from left to right and can contain numbers and operator symbols. Spaces serve to separate words, but are otherwise ignored. At any time a single "argument" is preserved; some operators set this argument, others use it.

Numbers must start with a digit and can end with an *H*, *O*, or *B* to indicate hex, octal, or binary, notation (otherwise, they are decimal). If a number is encountered in the string, the argument is set to it.

The operators available in **hardware** control strings are

% Sets the argument to a code off the "argument list."

address Sets the argument to the contents of a given byte of memory. The address can be a single number, indicating something in segment zero, or it can be *number:number*, indicating a segment and offset. Don't forget to put *H* on the end of the numbers if you want hex addresses.

> *address* Sets the given byte to the argument. The argument does not change.

>| *address* ORs the argument with the contents of the byte. This can be used to set various bits.

>& *address* ANDs the argument with the contents of the byte. This turns off bits.

>^ *address* XORs the argument with the contents of the byte. This toggles bits.

> *reg* Sets a given register to the argument. These register values are used during the next interrupt. On an 8086, the legal registers are AH, AL, AX, BH, BL, BX, CH, CL, CX, DH, DL, DX, SI, and DI. (**Note:** You can't set the segment or BP registers.)

in number Sets the argument to the input from the given I/O port.

out number Sends the argument to the given I/O port.

int number Does an interrupt. The argument is put in the AH register; the other registers are set as per the most recent > *reg* instructions, then an *int* instruction is done. The argument is set to whatever is an AX when the interrupt returns.

Warning: Don't use this command unless you really know what you're doing. You can easily crash the machine.

Example

```
hardware "0 > CH 8 > CL 1 int 10h"
```

This example does the BIOS interrupt call that turns the line cursor into a block cursor.

See Also **ioport, peek, peekseg**

if...else

Syntax

```
if Boolean command1 { else command2}
```

Returns Result of *command1* or *command2*

Function Conditionally does *command1* or *command2*, depending on the value of the *Boolean* expression.

Example

```
Bell :
   if isibm sounddur sound
   else rawout "^G"
```

This example performs the **sound** macro if the **isibm** variable is True; otherwise, the example sends a ^G to the operating system, which beeps the speaker.

See Also **case, do-while, while, ?**

imenu

Syntax

```
imenu "title" (# "item" commands, ...)
```

Returns Result of commands

Function Just like the **menu** macro, except that the text of *item* selected is placed into Q0 so that it can be referenced (for example, to be inserted into the buffer).

Note: *Item* can be one string or several strings. If it's more than one, the strings are concatenated to form the menu text.

Example

```
PickCommandMenu :
   imenu "Choose Command" {
      "again",
      "bottomOfFile",
      "bottomOfScreen",
      ...
```

This example allows the user to choose one of Potpourri commands in the default Sprint interface.

See Also

menu, qmenu

inagain (Variable)

Syntax

inagain

Returns

T/F

Function

The variable is True if a macro invoked by the **again** command is currently being executed. Macros can check this to more accurately reproduce the last action taken.

Example

```
Quote :
   (if !inagain {
      if (!menudelay || !(menudelay wait)) {
         status "\nControl character to insert:"
      }
```

This example inserts the same character as last time if it was inserted by the **again** macro, rather than ask for a new character.

See Also

again

inbuff

Syntax

inbuff *mark*

Returns

T/F

Function

Returns True if the *mark* is in the same buffer as the point. For more details on what *mark* can be, see **set (mark)** in this chapter.

Example

```
CloseFile :
   if modf {
      message "\nThe file"
      message fname
   if (ask "has not been saved, save it?") Save
   }
if (inbuff themark) (0 -> select)        ; turn off select
close
if !files DefaultRuler
```

See Also **mark, set(mark), themark**

#include

Syntax `#include "filename"`

Function This macro directive compiles the contents of the specified *filename*. If you don't specify an extension, .SPM is assumed. If you are specifying a path name, use forward slashes. The **#include** macros can be nested to any level, within the limits of the files parameter in the CONFIG.SYS file.

Example

```
#include "core"
```

This line in SP.SPM includes CORE.SPM when SP.SPM is compiled.

index

Syntax `N index "string"`

Returns #

Function Searches *string* for the character represented by *N* and returns the offset of the first occurrence. The offset of the first character in the string is 0. If the character is not found, the length of *string* is returned.

You can combine **index** with the **subchar** macro to perform character translations, such as those that change uppercase letters to lowercase letters and vice versa.

Example

```
message "options:"
set Q4
if (('B' index Q4) != (length Q4) {
message "Searching backward..."
...
```

This example determines if a user enters "B" in a string and makes a backward search if they did.

See Also **subchar**

indent (Variable)

Syntax `indent`

Function The value of the indent field from the current cached ruler line.

Example

```
SetLeftMargin :
   set QD "Left margin" leftmargin GetColumn -> x
   if (x >= rightmargin)
      (error "Left margin must be less than right")
   else {
      mark {
         InsertFirstRuler
         x -> leftmargin
         x -> indent
         writeruler refill
         }
      }
```

This example asks the user for the setting for the left margin and moves that value (if it is less than the right margin) into the **indent** variable and into the **leftmargin** variable.

See Also **leftmargin, readruler, writeruler**

infobox

Syntax **infobox** "*title*" (# "*item*", # "*item*", ...) (*commands*)

Returns Result of chosen command

Function Draws an information box. The box is drawn in the same fashion as done by the **menu** macro. The box will remain on the screen during *commands*; after that, any menu or prompt causes the box to disappear.

Note: *Item* can be one string or several strings. If it's more than one, the strings are concatenated to form the menu text.

Example

```
ToneMenu :
   do  {
      infobox "Tone" {
         soundfreq "Pitch\>%d Hz  ",
         sounddur "Length\>%d msec",
         "_",
         "UP\>Higher Pitch",
         "DOWN\>Lower Pitch",
         "LEFT\>Shorter Length",
         "RIGHT\>Longer Length"
         }
MenuKey case {...
```

See Also

imenu, menu, qmenu

Init

Syntax

```
Init :
```

Function

If defined, this macro is automatically called when the editor first starts up. When an **ovlread** is performed, *Init* is also done, but only if the argument is negative.

After *Init* is done, the *Main* macro is performed.

Example

```
Init :
   NormalMode
   forced draw
   0 -> n                    ; n = count of command-line files
     -> ALineLength     ; ASCII mode is forced off on startup.
     -> RulerEdit
     -> InHelp
   ...
```

This example shows the beginning of the *Init* macro in CORE.SPM.

See Also

InitArg, Main

InitArg

Syntax

```
InitArg
```

Function

After the editor calls *Init* or *Restart*, it calls *InitArg* for each file name provided on the command line that started the editor. This name is put in Q0, and then the editor calls *InitArg*.

Note: The editor considers any word on the command line that does not start with a dash to be a file name.

Example

```
InitArg :
   if (n < 6) (++n -> windows)
   0 AllCaps                           ; convert name to uppercase
   if (0 CheckWild) {
      open ""                     ; force user to select file name
      draw
      close
      set Q1 Q0          ; preserve so Q1 starts w/proper guess
      4 set Q0 flist Q1
      if !length Q0 {...
```

This example shows some of the structure of the *InitArg* macro in CORE.SPM.

See Also

Init, Main

inruler

Syntax

inruler

Returns

T/F

Function

Returns True if the point is in a ruler line (that is, on the ^K or anywhere between the ^K and the closing hard return). Returns False if the **raw** macro variable is True, or the point is not in a ruler line. This flag is used to decide whether an unbound key should be inserted into the text or used to edit the ruler. By examining this, a macro can act differently if it is executed in a ruler, so the characters in the ruler are not changed unexpectedly.

Example

```
IsOnlyRuler :
   mark (r to isend inruler && (toeol c isend))
```

This example checks to see if the ruler line is the only line in the file (that is, if the character immediately after the ruler line is the end-of-file marker).

See Also

insertruler, readruler, writeruler

insert

Syntax **# insert**

Returns True

Function Inserts a specified character code into the buffer. Always returns True. If the swap file fills up, an error message is produced, and the macro is aborted.

Example

```
OpenLine : repeat ('^J' insert r c)
```

This example inserts a hard return (^J) as many times as specified by the user.

See Also **insert** "string"

insert "string"

Syntax **insert** "*string*"

Function Inserts the specified *string* into the buffer. The string can be a Q register.

Example

```
InsertBegin :
   mark {
      '^J' r csearch
      if (current = '^J') c
      if inruler (toeol c)
      "^OBEGIN "
      insert Q8
      "^N^J"
      }
```

This example inserts the begin format marker and uses the name of the format in the Q8 register (a preceding macro would have already collected the name of the format).

See Also **insert**

insertruler

Syntax **insertruler**

Function Inserts a copy of the current cached ruler at the start of the current line and leaves the point before the ^K. If you want this ruler to be different from the current one, you should modify the ruler settings and then do **insertruler.**

Performing an **insertruler** turns off the **rulermod** variable.

Example

```
DefaultRuler :
   65 -> rightmargin 5 settab insertruler 0 -> modf
```

This example inserts the default ruler line into the file.

See Also

readruler, ruleredit, rulermod, writeruler

int (global)

Syntax

```
int name = value
```

Function

As long as **int** appears outside of any macro definition, it declares a global variable called *name* and initializes it to *value,* if *value* is specified. *value* must be a single integer or character expression. If *value* is not specified, *name* is initialized to 0.

Example

```
int SearchDirection 1      ; 1 = last search forward, else back
int SearchOpt       3       ; standard options to search macro
int GlobalSearch    0       ; 1 = search from beginning of file
                                                   ; 0 = from point
int GlobalReplace   0                    ; passed into DoReplace
int StrFound        0                        ; used by DoReplace
```

These lines from CORE.SPM define variables to hold the search options.

See Also

int (local)

int (local)

Syntax

```
int name
```

Function

Declares a local variable called *name* and initializes it to the current argument. The variable can be any name and will override any earlier definition of that same variable. Such variables are strictly local; if macro *A* defines a local *x* and calls macro *B,* macro *B* cannot refer to *A*'s *x.*

Local variables declared inside parentheses exist only inside that expression. Outside of parentheses, the local variables last for the entire macro.

The stack for local variables has 256 entries, so you can't put more than 256 bytes of local variable names in a single macro. Local variables can't be declared within parentheses more than 10 deep. Also, don't use a local variable declaration in a statement that's not in parentheses. For example, the statement **`int`** *x* **`int`** *y* **`if`** `0` **`int`** *z* is not valid.

Example

```
Sum :
   int x
   int y
   get "First Number" -> x
   get "Second Number" -> y
   x + y message "Sum = %d"
```

See Also **int** (global)

ioport (Variable)

Syntax **`ioport`** *#*

Function An 8-bit machine I/O port at the address #. If you assign it, an *out* instruction is done; if you read it an *in* instruction is done. Using this command, you can go straight to the hardware. But be careful; you can easily crash the machine.

Example

```
0b6h -> ioport 43h
```

This example sends 0b6h to the port at 43h.

See Also **hardware, peek, peekseq**

isascii

Syntax **`isascii`**

Returns T/F

Function Returns True only if the normal ASCII character set uses the character after the point for the same purpose as Sprint. This is true for all characters greater than 32, and for a tab (^I), carriage return (^M), and hard return (^J).

If you are translating a file into ASCII, you should replace any non-ASCII character with a space if it's a visible character, or with nothing if it's invisible.

Example

```
MyExport :
   while (to !isascii) {
      isvisible ?(' ' -> current)
      else (del)
      }
```

This example does a very crude export to ASCII.

See Also

isend, isnl, ispara, issent

isclose

Syntax

isclose

Returns

T/F

Function

Returns True only if the character after the point is a ^N close delimiter.

Example

```
InsertNL :                          ; turns off the attribute
   if AutoCorrect CheckLastWord
   past isclose
   ALineFill
   '^J' insert
```

This example moves the point past any closing delimiters before inserting a hard return.

See Also

isopen

isend

Syntax

isend

Returns

T/F

Function

If the direction is forward, this returns True if the point is at the end of the buffer. If the direction is reverse, this returns True if the point is at the start of the buffer.

Example

```
IsOnlyRuler :
   mark (r to isend inruler && (toeol c isend))
```

This example moves to the front of the file and then finds out whether the end of the file is immediately after the ruler line.

See Also **isnl, ispara, issent**

isgray

Syntax `isgray`

Returns T/F

Function Returns True if the current character is whitespace or a line delimiter. This macro is the same as `(iswhite || isnl)`.

Example

```
ParagraphFwd :
   past isgray
   '^J' csearch
   past isnl
```

The **past isgray** commands in this example make sure that the cursor is currently in the body of a paragraph.

See Also **isnl, iswhite**

isibm

Syntax `isibm`

Returns T/F

Function Returns True if the current machine fits the internal definition of whether it is an IBM PC.

Example

```
IsShift :
   0 -> peekseg isibm && (peek 417H) & 3
```

This example uses **isibm** to check on the value of the Shift key.

See Also **hardware**

isin

Syntax # **isin**

Returns #

Function

Determines what delimiters surround the current point. # is the code of an open delimiter. If the point is not between that delimiter and its matching ^N close delimiter, **isin** returns 0. If the point is between # and its matching close delimiter, **isin** returns the number of nesting levels to that delimiter; that is, 1 is returned if the point is right after the open delimiter, 2 is returned if the point is inside another delimiter inside the specified one, and so on.

In addition, specifying the input # as 0 matches all delimiters, so that 0 **isin** returns the total number of nested delimiters around the current point.

If you want to copy or compare the current setting, use the **curatt** macro variable, which is a single number.

Example

```
HypWord :
   if ((0 subchar engine) != 'h')
      { status "Loading hyphenator..." }
   set engine "hy_disk hyam"
   mark {
      if istoken (past istoken)
      else (r to istoken)
      if !('^O' isin) {
         29 r runengine past istoken
         }
      }
```

See Also **curratt**

ismarkset

Syntax *N* **ismarkset**

Returns T/F

Function

Returns True if mark*N* has been set. If that mark has not been set, it is set to the current point the first time you use that mark.

See Also **mark, set**

isnl

Syntax **isnl**

Returns T/F

Function Returns True if the current character is a line delimiter, that is, a NL, SpaceNL, HyphenNL, or CR. If you want to check only for a real NL (a hard return), do **current** = 10.

Example

```
ParagraphFwd :
   past isgray
   '^J' csearch
   past isnl
```

The **past isnl** command in this example makes sure that the cursor is currently at the beginning of a paragraph.

See Also **isgray, isend, ispara, issent**

isopen

Syntax **isopen**

Returns T/F

Function Returns True if the current character is a Sprint open delimiter; that is, if it is a ^A through ^E or ^O through ^X.

Example

```
FixRegion :
   int n
   regionfwd {                    ; get point before the mark
      f past isclose   ; don't include leading close delimiters
      r past isopen      ; include all leading open delimiters
      ...
```

The **isopen** command in this example makes certain that open delimiters will be included in region commands.

See Also **isclose**

ispara

Syntax **ispara**

Returns T/F

Function

Returns True if the point is on a hard return or at the end of file.

Example

```
IsBlankLine : ispara && column = 0
```

This example checks to see if the point is on a hard return or at the end of the file, and the column is equal to 0; thus, it returns True if the current line is blank.

See Also

isgray, isnl, issent, iswhite

issent

Syntax

```
issent
```

Returns

T/F

Function

This tries to detect if the point is before the punctuation mark at the end of a sentence. It returns True if **ispara** is True. It also returns True if the current character is a period, question mark, exclamation point, or colon, followed by zero or more close delimiters (^N and any of the set)] } > ' "), followed by an **isgray**. Otherwise, **issent** returns False.

Example

```
SentenceFwd :
   past isgray
   to issent
   to isgray
   past iswhite
```

The **issent** command in this example makes sure that the point moves to the end of the current sentence.

See Also

isgray, isnl, ispara, issent

istab

Syntax

```
# istab
```

Returns

T/F

Function

Returns True if there is a tab stop in the current ruler line at column #.

Example

```
if (x istab) (x -> leftmargin)
else (leftmargin + 5 -> leftmargin)
```

This example checks if there is a tab at a specified number and, if there is, sets the left margin to be the same as the tab.

See Also

tabsize

istoken

Syntax

`istoken`

Returns

T/F

Function

Returns True if the current character is considered part of a word. Letters, numbers, the underscore, @-sign, HYPHEN (that is, a discretionary hyphen, ^^) and HyphenNL (that is, a discretionary hyphen that appears as a dash at the end of a line, ^]) characters, and any character with the high bit set are considered tokens. Also, apostrophes and dashes (ASCII 45) are tokens only if the characters on both sides of them are also tokens.

Example

```
SentenceBack :
   r to istoken
   if inruler (r tosol r to istoken)
   r to issent
   past isgray
   if inruler (toeol c)                ; catches rulers at TOF
```

The **r to istoken** command in this example makes sure the point is backed up to a character considered to be a word.

See Also

isascii, isgray, iswhite

isvisible

Syntax

`isvisible`

Returns

T/F

Function

Returns True if the current character is "visible" on the screen. Only open and close delimiters and discretionary hyphen (^^) are "invisible." Everything else, including ends of lines and tabs that have zero width due to a preceding spring, is considered visible. If the **raw** macro variable is set, everything is visible.

The **isvisible** macro returns 2 if the current character is HyphenNL, which indicates the character is normally invisible but is currently visible. (The HyphenNL appears as a dash only when at the end of a line; otherwise, it's hidden.)

Example

```
DelBack :
   while (r c !isvisible) $
   DelFwd
```

This example makes sure that invisible characters are not deleted; in particular, this macro prevents open delimiters from being inadvertently removed while backspacing.

See Also **isascii, istoken**

iswhite

Syntax `iswhite`

Returns T/F

Function Returns True if the current character is a space, SpaceNL (soft return), tab, or wide space (spring); otherwise, returns False.

Example

```
SentenceFwd :
   past isgray
   to issent
   to isgray
   past iswhite
```

The **past iswhite** command in this example makes sure that the point is moved past any whitespace.

See Also **isgray, isnl**

jamount

Syntax `jamount`

Returns #

Function Moves the point to the end of the line and returns the amount of justification this line needs (that is, the difference between the **rightmargin** and the number of characters in that line).

Example

```
HypRegionRest :
   if ((0 subchar engine) != 'h')
    { status "Loading hyphenator..." }
   set engine "hy_disk hyam"
   while before themark {
      if (jamount > HypAsk && current != '^]')
      ...
```

The **jamount** command in this example is checked to see if it is greater than a predetermined hyphenation amount.

See Also **justify**

justify (Variable)

Syntax `justify`

Function The justify field from the current cached ruler line is encoded as follows:

0 = left ragged
1 = left-justified block
2 = center ragged
3 = center-justified block
4 = right ragged
5 = right-justified block

Example

```
ToggleJustify :
   InsertFirstRuler
   justify ? 0 -> justify : 1 -> justify
   writeruler
```

This example toggles the justification from left ragged to left-justified block or vice versa, as in the Wordstar key combination *Ctrl-O J*.

See Also **jamount**

key

Syntax # **key**

Returns #

Function

Does a 0 **wait**, then gets a single character from either the **keypushback** queue or, if that queue is empty, from standard input.

If **isibm** is True, and if input is not redirected, input is read from the IBM BIOS keyboard calls. The returned values from the BIOS are falsified into a zero-prefix byte stream so that **key** always returns bytes. Because of this translation, the BIOS stream does not match what is returned from the BIOS, although the letters, control keys, and arrow keys return the same things.

Input to the editor may be redirected with < *filename* on the command line. The editor detects this by checking the *isdev* ioctl bit on stdin, and checks for end-of-file. If the editor encounters the end of the input file, **key** does an automatic **exit** and returns ^Z.

Refer to Appendix E for a list and explanation of key codes.

Example

```
GetKey : key || key + 256
```

This example is from the CORE.SPM file and is that file's principal mechanism for retrieving a keystroke.

See Also

GetKey, **keyexec**, **keypushback**, **record**

## *keyexec*

Syntax

# keyexec

Returns

Result of key macro

Function

Executes the macro bound to the key specified by #, which should be in the range 0 to 511 (if not, it is masked to be in that range). Sprint takes action in the following order:

- If # has been assigned a macro, that macro is executed.
- If # is greater than 180H (the meta keys read from the IBM BIOS), it is masked to be less than 128 and tried again.
- If **inruler** is True, # is used to edit the ruler, and the editor enters ruler-editing mode.

- If # is less than 256, the code is inserted into the text; if # is greater than 256, an error message is produced.

The main use of **keyexec** is to use macros to combine multi-character key sequences into a single code.

For example, to simulate the EMACS ^X prefix, you might define ^X as

```
^X : key "C-X-" key + 180H keyexec
```

which effectively converts ^XY into the same code returned by an *Alt-Y*.

The **keyexec** macro saves the current macro state for **again** processing. The argument is first read from the previous saved state, so that an argument (perhaps entered with ^G) can be passed through to a prefixed function key.

The IBM function keys return a zero prefix followed by another code. Thus, the macro for the zero prefix must read that next code and turn out a function key number to perform, as shown in the following example. (Refer to Appendix E for a list and explanation of key codes.)

Example

```
^@ : key + 256 keyexec      ; handle IBM function key 0 prefix
```

This example is from SP.SPM and converts IBM function keys into a code from 256 to 511.

See Also **key**

keyhelp

Syntax

```
# keyhelp
```

Function

Does a "help parse" of the macro assigned to the given key. This results in a word that describes that key, and this word is put in Q0 in preparation for the **DoHelp** macro. This is used for interrogative help, where a macro can ask the user for the key to be described.

If there is no argument, the "help parse" is done to the macro to which the **again** pointer is pointing. This is for the "help on last action" function.

Example

```
HelpMenu :
   1 -> Inhelp
   menu "Help on..." {
      "_Press F1 for Template",
      "Subject" 0 ->
          InHelp message "Enter subject:" set Q0 DoHelp,
      "Key" 0 ->
          InHelp status "\nPress key:"
          GetKey keyhelp DoHelp,
      "Last Command" 0 ->
          InHelp $ keyhelp DoHelp
      }
   0 -> Inhelp
```

This example shows the macro that does the help menu for SP.SPM.

See Also

DoHelp

keypressed

Syntax

keypressed

Returns

T/F

Function

Returns True if there is a keystroke ready in either the **keypushback** queue or in the system.

Example

```
Restart :
   ...
   while keypressed key                   ; eliminate typeahead
   ...
```

This example "swallows" all typeahead keystrokes.

See Also

keypushback

## *keypushback*

Syntax

# keypushback
keypushback "*string*"

Function

The first form of **keypushback** is used to stuff the specified character # into the internal queue. This is designed for one-character pushback, and it is a last-in/first-out queue. (Keystrokes appear in the opposite order in which they were pushed.) To push back a function key, you must push back first the actual code and then the 0 prefix. The size of the queue is currently 1024.

The second form of **keypushback** stuffs all the characters in *string* into the internal queue from last to first, so they will be read back out in forward order. Unless *string* is a Q register, you can't push back zeros (function key prefixes) with this mechanism. This form of **keypushback** is usually used to play back a set of keystrokes created by **record**.

Refer to Appendix E for a list and explanation of key codes.

Example

```
MenuBind :
   do {
      status "\nShortcut for menu item: " GetKey -> x
      if x = '^[' { 0 return }    ; if ESC pressed return Null
      if (x CanAssign) {
         x keypushback
         if (x > 255) (0 keypushback)
         break
         }
      else AssignError
      }
   '^J' return
```

See Also **keypressed, record**

killswap (Variable)

Syntax

```
killswap
```

Function

If this flag is True when the editor exits, the backup swap file is deleted. If False, the swap file is preserved, so the next time the editor is run it comes up in the same state with the same files open. This flag is False when the editor starts up. Notice that if True, the swap file is deleted even if modified buffers exist.

SPRECOVE.COM can be used to create a *permanent* swap file. A permanent swap file causes the editor to ignore **killswap** when exiting. You can delete the file only with the DOS ERASE command. The main use of the permanent swap file is to reserve a set of contiguous disk blocks, which results in faster reading and writing (this can be important on floppy disks).

Example

```
ExitEditor :
   bufnum -> x
```

```
do {
   if modf {
      draw while keypressed (key draw)
      message "\nThe file"
      message fname
      if (ask "has not been saved, save it?") Save
      }
   else if (IsUnnamed && IsOnlyRuler) close
   } while (bufswitch && bufnum != x)
EraseSwap || !files -> killswap
GlossSave                          ; save glossary if in use
exit
```

See Also **exit**

leftedge (Variable)

Syntax `leftedge`

Function The number of the column that marks the left edge of the screen. This is normally 0, but is nonzero if the user is scrolling horizontally. If you set **leftedge** and, in doing so **dcolumn** is placed off the screen, the screen will be recentered the next time a **draw** macro is performed.

Example

```
ScrollRight :
   leftedge + 40 -> leftedge
ScrollLeft :
   if 1 leftedge (1 Bell return)
   if ((leftedge -40) > 0) (leftedge) -40) -> leftedge
   else 0 -> leftedge
```

These are the macro definitions of *ScrollLeft* and *ScrollRight* in the Potpourri menu.

See Also **dcolumn, draw**

leftmargin (Variable)

Syntax `leftmargin`

Function The leftmargin field from the current cached ruler line.

Example

```
SetLeftMargin :
   set QD "Left margin" leftmargin GetColumn -> x
   if (x >= rightmargin)
      (error "Left margin must be less than right")
```

```
...
```

This example queries the user for the value for left margin and moves that value into the **leftmargin** variable.

See Also **rightmargin, settab**

length

Syntax

```
length
```

Returns #

Function Returns the length of the current buffer in characters. It overflows after 65535.

Example

```
mark {
   toend
   length message "File contains %d characters."
   }
```

See Also **offset**

length "string"

Syntax

```
length "string"
```

Returns #

Function Returns the number of characters of the string.

Example

```
SetEmulation :
   status "Getting user interfaces..."
   10 set Q0 flist "*.UI"
   if (!length Q0) error
      "No alternative user interfaces found."
   ...
```

This example uses **length** to check whether anything has been returned in Q0.

line (Variable)

Syntax **line**

Function The current line number in the file. Setting this moves the point to the start of the desired line.

Example

```
GoToLine :
   if !select set themark
   LastLine get "Line number" -> LastLine -> line
   if (line < LastLine) message
      "Line was out of range. At end of file"
   if (LastLine < line) {
      1 -> LastLine message "At beginning of file"
      }
```

This example shows the *GoToLine* macro in CORE.SPM.

See Also **dcolumn, dline, lines**

lines

Syntax **lines**

Function Returns the number of lines in the current buffer.

Example

```
PrintGuess :                          ; try to guess which pass
                                    : this file should print on
   mark {                       ; RUN BEFORE CHANGING BUFFERS!!!
      if (lines > GuessCutOff) (2 return)
      ...
```

This example shows the start of the *PrintGuess* macro in the CORE.SPM file.

See Also **dcolumn, line**

macro

Syntax **# macro** "*string*"

Function Compiles and assigns *string* as a macro to the key given by the code #.

Example

```
MacroEntry :
   message "Enter macro:" set QI
   status "Execute (E) or Assign to a key (A):"
   do {
      GetKey CharToAlpha case {
         'E' macro QI break,
         'A' status
            "\nTo which key should the macro be assigned:"
         GetKey -> x if (x = '^[') abort
         if (x CanAssign) {
            x macro QI
            }
         else AssignError
         break, '^[', abortkey abort, $ 0 Bell
         }
      }
```

This example is from the SP.SPM file. The macro entered in QI will be compiled and then assigned to the key chosen by the user.

Note: The first occurrence of **macro** in line 6 of this example shows the other use of the term; see the next entry.

See Also **macro "string"**

macro "string"

Syntax **macro** "*string*"

Returns Result of macro

Function Compiles the *string,* then executes it once, and throws away the compiled version. This allows macros to be entered directly on the command line.

Example

```
MacroEntry :
   message "Enter macro: " set QI
   status "Execute (E) or Assign to a key (A):"
   do {
      GetKey CharToAlpha case {
         'E' macro QI break,
         'A' status
            "\nTo which key should the macro be assigned:"
         GetKey -> x if (x = '^[') abort
         if (x CanAssign) {
            x macro QI
            }
         else AssignError
         break, '^[', abortkey abort, $ 0 Bell
         }
      }
```

The **macro** command in line 6 of this example (from SP.SPM) will immediately execute the macro the user enters. For an explanation of the **macro** command in line 10, refer to the previous entry.

See Also **# macro**

Main

Syntax

```
Main :
```

Function

All .SPM files that define a complete user interface must have a *Main* macro. The editor automatically calls *Main* upon startup (after *Init* or *Restart* or *InitArgs*) and also immediately after an **mread** or **ovlread** instruction (which have to abort the current macro because they overwrite the space in which it is stored).

When *Main* ends, the editor is exited. *Main* is normally defined as something like **stopped do dokey,** but it can be changed to anything desired.

Note: *Init* is called before *Main* and after **ovlread** or **mread,** if any were negative.

Example

```
Main :
   if !files DefaultRuler          ; make sure files have rulers
   else {
      bufnum -> x
      do {           ; read any files that have newer versions
         if (!modf && (datecheck fname > 0)) {
                                         ; newer version of file
            clear
            read fname
            r toend
            }
         bufswitch
         } while bufnum != x
      }
   do {
      if stopped dokey
      else {
         AppendNext -> append
         0 -> AppendNext
         }
      }
```

This is the *Main* macro from CORE.SPM.

See Also

Init, InitArg, Restart

mark

Syntax

```
mark region
```

Returns

Result of region

Function

Pushes a new mark and then executes the region. After that, the point moves back to the mark (that is, the point does not move), and the mark is popped. You can use this to save your place while performing another operation.

The command **mark** is used to "not move" when doing a command. What it does is set a mark, then it does the command, and then it moves the point back to that mark. Therefore, if you do

```
mark (toend "This is the END\n")
```

the cursor will not move from its current position, but the text "This is the END" and a hard return will be added to the end of the file.

Example

```
AllCaps :                                        ; uppercase Qx
   mark { qswitch while !isend ToUpper }
```

See Also

after (mark), before (mark), to (mark), togmark

markN

Syntax

```
markN
```

Function

There are 16 mark variables that store positions (*N* can be 0 through *F* in hexadecimal). Some of the commands that work with any of the numbered marks are as follows:

at mark*N*
before mark*N*
set mark*N*
swap mark*N*
to mark*N*

The first time you use a mark after starting the editor, that mark is set to the current position.

Example

```
DeleteRegion :
   if select {
      FixRegion
      delete togmark
      curatt -> delatt
      Unselect
      1 -> AppendNext
      set markC                      ; save position of deletion
      }
   else DelFwd
```

See Also **after (mark), before (mark), to (mark), togmark**

marknumber

Syntax `marknumber N`

Function Defines a mark as *N*, where *N* can be any expression.

Example

```
MarkerJump :
   status "Go to marker (0-9):"
   GetKey -30h -> x
      (x<10 && x >= 0)? (to marknumber x) else abort
```

See Also **after (mark), before (mark), to (mark), togmark**

markregion

Syntax `markregion commands`

Function This macro is a shortened version of the command

```
   mark (to themark mark (to themark
         regionfwd commands))]
```

Example

```
RegionIndent : || tabsize -> y   ; if no argument, use tabsize
   markregion {
      while (before themark) {
         ...
```

See Also **after (mark), before (mark), regionfwd, to (mark), togmark**

match

Syntax

`# match "string"`

Returns

T/F

Function

Returns True if the point is at the start of the given string. The match is done using the normal search rules for uppercase, lowercase, and wildcards. The point does not move. # is the same argument as for **search.**

Example

```
ModifyEnv :
  mark {
  do {                      ; go to top of current environment
    status "Searching for last format command..."
    r (to isopen)
    if (r isend) {
      message "No commands found"
      return
      }
    if (1 match "begin") {
      ...
```

This example shows the **match** command in SP.SPM that finds the "begin format" statement in order to modify it.

See Also

found, search

menu

Syntax

`menu "title" (# "item" commands, ...)`

Returns

Result of chosen commands

Function

Draws a boxed menu on the screen, consisting of the *items* (the # before the *item* is optional; if you include it, you can put its value into the item with % commands). After the user picks one, the the editor executes the appropriate *commands*. If the *commands* execute normally, the menu is erased and processing continues. If the *commands* are aborted in any way (and no **exitmenus** macro has been included), the menu is redrawn, and the user can pick another item.

Note: *Item* can be one string or several strings. If it's more than one, the strings are concatenated to form the menu text.

The user can also bind a key to an item by pressing *Ctrl-Enter* when that item is selected. This is done by storing with the given key a pointer to the start of the correct menu *commands*. This method is also used to store the location for **again** processing.

The user can choose menu items by moving to them with the cursor keys or spacebar ,and by pressing *Enter*. Alternatively, a menu item can be chosen by pressing a letter for an item that has a capital letter as the first character. (Note that this will choose only the *first* item starting with that capital letter ,if there is more than one match.)

The new menu will appear to the left of any menu already on the screen (whether or not the new menu was called by a macro embedded in the old menu). The menus do not disappear until the screen is redrawn (such as with the **draw** macro).

Menu items that start with an underline character create division bars that can't be selected. Any text after the underline is centered in the division bar. The code assumes that the bottom item in the menu is not a division bar, and that there are no more division bars than the screen is tall.

You can use **exitmenus** and **abort** in your macros to alter the normal use of *Esc* to remove only the last-displayed menu. Instead, you can remove all displayed menus.

By putting **exitmenus** on a key in **menukey**, that key will remove all the menus. This was done to *Shift-Esc* in our default setup.

By putting **exitmenus** on an item in the menu, that item will not return to the menus even if the user does an abort (by pressing *Ctrl-U*). If you put **exitmenus** before a menu, you will prevent a return to that menu. For instance, if you assign *Ctrl-X* to do **exitmenus** *SprintMenu*, you will disable all "popping" of menus.

Assuming **exitmenus** does not override this, you can put **abort** on menu items to cause the menu to **repeat**. This is useful if you want only certain items to repeat. If you want the menu to repeat no matter what the user chooses, put the whole thing in a **do** loop.

Example

```
SprintMenu :
   menu "Sprint" {
      "File"           FilesMenu,
      "Edit"           EditMenu,
      "_",
      "Insert"         InsertMenu,
      "Typestyle"      TypeStyleMenu,
      "Style"          StyleMenu,
      "Layout"         LayoutMenu,
      "_",
      "Print"          PrintMenu,
      "Window"         WindowsMenu,
      "Utilities"      UtilitiesMenu,
      "Customize"      CustomMenu,
      "_",
      "Quit"           ExitEditor
      }
```

This example shows the *SprintMenu* macro from SP.SPM.

See Also **abort, exitmenus, imenu, infobox, qmenu**

menudelay (Variable)

Syntax `menudelay`

Function The time it takes a menu to show up after the **menu** or **imenu** macro is started. This delay is typically specified in milliseconds on a normal IBM PC (and shorter on faster machines). If you set this to 0 (the default), the editor draws the menus immediately.

The **menudelay** is not used for **qmenu** or for internal menus, such as those used by the spelling corrector.

Example

```
...
(menudelay/100) "Menu Display Delay\>%[NO DELAY%:%4u%]"
   menudelay/100 rangeget 100
   "Delay before menu display (in tenths of seconds)"
   * 100 -> menudelay,
   ...
```

These examples show the macros used to change menu delay in SP.SPM.

See Also **imenu, menu**

MenuKey

Syntax `MenuKey`

Function

Returns a **menu** keystroke. This macro should be defined to parse and interpret a key for menu item selection and return a number that tells the internal menu code what to do next.

The return values are as follows (most of these are 100H plus the IBM "scan code" for a given function key, making the macro simple for an IBM):

0	: throw away keystroke and redraw current menu
147H	: home
148H	: up
149H	: page up
14BH	: left
14DH	: right
14FH	: end
150H	: down
151H	: page down
18DH	: bind a key (Shift-Enter)
101H	: help (F1)
''	: go down/right
'^H'	: go up/left (backspace)
'^?'	: go up/left (backspace)
'^M'	: execute/confirm (CR)
'^['	: exit/cancel (ESC)
abortkey	: exit/cancel

All other codes less than 100H or greater than 180H are masked to capital letters and used to pick an item.

By changing *MenuKey*, you can change the actions of the keys in a menu, so selection works the way you want it to. For instance, to make a function key do the bind action, have *MenuKey* translate it to 18DH. To make a function key execute the current item, translate it to ^M.

This is also used for non-IBM clones, to translate whatever they send for the arrow keys to these codes so the menu-picker can understand it.

If *MenuKey* is not defined, the editor does **key** instead. Refer to Appendix E for a list and explanation of key codes.

Example

```
MenuKey :
   GetKey -> int ktmp case {
      '^C'      151h,                               ; page down
      '^D'      14dh,                                   ; right
      '^E'      148h,                                      ; up
      '^I'      150h,                                    ; down
      '^J'      MenuBind,                              ; rebind
      '^R'      149h,                                 ; page up
      '^S'      14bh,                                    ; left
      '^U'      exitmenus '^[',
      '^X'      150h,                                    ; down
      '^['      if IsShift exitmenus '^[',
      101h      inhelp ? (HelpTemplate 0) : 101h,   ; F1: help
      10Ah      '^M',                             ; F10: accept
      14ch      150h,    ; make '5' be down arrow if code comes
      140h,144h '^M',                ; mouse left key is accept
      141h,145h '^[',               ; mouse right key is cancel
      189h      148h,                             ; backtab: up
      1adh      0 -> ShowKeys 0,                        ; grey-
      1abh      1 -> ShowKeys 0,                        ; grey+
      19bh      exitmenus '^[',                  ; Ctrl/Alt-ESC
      $         ktmp
      }
```

This example is the *MenuKey* macro from SP.SPM.

See Also *GetKey*

message

Syntax `# message "string"`

Function Prints the *string* (the optional argument (#) can be included with % commands) on the status line. If you put *n* at the start of the string, the string overwrites any messages already on the status line; otherwise, the message is appended to the existing messages.

You can use a Q register as the source for the message; the characters in the Q register are echoed literally, regardless of *n* or imbedded % characters.

Messages stay on the status line until

- a **message** or a **status** command starting with *n* is performed
- after a question is asked on the status line
- the user presses a key

The **message** command should be used to report back to the user and to set up prompts for questions. If you want the messages to disappear when an operation finishes, use the **status** command. If you want a message to stay through many redisplays, use the **mode** command.

Example

```
InsertFigure :
   set QD ""
   message "Caption:" set QD
   set Q8 "FIGURE" MakeBegEnd
   if length QD {
      "^J^OCAPTION" insert QD "^N"
      r (tosol c)
      }
```

The **message** command in this example asks the user to enter a caption for a figure.

See Also **ask, error, mode, prompt, status**

mode

Syntax

```
mode "string"
```

Function

Sets up the string displayed on the second controllable status line.

The second status line is normally not shown but can be turned on or off by setting the appropriate values for **statline**. Generally, you use the second status line to show a "modal" instruction, such as "Keyboard recording on; press ESC to cancel."

Example

```
KeyRecordGloss :
   mode "Keyboard recording on. Press ESC to cancel."
MacroCallBegin
   set QF ""
   if record {
      1 Bell message "\nRecording canceled."
      0 -> record
      1 -> statline
   }
else {
   15 -> record
   2 -> statline
   }
```

See Also **flagN, flags, statline**

modf (Variable)

Syntax `modf`

Returns T/F

Function This flag is True if the current buffer has been modified since it was last read or written. You can set this on or off to fool the editor into thinking changes have or have not been made. The asterisk (*) in the status line is on if this is True.

The **modf** variable can also be set to 2 to indicate "read only." The editor will then refuse to insert any characters into the buffer and will not allow the buffer to be written.

Example

```
DefaultRuler :
   65 -> rightmargin 5 settab insertruler 0 -> modf
```

This example resets the **modf** variable to 0 so that the insertion of the original ruler is not viewed by the editor as a modification to the file.

See Also **ovlmodf, smodf**

mousecursor (Variable)

Syntax `mousecursor`

Returns T/F

Function This flag controls where the cursor is positioned on the screen. If **mousecursor** is False, the cursor is placed at **dline/column**, which is how most stream editors place it. If **mousecursor** is True, the cursor is placed at **dline/dcolumn**, which means it does not actually correspond to where the next character will be inserted, but it moves smoothly up and down between lines. This is useful if you are controlling the cursor with a mouse.

Example

```
mousetrack :
   int x
   1 -> mousecursor
   0 -> select
   mark do { ... }
   $ 0 -> mousecursor
   if x (x keyexec)
```

This example turns mouse tracking on and off.

See Also **column, dline**

move

Syntax

```
# move
```

Returns T/F

Function Moves # characters forward or # characters backward (if # is negative).

Example

```
5 move
```

This example moves five characters forward.

See Also **c, f, r**

mread

Syntax

```
mread "macrofile"
```

Function Reads and compiles an .SPM file. Because this writes over the existing macros, this aborts the current macro (you can't put anything after the **mread**). Any commands in the macro file (lines starting with just ":") will be done. After the macro file is read, the *Init* macro (whether read from this file or left over from older **mreads**) will be executed.

Example

```
QuickCard : 2 mread "refcard"
```

This example reads and compiles the REFCARD.SPM file, which builds the Sprint Quick Reference card.

See Also *Init*, **macro, ovlread**

nexttab

Syntax

```
# nexttab
```

Returns

#

Function

Returns the next tab stop to the right of column #. For instance, if you are in column 10, this will return a number of 11 or higher, depending on the tab settings. The first column is 0.

Example

```
RulerFromText :          ; Use the current line to set a ruler
   mark {
      tosol
      if isnl return
      mark insertruler
      0 -> x                               ; clear all old tabs
      while ((x nexttab -> x) < rightmargin) (x cleartab)
      ...
```

This example reads in all the tabs on the inserted ruler and clears them.

See Also

cleartab, settab

offset

Syntax

```
offset
```

Returns

#

Function

Returns (in characters) the offset of the point; **offset** overflows after 65535 and looks negative after 32767.

Example

```
CtrlQDispatch :
   (0 -> x $) -> x
   if (!menudelay || !(menudelay wait)) (status "\n^Q")
   GetKey CharToAlpha case {
   ...
   '?' offset message "Cursor is at character %3u.",
   ...
```

See Also

length

open

Syntax

open "*filename*"

Returns

T/F

Function

Creates a new buffer, sets **fname** to *filename,* reads that file into the buffer, and places the point at the start of the buffer. If the file does not exists, **open** returns False but creates the buffer anyway. Also, a null *filename* allows you to create an unnamed buffer.

Example

```
DiskDirectory :
   ...
   "Open" if x (set Q0 "")
   set Q1 Q0
   if !length Q1 (set Q1 "*.SPR")
   set Q1 flist Q1                      ; get a file name in Q1
   if !buffind Q1 open Q1
   set Q0 ""
   exitmenus,
   ...
```

See Also

close, fname

overwrite (Variable)

Syntax

overwrite

Returns

T/F

Function

This flag indicates overwrite mode. If True, keys that have no macros bound to them will replace any printing characters rather than be inserted into the buffer. This does not affect how any macros (such as **insert**) work, so check the flag if you are constructing such things as foreign letters.

Example

```
ToggleIns : ++overwrite
```

This example toggles the overwrite mode on and off.

ovlmodf (Variable)

Syntax

```
ovlmodf
```

Function

Returns True if anything has been changed that would require the overlay file to be written. If you exit while this is set, the last-read overlay (usually SP.OVL) is written with the current macro contents. You can turn off **ovlmodf** to prevent this.

User-defined macro variables are stored in the overlay and changing them sets **ovlmodf**. Therefore, every time you start the editor, the defined variables are in the state they were when the overlay was last written.

Don't set **ovlmodf** needlessly; it results in an annoying delay when exiting the editor.

Example

```
PrintDestToggle :
   !PrintDest -> PrintDest 1 -> ovlmodf
```

This example toggles the printer destination; then, in order to save the toggled state, it sets **ovlmodf** to True.

See Also

modf, ovlread, ovlwrite

ovlread

Syntax

```
# ovlread "overlay"
```

Function

Reads in a compiled overlay file and executes the *Main* macro in that file.

Because this command writes over the existing macros, **ovlread** aborts the current macro (thus, anything after the **ovlread** in the same command chain is ignored).

Because the previous overlay file is lost, you may want to do

```
   if ovlmodf ovlwrite "%"
```

to preserve any user key rebindings before you do the **ovlread**.

The # argument is the same as that for the **flist** macro, with the addition that you can give a negative number if you want to force the *overlay* to be written as SP.OVL upon exit. Moreover, if the argument is negative, the *Init* macro is executed before passing control to *Main*.

Example

```
MacroClear :
   if (2 exist "sprint.ui") (2 ovlread "sprint.ui")
   else mread "sp"
```

This example reads in the SPRINT.UI file if it exists on disk.

See Also **flist, ovlmodf, ovlwrite**

ovlwrite

Syntax

```
ovlwrite "overlay"
```

Function

Writes the current compiled set of macros to the specified overlay file. Use % as the *overlay* name to write to the overlay file most recently read.

When the user exits Sprint, the current overlay file will be written to SP.OVL. If you want your variables saved into a different overlay file, do an explicit **ovlwrite** command to the appropriate file.

Example

```
SaveUI :
   $ ovlwrite "sprint.ui" ovlwrite "sp"
```

This example saves the current overlay file into the files SPRINT.UI and SP.OVL.

See Also **ovlmodf, ovlread**

pageread

Syntax

```
pageread "filename"
```

Function

Interprets the specified file as a .LOG file from the formatter and inserts soft page breaks (formfeeds with a soft return at the end of the line) into the current file at the indicated lines.

The **pageread** command also interprets error log files from other programs such as Turbo C. Any file to be interpreted must consist of lines that are formatted as follows:

```
   xxxfilename xxx# message
```

where *xxx* is 0 or more characters, spaces are one or more spaces or tabs, and # is a set of digits. If the # is greater than or equal to the previous number, the editor goes to that line, inserts a formfeed, the message, and a soft return. Any lines that aren't formatted as shown are ignored.

Example

```
Restart :
   NormalMode
   if (exist "log.$$$") {
      draw pageread "log.$$$"
      fdelete "log.$$$"
      while keypressed key              ; eliminate typeahead
      }
```

This example reads in the LOG.$$$ file.

See Also **readpage**

past

Syntax **past** *Boolean*

Returns T/F

Function If the direction is *forward,* **past** executes *Boolean.* If True, **past** moves right one character, executes it again, and so on until either *Boolean* is False, or it reaches the end of the file. For example, **past iswhite** moves right to the next printing character.

If moving backwards, **past** first moves left one character, then tries *Boolean.* If False, it moves right one before exiting. Thus, **r past iswhite** places the point right after the last printing character.

The *Boolean* expression can be a whole series of commands; however, don't use commands that move the point.

Example

```
WSWordFwd :
   if isnl c else to isgray
   past iswhite
```

The **past** command in this example makes sure that the Wordstar-like "move forward one word" command leaves the cursor on the next printing character.

See Also **isgray, isnl, ispara, issent, istoken**

peek (Variable)

Syntax

peek #

Function

The byte in the given location in memory. Writing (and sometimes reading) it can do strange things to your machine; be sure you know what you're doing. Since the # can range only between 0 and 0FFFFH, you must also set the **peekseg** variable to get at any location on your machine. **Hint:** The IBM keyboard shift flags are at 0417H.

Example

```
IsShift :
   0 -> peekseg isibm && (peek 417H) & 3
```

This example is used to determine the state of the *Shift* key.

See Also

peekseg

peekseg (Variable)

Syntax

peekseg

Function

Read/write location containing the segment number for the **peeks**. You should set this before doing a **peek**. The **peekseg** variable is 0 when the editor starts up.

Example

```
IsShift :
   0 -> peekseg isibm && (peek 417H) & 3
```

This example is used to determine the state of the *Shift* key.

See Also

peek

## *pickcolor*

Syntax

# **pickcolor** "title"

Returns

#

Function

Draws a menu of all attribute numbers from 0 to 255, lets the user pick one, and returns the selected attribute. The number returned is the number that corresponds to the selected attribute.

Example

```
ChangeAtt : -> which
   tct which -> old
   old pickcolor Q0 -> color -> tct which
```

This example (from the file COLORS.SPM) allows the user to set screen attributes.

## *pickfile*

Syntax

`# pickfile "string"`

Function

Draws a menu of all open files (that is, the **fname** of all of the buffers) and lets the user pick one. The current file is at the top of the menu, the *next* one is below it, all the way down to the *previous* one at the bottom. Picking one of the files makes it the current one and makes the current one the previous one.

# determines which file to highlight by default, as follows:

–N	Highlight *N*th + 1 item from bottom of menu
0 or no argument	Highlight previous file at bottom of menu
1	Highlight current file at top of menu
N (positive #)	Highlight *N*th item in menu from the top

If you include the optional *string,* **pickfile** uses that string as the title of the file menu; if you don't include the string, Sprint automatically titles the menu "Open Files."

Example

```
FilesMenu :
   menu "File" {
      "New"             NewFile,
      "Open"            OpenFile,
      "Close"           CloseFile,
      "Insert"          InsertFile,
      "_",
      "Save"            Save,
      "Write As"        WriteFile,
      "Revert to Saved" RevertToSaved,
      "_",
    "Translate"       FileTrans,
    "File Manager"    DiskDirectory,
    "Pick from List"  pickfile
    }
```

This example is the file menu from SP.SPM.

See Also **flist**

pickfont

Syntax **pickfont** "*string*"

Function Reads the printer description of the name set with **printer**, draws a menu of all of the fonts and attributes in it, and lets the user pick one. The **pickfont** command should be used where a string is expected. If a string is given, that font is initially highlighted.

If the user picks a font, the **fontcpi** variable is set to a suggested value for **cpi**; you may want to put **fontcpi** into the ruler line. If the user selects an attribute, **fontcpi** is set to 0.

Example

```
SettheFont :
  mark {
    r toruler -> x
    set QD field "font"
    if !x DefaultRuler
    r toruler set field "font" pickfont QD
    }
  if (fontcpi != 0) fontcpi -> cpi
```

This example provides the font option in SP.SPM.

See Also **fontcpi, printer**

previous (Variable)

Syntax **previous**

Returns #

Function The ASCII code of the character to the left of the point or zero, if the point is at the start of the buffer. This is a read-only variable.

Example

```
InsertTab : -> x
```

```
if AutoCorrect CheckLastWord
if (previous = 32) {
   x repeat ('^I' insert)
   column -> x
   r erase past (current = 32 || current = '^I')
   if (previous = '^_') (r c '^J' -> current)
   while (column != x) ('^I' insert)
   }
else {
   if (previous = '^_') (r c '^J' -> current)
   x repeat ('^I' insert)
   }
```

See Also **current**

prevmark

Syntax `prevmark`

Function Moves one mark down in the stack. If you are inside a **mark** command, **prevmark** is the same as **themark** outside of a mark command.

See Also **after (mark), before (mark), mark, set (mark)**

printer (Variable)

Syntax `printer`

Function Contains the name of the printer. To change the name of the printer, use **set printer** *"name"*.

Example

```
SetQDPrinter :                    ; set QD = name of printer,
                               ; if printer = nul, use 'DEFAULT'
   if !length printer set QD "DEFAULT"
   else set QD printer
```

This example supplies the "set printer" option in SP.SPM.

See Also **pickfont**

put

Syntax

```
# put "string"
```

Function

Inserts the specified number as text into the buffer. If *string* is given, it should contain % commands describing how to format it (otherwise, the number is inserted as a decimal number).

Example

```
TimeDate :              ; insert the date as "January 1, 1980"
   TimeMonth
   time 3 put " %d, "
   time 5 + 1900 put "%d"
```

qmenu

Syntax

```
# qmenu "title" (# "item" commands, ...)
```

Returns

Result of chosen commands

Function

Just like **menu**, except:

- there is no menu delay
- you can't assign keys with *Ctrl-Enter*
- the menu is not returned to if a macro within **qmenu** aborts
- no pointer is set for **again** processing

This is called a **query** menu and is used for asking questions, such as in the Yes/No/And the Rest menu in the Search & Replace process.

The argument is the number of the item to highlight, numbered from 0.

Example

```
ModifyEnv :
   ...
   if (1 match "begin") {
      draw
      qmenu "Modify..." {
         "This environment" break,
         "Previous environment" 0
         }
      }
   ...
```

This example shows some of the macro that does the "Edit Format" function in SP.SPM.

See Also

menu, imenu

qnumber

Syntax

`qnumber N`

Function

Just like invoking a numbered Q register, except that *N* can be any expression. However, **qnumber** does not work as the destination of **copy, delete,** or **to.**

## *qswitch*

Syntax

`# qswitch`

Function

Places the point at the start of the specified Q register, which the macro can edit just like a buffer.It is recommended that you enclose this in a **mark** macro, so the point is returned back to the previous editing buffer. While in a Q register, the editor will act as if **fname** is blank.

# can be from 0 to 25.

Example

```
AllCaps :                                 ; uppercase Qx
   mark {qswitch while !isend ToUpper}
```

The Q register in this example would have already been specified by a former macro.

See Also

to Q0-P

r

Syntax

`r command`

Returns

Result of command

Function

Executes *command* with the direction set to reverse.

Example

```
SetWholeFile :
   r toend set themark 1 -> select toend
```

The **r** in this example makes the **toend** command return the point to the start of the file.

See Also

f

rangeget

Syntax

X **rangeget** "*prompt*" *Y*

Returns

#

Function

Asks the user to enter a number and checks that the number is between 0 and *Y*; *X* is a default response.

The number will be entered on the status line using normal *EditKey* editing. The user can edit the number and confirm the response by pressing *Enter*. If a numeric argument is provided, it is unparsed in decimal as a default reply. The number can also be given with a terminating *H*, *O*, or *B* to indicate hex, octal, or binary, respectively.

Example

```
GoToPage :
   1 rangeget 10000 "Page number"  -> x
   mark {
      ...
```

See Also

ask, *EditKey*, **get**, **message**, **prompt**

raw (Variable)

Syntax

raw

Function

This flag indicates **raw** mode. When **raw** is set to 1, all characters are visible, and the **inruler** variable is always False. All control characters except the line-terminators are displayed in caret notation. Raw mode also disables automatic wordwrap.

Example

```
DelFwd :
   if (raw) (del return)
   ...
```

See Also

inruler

rawout

Syntax **rawout** *"string"*

Function Sends *string* to stdout. You can use **rawout** to draw random things on the screen. Be careful, however; the editor won't know the screen has changed and may not erase the output during the next redisplay.

Example

```
Bell :
   if isibm sounddur sound
   else rawout "^G"
```

See Also **hardware, ioport**

read

Syntax **read** "*filename*"

Returns T/F

Function Inserts the contents of the file into the current buffer, leaving the point after the inserted text. If the file does not exist, nothing is inserted and False is returned.

Example

```
ReReadFile :
   if (exist fname) {
      line -> x
      dline -> y
      clear
      $ read fname
      x -> line
      y redraw                          ; force back to same line
      }
   else (error "File not yet saved")
```

This example from CORE.SPM is used to read the original version of the file from disk when the user wants to revert to the last saved version.

See Also **write**

readpage

Syntax **readpage**

Function When the editor displays the current page number on the bottom line, it keeps a cached copy of it and assumes this cached copy is correct, until the cursor is moved past a page break. The editor cannot know when a page break has been deleted, modified, or inserted. If a macro does any of those actions, it should run **readpage** to force the page break to be reread.

Example

```
ErasePages :    ; depaginate - remove all temporary page breaks
   mark {
      r toend
      while ('^L' csearch)
         (toeol if (current != '^J') (del erase tosol))
      }
   readpage             ; make editor know page # is different
```

This example from CORE.SPM is used to read the original version of the file from disk when the user wants to revert to the last saved version.

readruler

Syntax **readruler**

Function To speed up processing, the editor stores copies of the rulers internally. If the text of a ruler gets changed, or if a ruler gets inserted, the editor might not notice this. The **readruler** command actually causes the editor to throw the current ruler out of the cache; the next redisplay will usually cause it to be reparsed from the file.

Example

```
DefaultRuler :
   ...
   {                                   ; glossary item exists....
      GlossLookUpRest
      0 -> modf                    ; so user can easily throw away
      readruler
      return
   }
   ...
```

The **readruler** command in this example reads in the default ruler.

See Also

insertruler, writeruler

record (Variable)

Syntax

```
# record
```

Function

If you set **record** to any number except 0, all keystrokes are appended to the Q register specified by #. Setting it to 0 turns off recording. To playback the Q register, use the **keypushback** macro. You can check the contents of **record** to see if recording is being done.

Note: You can record any number of keystrokes, but you can only push back 1024 of them before the keyboard buffer overflows.

Example

```
MacroCollBegin :
   if record {
      1 Bell message "\nKey recording canceled."
      0 -> record
      }
   else {
      15 -> record
      }
   set QF ""
```

See Also

keypushback

redraw

Syntax

```
# redraw
```

Function

Redraws the screen from scratch.

The **redraw** macro is much slower than **draw**, so you probably want to use **redraw** only if another program has affected part of the screen, or if you have changed the attribute vector or terminal type, or if you have performed a **rawout**. The # controls the macro as follows:

–1	Redraw the screen so that the cursor is in the middle

0	Redraw the screen exactly as it appears
N (positive #)	Redraw the screen so that the cursor is on the line Nth line

Actually, **redraw** marks all the lines on the screen for updating; the screen isn't drawn until the next **draw** command.

Example

```
Reformat :
   0 redraw draw   ; so the command also fixes corrupt screens
   ...
```

As the comment indicates, the first line of this example fixes the screen.

See Also **draw**

refill

Syntax **refill**

Function

Informs the editor that all lines between the current point and the next ruler needs to be "refilled"; that is, the wordwrapping needs to be refreshed. Any text on the screen will be refilled during the next **draw**, so the screen display always shows the text correctly. However, the editor normally leaves other text in the buffer unchanged. When **refill** is done, the editor continues to work on refilling the appropriate region after the next **draw**. A keystroke will interrupt the process, but then it will continue during the next **draw** until it finishes.

Normally, you won't have to use **refill**, since an automatic **refill** is done when the user exits the ruler. An explicit **refill** is usually only necessary when you modify a ruler directly, or read in a file that is not correctly wordwrapped.

Example

```
SetRightMargin :
   set QD "Right margin" rightmargin GetColumn -> x
   if (x <= leftmargin)
      (error "Right margin must be greater than left")
   else {
      mark {
         InsertFirstRuler
         x -> rightmargin
```

```
            writeruler
            refill
            }
         }
```

This example presents the user with a prompt to set the right margin and then refills the text.

See Also

insertruler, readruler, writeruler

regionfwd

Syntax

regionfwd *commands*

Function

This macro is a shortened version of the command

```
   if after themark (swap themark f <commands> swap themark)
   else f <commands>
```

Example

```
MakeBegEndRegion :      ; make a begin/end pair around a region
   if inruler (toeol c)     ; start following a ruler if on one
   if !select set themark
   regionfwd {
      "^OBEGIN "
      ...
```

This example has a **regionfwd** to ensure that the ^OBEGIN is put at the top of the region.

See Also

after (mark), markregion, swap **(mark)**

repeat

Syntax

```
# repeat command
```

Function

Does the *command* # times. This returns no value unless a **break** exits the loop. The count is checked after *command* is performed, so *command* is always performed at least once, even if the argument is 0 or negative.

Note: If you need to not repeat on zero, enclose **repeat** in an **if** statement.

Example

```
DeleteLine :
   (tosol $)
   delete repeat (toeol c)
   ...
```

The **repeat** command in this example accepts the user's number as a request to delete a specified number of lines.

replace

Syntax

replace *region* "*string*"

Function

Erases the area covered by *region* and inserts the replacement *string*.

If the area corresponds to that found by the last **search** or **match** macro (that is, **region** is a **found** region), the editor performs wildcard replacement for any wildcards passed in the last **search** or **match.**

Example

```
DoReplace :
   0 -> StrFound
   0 -> x                                      ; number found
   set QD Q2   ; extra level of MARK to get around SET THEMARK
   if (SearchOpt & 1)
      (mark (to QD mark (toend RegionLower)))
   mark {
      if GlobalSearch (r toend f)
      while (SearchOpt search QD) {
         1 -> StrFound               ; set to display message
         if GlobalReplace {
            replace found Q3
            ++x status "\nReplaced %d."
            }
         ...
```

The **replace** command in this example replaces the found string with a user-specified string.

See Also

found, match, search

Restart

Syntax

Restart

Function

If started with the "-r" switch, the editor automatically calls *Restart* instead of the *Init* macro.

Example

```
Paginate :
   if (IsOnlyRuler) return
   if (modf || (!length fname)) Save
   status "\nPaginating..."
   PArg+8 call "spfmt -l=log.$$$ -p0" cdstrip fname
   Restart
```

This example from CORE.SPM does repagination and calls the *Restart* macro.

See Also *Init, Main*

return

Syntax **return**

Function Exits the current macro (going back to whatever macro called it) and passes the current argument to the calling macro.

Example

```
DelFwd :
   if (raw) (del return)
   ...
```

This example deletes a character and returns immediately if the **raw** variable is set.

rightmargin (Variable)

Syntax **rightmargin**

Function The *rightmargin* field from the current cached ruler line.

Example

```
DefaultRuler :
   65 -> rightmargin 5 settab insertruler 0 -> modf
```

This example sets up 65 as the value for the right margin in the default ruler line.

See Also **indent, leftmargin**

ruleredit (Variable)

Syntax **ruleredit**

Function Determines whether the ruler can be edited, as follows:

0 Rulers display as plain text and can't be edited, although the user can move the cursor using the arrow keys.

1 Ruler can be edited.

2 Ruler can be edited but, when the user exits the ruler, **ruleredit** is automatically set to 0.

Example

```
ChangeRuler :
   if !rulerEdit {
      mark mark {
         1 -> ruleredit
         r toruler ? {               ; if ruler found, edit it.
            while inruler stopped dokey
            }
         else {
            message "No rulers found."
            }
         0 -> ruleredit
         }
      }
   else message "\nRuler editing is already on."
```

See Also **insertruler, readruler, rulermod, writeruler**

rulermod (Variable)

Syntax `rulermod`

Function Returns True if something has changed in the current cached ruler line, indicating that you should do a **writeruler** or **insertruler**.

Example

```
if rulermod (SetRight mark (writeruler refill))
```

If the **rulermod** variable has been set to True, this example sets the right margin, writes the ruler, and wordwraps the text.

See Also **insertruler, readruler, writeruler**

runengine

Syntax # **runengine** *region*

Returns T/F

Function

Calls the Borland word engine previously named in the **engine** macro variable, passes the engine the text in *region*, and acts on what the engine returns.

The macro executes *region* and sends the area it covers to the engine. Leading blanks, trailing blanks, control characters, and all invisible characters are stripped before passing the text to the region. The largest *region* allowed to be passed is 80 characters.

Note: Most engines are set up to handle only single words or phrases.

The editor will not call the engine with an empty region or a region that starts in a ruler line.

An engine returns True or False based on whether it finds *region* in its dictionary. In either case, an engine can return a list of replacements. If the returned True/False value matches the bits in #, the editor highlights *region* and draws a menu to show the user the replacements. The bits in # are as follows:

1 Draws a menu if the engine finds the word.

2 Draws a menu if the engine does not find the word. Note that if both 1 and 2 are zero, no menus appear, and the True/False value is returned unchanged.

4 All returned dashes are turned into soft hyphens when the replacement is done (although the soft hyphens are not passed to the engine; they're stripped because they're invisible).

8 The entry *Add to Dictionary* is added to the menu. If the user picks the option, the engine is told to add the previous word to its dictionary, and the editor asks as though "Pass" was done. If the engine can't perform this function, it returns an error message.

16 The entry *Replace with...* is added to the menu. If the user picks the option, the engine asks the user to type in a replacement. The engine is not informed about whether the user completes the action.

32 Does the *Lookup* operation without putting a question on the menu.

64 The entry *Ignore* is added to the menu. If the user picks it, the word is added to an in-memory dictionary and will be assumed to be correct for the rest of the editing session. When the session ends, the words are not saved.

Note that an editing section ends when the speller engine is no longer in memory. This will happen if the user loads another engine (Thesaurus or Hyphenation), exits the editor, or calls the formatter for a preview, paginate, or print.

If a user picks the Ignore option, the corresponding bit will be set in the returned value.

128 The entry *Original Word* is added to the menu. If the user picks it, the returned value will have the corresponding bit set. It is up to the macros to handle the replacement.

256 The entry *Previous Word* is added to the menu. If the user picks it, the returned value will have the corresponding bit set. It is up to the macros to handle the replacement.

If an engine returns an empty list of replacements, the item *Lookup* is also put on the menu. If the user picks this option, the editor requests that the engine generate a real list and then redraws the menu. For example, this technique is used by the spelling corrector, which generates the list too slowly to be done without first notifying the user. However, the word *Lookup* will appear for any engine that returns a zero-length list when first passed a word.

The returned value contains the following bits:

1	Engine found the word.
2	User picked something that changed the word.
4	Reserved for future use.
8	User chose *Add to dictionary.*
16	User chose *Replace with....*
32	User chose *Lookup.*
64	User chose *Ignore.*
128	User chose *Original Word.*
256	User chose *Previous Word.*

You can use the return value to make the engine loop as desired.

Example

```
ThesMenu :
   if ((0 subchar engine) != 't')
      { status "Loading synonym list..." }
   set engine "thesaurus"
   mark {
      if select (1 runengine togmark)
      else {
         if !istoken (r to istoken)
         r past istoken
         while (1 runengine past istoken) $
         }
      }
```

See Also **engine**

rwtrans (Variable)

Syntax **rwtrans**

Function Read/write translations. Normally, the editor does not work on the exact image in the DOS file. When a file is read and written, the editor does last-minute translations between what is in the swap file and what is on disk. The **rwtrans** flag controls these translations. When set to 0, no translations are done, resulting in the fastest reading and writing. The bits that control the translation are as follows:

1 Add a ^M before each ^J on writing and then strip them on input. Most modern programs (including the formatter) do not care if the ^M is there; however, a few do require it, such as the DOS TYPE command and MASM. Setting the flag to 0 results in significantly faster file reading and writing.

2 Change SpaceNL (a soft return) to SPACE and HyphenNL to HYPHEN on writing. Each paragraph is written as one very long line (a scheme that is fast becoming a standard). The formatter can format such a line; the only problem is that error messages will have different line numbers than the editor. Also, a refill must be done after reading any file.

4 Strip any trailing ^Z's on input. The need for an EOF marker is left over from CP/M, and probably not required by any MS-DOS program. If you need a ^Z when writing a file, use Sprint's quote facility to place one in the file before writing the file.

By default, **rwtrans** has a value of 5. There is only one flag, and it applies to all files. Setting the flag to zero will result in faster file I/O.

scancode (Variable)

Syntax **scancode**

Function If input is being read from the BIOS, the **scancode** variable is set to the scan code of the last character read.

Note: The code might be wrong if the editor has processed typeahead keystrokes.

The code can be used to differentiate between keys that return identical codes from our BIOS translation. For example, *Shift, Ctrl,* and *Enter* return identical codes but different scan codes.

You can assign values to **scancode,** and they will remain there until the next time a key is read from the BIOS.

However, we strongly recommend that you use **scancode** only to duplicate keystrokes of other word processors.

See Also *GetKey*, **key**

scroll

Syntax **# scroll**

Function Moves the top line of the screen forward or backward # lines (thus scrolling up or down). This does not move the point; if the point goes off the screen, the next redisplay will center it again. If you're using **scroll**, you probably want to check **dline** to handle those boundary conditions.

Example

```
ScreenFwd :
   || wlines-ScrollBorder -> x if x <= 0 (1 -> x)
   x repeat (toeol c)
   if isend return
   x scroll
   dcolumn -> dcolumn
```

This example moves one screen forward.

See Also **dline, scrollborder**

scrollborder (Variable)

Syntax **scrollborder**

Function The number of lines the cursor tries to stay away from the edge of the window (normally 2).

Example

```
ScreenFwd :
   || wlines -ScrollBorder -> x if x  <= 0 (1 -> x)
   x repeat (toeol c)
   if isend return
   x scroll
   dcolumn -> dcolumn
```

The value of **scrollborder** in this example would have been set by a previous operation. (The default is 2.)

See Also **scroll**

## search

Syntax

`# search "string"`

Returns

T/F

Function

Moves the point forward or backward to a match of the specified *string*. The # controls the type of search that is done, as follows:

0	Do an exact literal search
1	Lowercase in search string can match uppercase in file
2	Do wildcards (? and [set]) and understand \ escape character
4	Word-only match (character on each side must be **!istoken**)

The point is left before the first character in the matched string. If not found, the point is restored back to where it started, and this returns False.

Example

```
RunFile :
   set Q0 fname
   if (mark (0 qswitch 1 search ".spm")) {
      if modf Save
      mread fname
      }
   else (message "\nFile does not have .spm extension.")
```

The **search** command in this example is set to ignore case and find any files with an extension of .SPM.

See Also

found, match, replace

select (Variable)

Syntax

select

Returns

`#`

Function

This flag turns on region highlighting on the screen. Everything between the point and the current mark is highlighted the next time the screen is drawn. Also, setting this to 2 makes *column highlighting* for column selection.

Example

```
CopyRegion :
   if select {
      FixRegionNoMod
      copy togmark
      Unselect
      1 -> AppendNext
      }
```

This example copies any selected region into the Clipboard.

See Also **copy, delete, search**

set

Syntax `set "string1" "string2"`

Function Copies *string2* to *string1*. If *string2* is not supplied, the user is prompted for it, and the old contents of the destination are provided as the default.

Q registers can be used as the source or destination.

Example

```
RepeatCount :
   set QD "Repeat" 1 Arg -> RepCount
   key -> RepChar
   if (RepChar >= 32) {
      RepCount repeat (RepChar keypushback dokey)
      }
   else {
      RepChar keypushback
      RepCount dokey
      }
```

This example sets the register QD to the word **Repeat**.

set (mark)

Syntax `set themarkN`

Function Moves the specified mark to the current point. A *mark* is a pointer to a certain location in the text. You can move a mark to the current point; conversely, you can move the point to a mark. For instance,

```
   mark (toeol set themark)
```

will move the point to the end of the line despite the attempts by the first **mark** to restore the point to its original place.

There is a *stack* of marks. The bottommost mark on this stack is called the *global mark*. This is the mark that is used to indicate selected regions in the editor .The top mark is called the *current mark*.

Every mark-setting editing command adds a new mark to the top of the stack. This mark is set to the current point. Then, the next command is executed. When it is completed, the original editing command uses the mark and the *new* point (that is, the cursor position after the second command executes) and does something with them (such as delete the text between them). The mark is then removed (popped) from the stack.

You can set several different kinds of marks, as follows:

gmark	The bottom of the mark stack, set to current position at startup
mark0-markF	The 16 mark variables
marknumber *N*	A mark variable set up by any expression *N*
prevmark	The mark one below the top of the mark stack
themark	The top of the mark stack

Example

```
ToggleSelect :
   if !select {
      set themark
      SelectLoop
      }
   else (0 -> select -> ColMode)
```

If **select** is currently off, this example sets a mark and goes to the *SelectLoop* macro.

See Also

after (mark), before (mark), mark, to (mark)

## *settab*

Syntax

# settab

Function

Sets a new tab stop at # in the internal cached copy of the current ruler line. You should then normally use a **readruler** or **writeruler** command.

Example

```
DefaultRuler :
   65 -> rightmargin 5 settab insertruler 0 -> modf
```

This example sets a tab stop at 5 in the default ruler line.

See Also

cleartab, readruler, writeruler

showkeys (Variable)

Syntax

showkeys

Function

If this variable is set to 1, keyboard shortcuts are shown in the menus. If **showkeys** is set to 0, the shortcuts are not displayed.

Example

```
CustomMenu :
   menu "Customize" {
      ...
      "Ascii File Handling"   AsciiStuffMenu,
      showkeys  "Menu Shortcuts \>%[NO%:YES%]"
      !showkeys -> showkeys abort,
      "_",
      "Options"   CustOptMenu
      }
```

This example sets up the custom menu in SP.SPM.

smodf (Variable)

Syntax

smodf

Function

Returns True if the **tct** array has been changed since the last **swrite** or **sread**. If this is True when the editor exits, an automatic **swrite** is done.

The **smodf** variable is also True if one of the built-in IBM screen types has been selected, so running the editor will always create a DEFAULT.SPS file if none exists. However, SPFMT and SprintMerge will not run without an .SPS file. If you want to prevent the creation of such files, turn **smodf** off during the *Init* macro.

See Also **sread, swrite, tct**

sound

Syntax `sound`

Function Sounds the system speaker. The pitch of the tone is determined by the current setting of the **soundfreq** variable; the duration of the tone is determined by the current setting of the **sounddur** variable.

Example

```
Bell :
   if isibm sounddur sound
   else rawout "^G"
```

This example from CORE.SPM uses the **sound** macro to generate a tone on an IBM PC.

See Also *Bell,* **sounddur, soundfreq**

sounddur (Variable)

Syntax `sounddur`

Function This variable sets the duration in milliseconds of the tone generated by the **sound** macro.

Example

```
Bell :
   if isibm sounddur sound
   else rawout "^G"
```

The **sounddur** in this example sets up the duration of the tone generated on an IBM PC.

See Also *Bell,* **sound, soundfreq**

soundfreq (Variable)

Syntax `soundfreq`

Function This variable sets the frequency (the pitch) of the tone generated by the **sound** macro. The frequency is given in Hz.

Example

```
ToneMenu :
   do {
      infobox "Tone" {
         soundfreq "Pitch\>%d Hz",
         sounddur "Length\>%d msec",
         ...
```

The **soundfreq** in this example allows the user to set the pitch of the tone to be generated.

See Also *Bell*, **sound, sounddur**

sread

Syntax **sread** "*filename*"

Function Reads the terminal description from the given .SPS file (or, if not found, attempts to "guess" whether this is an IBM monochrome or color screen).

See Also **smodf, swrite**

statline (Variable)

Syntax **statline**

Function A status variable that determines what is on the status line. Settings for the bits are as follows:

0 No status line; the status line disappears, giving you 25 editing lines per screen.

1 Status line exists.

2 Display status line and **mode** line.

Example

```
ScreenMenu :
   ...
   !statline "status Line\>%[YES%:NO%]"
      (statline ? (0 -> statline) : (1 -> statline))
   ...
```

These lines present the status line option to the user in SP.SPM.

See Also **flag3, flag4, flag5, flag6, flags, mode**

status

Syntax

status "*string*"

Function

Prints the string on the status line. If a previous **message** command has been done, and the **status** command does not start with *\n, string* will be appended to the end of the existing message.

Status messages stay on the status line until another **message** or **status** is done, or a question is asked on the status line, or the next **draw** is done. If you want the message to remain until the user has read it, use the **message** command. Use **status** for "Please wait a moment" messages.

Example

```
Reformat :
   0 redraw draw
   mark {
      if (ALineLength && !rightmargin) {
         status "Reformatting..."
         ...
```

See Also

ask, error, message, prompt

stopped

Syntax

stopped *command*

Returns

T/F

Function

Executes *command*, and returns True if any of the following are true:

- if the user aborts the command by pressing *Esc* during a loop, or during string input, or during a menu
- if an error occurred
- if the editor executed the **abort** macro

The **stopped** macro normally returns False and is the only way to prevent a user from aborting a macro. Normally, when a **do** or **do-while** loop is executed, the editor checks the keyboard for a press of the **abortkey**. If a **stopped** is executed inside a loop (but not inside an inner nested loop), this test is disabled. This prevents the editor from discarding typeahead keystrokes and from unexpectedly exiting loops.

Example

```
AssignError :
   stopped error "That key cannot be reassigned."
```

This example gives an error message to the user but keeps the automatic **abort** done by the **error** macro fromexiting back to *Main.*

See Also **abort**

subchar

Syntax *N* **subchar** "*string*"

Returns T/F

Function Returns the *N*th character (starting from zero) of *string.* If *N* equals the string length, **subchar** returns zero (values greater than the string length return undefined values).

For example, *N* ***subchar*** *"Constant String"* can be used to implement a lookup table of character codes.

Example

```
ThesMenu :
   if ((0 subchar engine) != 't')
   ...
```

See Also **index, length**

swapdelay (Variable)

Syntax **swapdelay**

Function The time the editor should wait for a keystroke before updating the backup swap file (in milliseconds on an IBM PC). If you set this to zero, the editor will never update the backup file, except for necessary swaps. By default, **swapdelay** is 3000 (3 seconds).

Example

```
CustOptMenu :
   ...
   3 rangeget 60
      "Background save period (in seconds)" -> swapdelay
   swapdelay * 1000 -> swapdelay,
   ...
```

These lines allow the user to set the time between swaps in SP.SPM.

swap (mark)

Syntax **swap themarkN**

Function Exchanges the point and the specified mark.

Example

```
SetEnv :
   markregion {
      InsertBegin
      swap themark
      InsertEnd
      }
   Unselect
```

See Also **set (mark)**

swrite

Syntax **swrite** "*filename*"

Function Writes the current terminal description (including the **tct** array) to the specified .SPS file (you probably want to **swrite** "%").

See Also **smodf, sread**

tabsize (Variable)

Syntax **tabsize**

Function The width of tabs when there is no ruler line or when there are no tabs set in the current ruler line. By default, **tabsize** is 8.

Example

```
PrintOptions :      ; add print options to end of Q reg passed
   ...
   tabsize put "-t = %d"
   }
```

This line in This example places the current value of **tabsize** into the print options.

See Also **cleartab, settab**

tct

Syntax `tct #`

Function Specifies the translation table. # is the index into the array, which specifies how characters show up on the screen. There is one entry for each of the 256 possible character codes, plus some extra entries for other entities.

You can also use **attribute** as an exact synonym for **tct.**

For visible characters, this is the code that is to be put on the screen. For instance, entry 10 is the hard return character. If you put a space in entry 10, the hard returns will print a space on the screen (which can't be seen). If you put 17 in entry 10, the hard returns will print a left-pointing triangle character (on an IBM PC).

For open delimiters, such as ^B, the entry is the onscreen colors to use for that delimiter. When delimiters are nested, the editor XORs each color with the color of the plain text (**tct** 0), ORs the results together, and then XORs that result with the plain color to determine what to display on the screen.

The values for the **tct** entries are as follows:

–5	Color used for infoboxes
–4	Color used for menus
–3	Color used for error messages
–2	Color used for status line
–1	Color used for selected region
0	Color used for plain text
1-5	Colors for ^A thru ^E open delimiters
6	Character used to print at start of ^F springs
7	Not used (^G)
8	Not used (^H)
9	Character to print at start of ^I tabs
10	Character to print for hard new line (^J)
11	Color used to draw ruler lines (^K)

12	Color used to draw a page break line (^L)
13	Character to print for a carriage return (^M)
14	Not used (^N, close delimiter)
15	Color used for ^O commands
16-24	Colors used for ^P through ^X open delimiters
25	Not used (^Y)
26	Not used (^Z)
27	Not used (^[)
28	Character used for hard space
29	Character used for HyphenNL (should be a dash)
30	Not used (HYPHEN)
31	Character used for SpaceNL soft return (should match **tct** 32)
32	Character used for spaces, also tick marks in ruler lines
33-255	Character used for given code in text

Note that the entries for all codes from 33-255 are used everywhere on the screen. Thus, changing the entry for *S* will change how the word *Sprint* is printed on the status line. You can change the menu borders and the dots in ruler lines by altering the entries for the IBM line-drawing characters the editor uses. Be careful; that will also change the entries for those characters in text.

Changing an entry in the **tct** will set **smodf**. The next **draw** will do a minimal update to change the character; this is usually right, but be aware that **draw** does not check for changes to the status line.

Example

```
ScreenMenu :
  do {
    menu "Screen" {
      (tct 10 = 32) "Paragraph Marks\>%[ON%:OFF%]"
        ((tct 10 = 32) ? 17 : 32) -> tct 10,
      (tct 9 = 32) "Tabs\>%[ON%:OFF%]"
        ((tct 9 = 32) ? 16 : 32) -> tct 9,
      ...
```

This example from SP.SPM controls the display of some of the characters on the screen.

themark

Syntax **themark**

Function The top of the mark stack.

Example

```
CloseFile :
   if modf {
      if length fname {
         message "\nThe file"
         message fname
         }
      else message "\nThis Unnamed file"
      if (ask "has not been saved; save it (Y,N,ESC)?") Save
      }
   if (inbuff themark) (0 -> select)         ; turn off select
   close
   if !files DefaultRuler
```

See Also **after (mark), before (mark), mark, set (mark)**

time

Syntax **time** *N*

Returns #

Function Returns a number for some part of the time (the time comes from DOS). *N* specifies the desired part of the time as follows:

0	second (0-59)
1	minute (0-59)
2	hour (0-23)
3	day (0-31)
4	month (1-12)
5	year – 1900
6	day of week (Sunday = 0)
7	day of year (0-365, UNIX only)
8	>0 if daylight savings time (UNIX only)

Example

```
TimeDate :                 ; insert the date as "January 1, 1980"
   TimeMonth
   time 3 put " %d, "
time 5 + 1900 put "%d"
```

This example retrieves the current time and date.

See Also

datecheck

to

Syntax

```
to Boolean
```

Returns

T/F

Function

If **direction** is forward (True), this executes *Boolean,* and if the result is False, the command moves right one character, executes *Boolean* again, and so on until either *Boolean* is True or the command reaches the end of the file. For instance, **to iswhite** moves right to the next whitespace character.

If **direction** is backward (False), **to** first moves left one character, then tries *Boolean.* If True, it moves right one before exiting. Thus, **r to iswhite** places the point right after the last whitespace character.

Example

```
WSWordFwd :
   if isnl c else to isgray
   past iswhite
```

The **to isgray** command in this example moves the point forward until the character is a whitespace character or a newline character.

See Also

isgray, isnl, isend, issent, istoken, iswhite, toend, toeol, toruler, tosol

toend

Syntax

```
toend
```

Returns

True

Function

If the **direction** is forward, **toend** goes to the end of the file. If **direction** is reverse, **toend** goes to the start of the file. Always returns True.

Example

```
CorrectFile : mark (r toend CorrectRest)
```

The **r toend** command in this example moves the point to the beginning of the file.

See Also **toeol, tosol**

toeol

Syntax

```
toeol
```

Returns T/F

Function Moves to the end of the current line. The **toeol** command ignores the current direction and always goes forward.

Example

```
Down :
   (if action (tosol set themark) $)
   repeat (toeol c)
   if !action (dcolumn -> dcolumn)
```

The **toeol** command in this example moves the point to the end of the line so that the down action can be repeated correctly.

See Also **toend, tosol**

togmark

Syntax

```
to gmark
togmark
```

Function Moves the point to the global mark (that is, the one on the top of the stack if no others are pushed). Usually this mark is the other end of the selected region.

Example

```
CopyRegion :
   if select {
      FixRegionNoMod
      copy togmark
      Unselect
      1 -> AppendNext
      }
```

The **togmark** command in this example defines the end of the region to be copied.

See Also **after (mark), before (mark), to (mark)**

to (mark)

Syntax

```
to markN
```

Function

Moves the point to the specified mark. For more details on what **mark** can be, see **set (mark)** in this chapter.

Example

```
DeleteToChar :
   status "Delete to: "
   mark {
      if (CharFind = 1) delete to themark
      }
```

See Also

togmark

to Q0-QP

Syntax

```
to Qn
```

Returns

T/F

Function

Moves the point to the specified Q register. You can also use the **qswitch** macro to do the same thing.

Example

```
CopyFile :
   message "File to copy: " set Q0
   set Q1 Q0
   set Q0 flist Q1
   if !length Q0 {         ; if spec'd file mask had no matches
      set QD Q1
      mark { to QD "No files match '" toend "'." }
      error QD
      }
   set Q1 "" message "Copy "
      message Q0 message " to: " set Q1
   status "\nCopying..."
   fcopy Q0 Q1
   message "\nCopy complete."
```

The **to** *Qn* command in this example moves the point to the Q register containing the specified file name and allows editing within the Q register. The enclosing mark will force the point back to the file buffer when done.

See Also

qswitch

toruler

Syntax **toruler**

Returns T/F

Function If the **direction** is forward, **toruler** moves the point to the ^K at the start of the next ruler line. If the **direction** is reverse, **toruler** goes to the ^K at the start of the current ruler line. This is much faster than searching because it uses the cached ruler marks. The **toruler** command returns False if the specified ruler line does not exist.

Example

```
SetLeftIndent :
   mark {
      r toruler -> x
      set QD field "leftindent"
      message "\nLeft indent: " set QD
      if !x DefaultRuler
      r toruler set field "leftindent" QD
      }
```

The **toruler** command in this example moves the point back to the last ruler so that the left indent can be changed.

tosol

Syntax **tosol**

Returns T/F

Function Moves to the start of the current line, regardless of the current direction.

Example

```
Down :
   (if action (tosol set themark) $)
   repeat (toeol c)
   if !action (dcolumn -> dcolumn)
```

The **tosol** command in this example moves the point to the start of the current line if the point is currently in an action region.

See Also **toeol**

true

Syntax `true`

Returns True

Function Returns True. You can use this to directly set a Boolean variable to True.

See Also false

undelete

Syntax **undelete**

Function Inserts the contents of the Clipboard into the buffer at the current point and leaves the point after the inserted text.

Example

```
UndeleteN :
   ...
   repeat undelete
   ...
```

The **undelete** command in this example inserts the text from the Clipboard.

See Also **copy, delete, erase**

version

Syntax **version**

Returns #

Function Returns the version number of the editor.

Example

```
InitScreen :
   stopped {
      statline -> x
      0 -> statline
      mark {
         to QD clear
         version put "%d" toend r (c c)
         if (version < 100) "0." else "."
         }
      open "" draw
      ...
```

wait

Syntax

N **wait**

Returns

#

Function

Waits *N* milliseconds, or until the user presses a key. Returns 0 if no key was pressed, or the number of milliseconds remaining if a key was pressed.

If the amount of time specified by **swapdelay** passes, **wait** starts writing swap file pages to disk for crash protection.

If *N* is zero, **wait** will not return until a key is pressed. After the **swapdelay** passes, and the swap file is updated, **wait** does **pushback** *GetKey,* which will call MS-DOS for a keystroke and thus allow a context switch in multitasking programs.

Example

```
HiLiteFound :
   mark (found 1 -> select draw 0 wait Unselect swap themark)
```

See Also

delay, menudelay, swapdelay

while

Syntax

while *Boolean command*

Function

Executes *Boolean,* and if True, executes *command,* then executes **Boolean** again repeatedly until either *Boolean* is False, or the loop is broken.

Example

```
Restart :
   NormalMode
   if (exist "log.$$$") {
      draw pageread "log.$$$"
      fdelete "log.$$$"
      while keypressed key                ; eliminate typeahead
      }
```

See Also

do...while

windows (Variable)

Syntax

windows

Function

The number of windows on the screen. You can set this anywhere from 1 to 6. If you change it to a larger number, the current window is split in half enough times to make that many windows total, and the current setting of **direction** controls which window is current after each split. If **direction** is forward, the lower window is current; if **direction** is reverse, the upper window is current.

If you change **windows** to a smaller number, the current window is merged with a neighbor window enough times to make that many windows. The **direction** controls this as well; if forward, the window above the current one is merged, and, if reverse, the window below is merged. This is so **++windows, –windows** will leave the display unchanged.

Example

```
WindowDown :
   windows repeat {
      winswitch $
      if !dline Down
      scroll
      }
```

See Also

wlines, wtop

winswitch

Syntax

winswitch

Returns

#

Function

If the direction is forward, **winswitch** goes to the next window down (or to the top window if you are already in the bottom one). If the direction is reverse, **winswitch** goes up. Returns the number of the current window (top one is zero).

Example

```
WindowDown :
   windows repeat {
      winswitch $
      if !dline Down
      scroll
```

```
            }
```

See Also **windows**

wlines (Variable)

Syntax `wlines`

Function The number of lines in the current window. Assigning this variable changes the size of the current window.

Example

```
WindowUp :
   windows repeat {
      winswitch $
      if (dline = wlines - 1) Up
      r scroll
      }
```

See Also **windows**

write

Syntax `write "filename"`

Function Changes **fname** to the specified *filename* and then writes the entire contents of the buffer to that file on disk.

Example

```
WriteFile :
   set Q0 cdstrip fname
   message "Write file as:" set Q0
   0 SetSPRext
   if (!(exist Q0) || ask "Overwrite existing file?") {
      bufnum -> x
      if (buffind Q0 && (bufnum != x))
         close                  ;throw away copy if in buffer,
                                         : and not current file
      if !(stopped (write Q0)) {
         set fname Q0
                   ;change file name only if write successful
         }
      }
```

See Also **fname**

writeregion

Syntax

writeregion *region "filename"*

Function

Writes the area covered by *region* to the given file name. Unlike **write**, this command does not change ***fname***.

Example

```
WriteSelected :
   if (select && !ColMode) {
      set Q0 ""
      message "Name of file to write block to:" set Q0
      if (!(exist Q0) || ask "Overwrite existing file?") {
         writeregion togmark Q0
         mark {
            if (buffind Q0) ReReadFile
                                    ; if file was open, reread
            }
         Unselect
         }
      }
   else (ColMode ? message
         "Columns cannot be written" : NoBlock)
```

See Also

write

writeruler

Syntax

writeruler

Function

Deletes the current ruler line and inserts it again using the cached data. When you use macros to change ruler information (as with the macro **linelength**), you only change the editor's internal cached copy. You must then execute **writeruler** to update the file to match the cache. You must do this before moving the point a large distance because the current ruler might get thrown out of the cache and thus lose the saved data.

Example

```
ToggleJustify :
   InsertFirstRuler
   justify ? 0  -> justify : 1 ->justify
   writeruler
```

See Also

insertruler, rulermod, readruler

wtop (Variable)

Syntax `wtop`

Function Shows where the window starts in number of lines down from the top of the screen. The variable cannot be assigned.

See Also **windows, wlines**

zoom (Variable)

Syntax `zoom`

Function If this flag is set to True, the current window will be expanded to take up the entire screen.

Example

```
WindowClose :
   0 -> zoom
   if windows (--windows)
```

See Also **windows, winswitch, wlines**

PART

3

Appendixes

APPENDIX

A

Commands Defined in STANDARD.FMT

Table A.1 lists and briefly explains each of the commands defined in STANDARD.FMT. You can change any of these commands using the techniques described in Chapter 3.

Table A.1: Commands Defined in STANDARD.FMT

Command	Description
Format Regions of Text	
Address	Left-justifies the text halfway across the page.
Asterisks	Places an asterisk (* or, on a PostScript printer, ♦) before each new paragraph. If these commands are nested, the second level has bullets; the third, hyphens.
Bullets	Places a bullet (•) before each new paragraph. If your printer cannot print true bullets, it uses lowercase *o*'s instead. If these commands are nested, the second level has hyphens; the third, asterisks.
Center	Centers the specified text between the current left and right margins.
Closing	Same as Address; left-justifies the text halfway across the page. Used mainly for the closing of letters typed in a modified-block style.
Column	Sets up parallel (not snaking) columns. The text following the command starts printing at exactly the same spot as the text governed by Column.
Description	Outdents text followed by a tab by one-quarter line. This table is an example of the Description command.
Display	Moves the left margin in (to the right) by one-half inch. You must press *Enter* to end every line; onscreen worwrapping is not kept.
Example	Moves the left margin in (to the right) by one-half inch, and prints the text in a typewriter (fixed-width) font. Onscreen wordwrapping is not kept; you must press *Enter* to end every line.
FlushLeft	Formats the selected text against the left margin.
FlushRight	Formats the selected text against the right margin.
Hyphens	Places a hyphen (-) before each new paragraph. If these commands are nested, the next level has an asterisk (*) mark; the third has bullets (•).
Multilevel	Numbers each paragraph like the Numbered command, but if these commands are nested, the inner levels are numbered as 1.1, 1.2, 1.3, 1.3.1, and so on.
Numbered	Numbers each new paragraph. If you nest these commands, the top level is numbered 1,2,3,..., the next level is a,b,c,..., the next is i,ii,iii,..., and then the cycle starts again with 1,2,3....

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	Description
Outline	Placed an uppercase Roman numeral before each paragraph. Nested commands use uppercase letters, then Arabic numbers, then lowercase letters, then lowercase Roman numerals. The sequence is like this: I, A, 1, a, i.
Quotation	Moves the left and right margins in (toward the center of the page) by one-half inch and single-spaces the marked text. Onscreen wordwrapping is not kept; you must press *Enter* to end every line.
Text	By itself, Text doesn't do anything. It is typically used with formatting parameters to create special effects. For example, the command Text, *columns=2* formats marked text in two columns. The command Text, *font Times* prints the marked text in a Times font.
Undent	Outdents the first line of each paragraph by one-half inch. The first line appears one-half inch to the left of all remaining lines in the paragraph.
Verbatim	Prints the text exactly as entered; Verbatim does not change margins, indent text, or wordwrap lines.
Page Headings and Footings	
PageHead *text*	Prints the specified text at the top of the current page only.
PageFoot *text*	Prints the specified text at the bottom of the current page only.
Header *text*	Prints the specified text at the top of every page except the first.
HeaderEven *text*	Prints the text at the top of every even-numbered page.
HeaderOdd *text*	Prints the text at the top of every odd-numbered page.
HeaderT *text*	Prints the text at the top of the first page (title page) only.
Footer *text*	Prints the specified text at the bottom of every page except the first.
FooterEven *text*	Prints the text at the bottom of every even-numbered page.
FooterOdd *text*	Prints the text at the bottom of every odd-numbered page.

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	Description
FooterT *text*	Prints the text at the bottom of the first page (title page) only.
Document Organization	
Chapter *title*	Starts a new chapter (begins a new page, prints a big, centered, sequentially numbered major heading, and creates an entry in the table of contents).
Section *title*	Starts a new section (prints a big, left-justified, sequentially numbered subheading, and creates an entry in the table of contents).
Subsection *title*	Starts a new subsection (makes a bold, left-justified, sequentially numbered subheading, and creates an entry in the table of contents).
Paragraph *title*	Starts a new paragraph (same format as Subsection, but the numbering is one level lower).
Appendix *title*	Starts an appendix. This command is just like Chapter, but the appendixes are numbered with capital letters.
AppendixSection *title*	Starts a new section in an appendix. This command is just like Section.
Headings/Document Divisions	
HeadingA *text*	Prints the specified text as a large, centered title. If you are printing two-column text, this heading is centered above both columns.
HeadingB *text*	Prints the specified text as a large, centered title. It's similar to HeadingA, but it's a bit smaller and, if you're printing two-column text, this heading is centered above one column.
HeadingC *text*	Prints the specified text as a large, left-justified title.
HeadingD *text*	Prints the specified text in bold type, justified at the left margin.
TOC *text*	Prints the specified text in the table of contents. Be sure to insert an * (asterisk) formatter command after the TOC command; otherwise, the next entry in the table of contents will print on the same line.
TOF *text*	Prints the specified text in the table of figures.
TOT *text*	Prints the specified text in the table of tables.

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	**Description**
Figures and Tables	
Figure *text*	A format for figures. Similar to Verbatim except that Figure keeps the marked text together on a page.
Caption *text*	Sequentially numbers a figure, and lets you specify a title for the figure. The figure number and caption print in the List of Figures in the table of contents.
FCapt *text*	Like Caption, but no entry is generated in the table of contents.
Table *text*	A format for tables. Similar to Verbatim except that Table keeps the marked text together on a page (unless the text exceeds the length of the page).
TCaption *text*	Sequentially numbers a table, and lets you specify a title for the table. When printed, the table caption is centered between the left and right margins. The table number and caption print in the List of Tables in the table of contents.
TCapt *text*	Like TCaption, but no entry is generated in the table of contents.
	Note: When using Figure and Table be sure to place tags *after* the Caption, TCaption, or TCapt command to avoid reference discrepancies.
Footnotes and Endnotes	
ENote *text*	Prints the specified text in the endnotes of the document, and prints a small, superscripted reference number in the text.
FNote *text*	Prints the specified text as a footnote (at the bottom of the page), and prints a small, superscripted reference number in the text.
NoteChapter	Prints the number and title of the current chapter at the beginning of the endnotes.
NoteSection	Prints the title and number of the current section in the endnotes.
SNote *text*	Prints the specified text as a footnote (at the bottom of the page), and prints a superscripted star (asterisk) as the reference marker.

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	Description
Typeface Commands	
B *text*	Prints the marked text in **bold** or **overstrike**. This is the same as the ^B open delimiter Sprint inserts when you choose **B**old from the **T**ypestyle menu.
Large *text*	Prints the marked text in a large, bold, or double-width font. This command is used when printing all Heading commands. This is the same as the ^A open delimiter Sprint inserts when you choose **T**ypestyle/ **L**arge.
E *text*	Prints the marked text in *italics* or underline (depending on what your printer is capable of doing). This is the same as the ^E open delimiter Sprint inserts when you choose **I**talic from the **T**ypestyle menu. **Note:** Although the formatter will only underline the words and not the spaces, the editor (and screen output) will display a solid underline.
I *text*	Prints the specified text in *italics*. Same as *E <text>*.
Q *text*	Prints the text as a $_{\text{subscript}}$. If possible, a smaller font size is used. This is the same as the ^Q open delimiter Sprint inserts when you choose **–** Subscript from the **T**ypestyle menu.
S *text*	Prints the text as a $^{\text{superscript}}$. If possible, a smaller font is used. This is the same as the ^S open delimiter Sprint inserts when you choose **+** Superscript from the **T**ypestyle menu.
T *text*	Prints the marked text in a fixed-width `typewriter` font. If both "elite" and "pica" are available, the smaller "elite" font is used. This is the same as the ^T open delimiter Sprint inserts when you choose **T**ypewriter from the **T**ypestyle/**F**ont menu.
U *text*	Underline all nonblank characters, including punctuation. This is the same as the ^W that Sprint inserts when you choose **w**ord Underline from the **T**ypestyle menu.
UN *text*	Underlines only the alphanumeric characters. Blanks and punctuation marks are not underlined.
UX *text*	Underlines everything within the marked region. This is the same as the ^U open delimiter Sprint inserts when you choose **U**nderline from the **T**ype style menu.

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	Description
X *text*	~~Strikeout~~ the text with a solid line. This is the same as the ^X open delimiter Sprint inserts when you choose **O**verstrike from the **T**ypestyle menu.
– *text*	Same as *Q <text>*.
+ *text*	Same as *S <text>*.
Indexing	
D *text*	Prints the marked text, and also places it in the index. (This is the *only* index command that prints the text both in the text and in the index.) This is the same as the ^D open delimiter Sprint inserts when you choose **W**ord from the **S**tyle/**I**ndex menu.
IXRef *item,item,...*	Adds the specified item into the index with a reference to the current page. Commas represent multilevel entries. This is the same as choosing **R**eference Word from the **S**tyle/**I**ndex menu.
IXMaster *item*	Adds the specified item into the index, references the current page, and prints the page number in bold. This is the same as choosing **M**aster key word from the **S**tyle/**I**ndex menu.
IXRange	Adds the specified item into the index and prints the range of pages as defined from a user-defined tag. This is the same as choosing **P**age Range from the **S**tyle/**I**ndex menu.
IXSee	Creates a *see* reference in the index. This is the same as choosing **I**ndex/**S**ee.
IXSeeAlso	Creates a *see also* reference in the index. This is the same as choosing **I**ndex/**A**lso See.
Cross-Referencing/Variables	
Incr *variable*	Increments the variable, and sets *SectionNumber* to the increment value (so the formatter command Label will use it).
Label *tagname*	Sets a tag equal to *SectionNumber*.
Title *variable*	References the title of a built-in variable.
V *variable*	References the specified variable. This is the same as the ^V open delimiter Sprint inserts when you choose Insert/Variable command.

Table A.1: Commands Defined in STANDARD.FMT, continued

Command	Description
Print-Related Commands	
EndF	Reverts to the font that was current before the last Font command.
EndS	Reverts to the font size that was current before the last Size command.
Font *name*	Changes the current font to the named font.
Kern *dimension*	Moves the print head back (to the left) by the specified amount.
Size *dimension*	Changes the current font size to the specified dimension.
Miscellaneous	
Keep Text Together	
NeedSpace *dimension*	Starts a new page if the specified amount of space is not left on the current page.

APPENDIX

B

Built-In Format Commands

Some formatting commands are built-in to the Sprint program, and can't be modified. Table B.1 lists these built-in commands. Most of the commands in this table can be used (when appropriate) with the Style/Other Format menu command. You can also use them with @-sign commands.

Table B.1: Built-In Formatting Commands

Command	Description
|	Same as a soft hyphen. Specifies a conditional hyphenation point; if necessary, a word can be broken at this point, and a hyphen inserted.
_	Inserts a single hard space between two words.
@@	Prints the @ sign.
!	Allows a line break in the middle of a word, but does not print a hyphen (for example, this/!that tells the formatter it can break *this/that* after the slash). The **Hyphenation** utility uses this command to insert discretionary hyphens in a file.
[	Sets the left margin and the indent dimension to the column containing the [command. This commands acts as if there were a ruler here that simply changed the left margin.
]	Sets the right margin to the column containing the] command.
$	Sets the left margin to the column containing the $ command, but does not change the first-line indent. This command affects only the current paragraph.
' (close quote)	Ignores all the whitespace (that is, spaces, tabs, and blank lines) after this command, as well as the line break that appears immediately after the command.
' (open quote)	Deletes all the preceding whitespace on the current line. The next word will then appear right next to the previous one. Usually, this command is used in conjunction with the ' command.

Command	Description
*	A forced new line. This acts exactly like a regular hard return, except the *Fill* parameter cannot treat it as a space. Any whitespace or new lines after an * command are ignored, so an * command at the end of a line won't unexpectedly act like two new lines. The * command is most often used to show where new lines should be in a macro definition, and to mark the ends of lines that shouldn't be wordwrapped when the *Fill* parameter is On.
, (comma)	Two of these commands formatted next to each other will print a comma. Used by macros such as *foot*.
/	Moves to the next formatter tab stop. This serves the same function as the *Tab* key but only takes effect when you print the file.
\	Same as /.
^	Sets a formatter tab stop at the current column position. This command is used in conjunction with the / command, which moves the formatter to the tab stop set by a ^ command. It does not affect the ^I tabs in the input file.
;	Prints nothing. This command is useful to prevent ~ or another command from removing whitespace. You can also place this command between characters to prevent them from being recognized as a TCT sequence.
<	Starts a new line that prints on top of the current line (similar to Strikethough, except that you can specify the character(s) used to overprint).
~ (tilde)	Ignores any whitespace (including hard returns).
> *text*	Wide break; that is, forces text following the command to the current right margin. This is the same as the Insert/Wide Space (Spring) command. The (*text*) argument is optional. If an argument is given, it is replicated to fill the space the = takes up. In that case, it operates the same as the Repeating Character command.

Table B.1: Built-In Formatting Commands, continued

Command	Description
= *text*	Acts like Insert/Wide Space (Spring), but only expands half as much, centering the text after it.
AtEnd *command, /c DOS command*	Gives Sprint a DOS command to carry out after completing its formatting of a document.
Begin *command, arguments*	Starts a command or a specific format. Similar to @<*command*> except the arguments can be used to modify the command. For example, Begin Description, *indent 5 spaces* will begin the description format and indent all new paragraphs five spaces.
Case *variable, value "text", value "text",...*	Executes a case statement on string variables.
CenterPage *dimension*	Centers all the text on the current page vertically around the given position. For example, using the command Centerpage .5 page at the start of a short letter is often easier than fiddling with the leading blank lines to center the letter on the page. The Title Page command uses *CenterPage*.
Char #	Formats the specified character with the given ASCII code. TCT (character translation) is not done on this character. Usually this is placed in TCT entries, in macros, or in conjunction with the <*fontname*> command.
ColumnBreak	Inserts a mandatory break in the column. The text following this command will start at the top of the next column.
Comment *text*	Ignores everything within the comment delimiters, and the new line that follows the end of the comment text. Comments cannot be nested within each other .The formatter will end the comment field at the first matching close delimiter. You may want to use BEGIN and END COMMENT commands when commenting out a large area of text. This is the same as Typestyle/ Hidden.

Table B.1: Built-In Formatting Commands, continued

Command	Description
Default *text*	Prints the specified text in the default font, without any additional attributes (such as bold, underlining, and so on). This is easiest way to turn off enclosing attributes.
Define *new command* *{=existing command},* *attributes/parameters...*	Defines a new command to affect the format of a region of text.
End *command, arguments*	Closes a command started with BEGIN, and provides some arguments that can affect the closing. The arguments are currently ignored, but allowed for future enhancements.
Error *text*	Prints the given text (which can contain Value commands) as an error message. The error message will also include the input file name, and line number on which the error occurred.
Escape *h = dimension,* *d = dimension,* *w = dimension,* *s = "string"* *or f = "filename"*	Sends raw data to the printer (using the printer's command language, not Sprint's).
Eval *expression, template "string"*	Immediately expands the specified expression. See the entry on the Value command for the difference between Eval and Value.
<fontname> *text*	Prints the specified text in *fontname* font. The *<fontname>* is a name of any font or attribute in the printer description that does not have a dot in it. This is the same as the **Typestyle/Font** command.
Format *file.fmt*	Specifies a file other than STANDARD.FMT to be automatically included at the start of the formatting process. This is the same as **Layout/Document-Wide/Style Sheet**.

Table B.1: Built-In Formatting Commands, continued

Command	Description
Group *text*	Does not allow a page break to separate the specified text (unless a PgBreak command is entered somewhere in the specified text). This is the same as Layout/Page Breaks/Group Together on Page.
Havespace *distance {,y "text"} {,n,"text"}*	If there is *distance* space left on the page, then executes the *y* text. If there is not enough space left on the page, executes the *n* text.
Hsp *distance*	Moves *distance* horizontally from the last character printer. A negative number moves left; the maximum leftward movement is to the beginning of the current word.
HUnits *number*	Moves the print head the specified number of units to the right (if the number is positive) or to the left (if the number is negative).
If *expression, {,{y}"text"} {,{n | else} "text"}*	If the specified expression is not equal to zero, executes the *y* part of the text. Otherwise, if the expression is equal to zero, executes the *n* part (which can also be written as the *else* part) of the command line. This command is also used to check whether an argument was passed to a multi-argument macro.
IfDef *name {,{y} "text"} {",n | else} "text"}*	If the *name* command exists, executes the *y* part of the command. If the *name* variable does not exist, executes the *n* part (which can also be written as the *else* part) of the command.
Include *file*	During formatting, inserts the contents of the specified file here. If the desired file is in a different directory than the input file, you must also specify the appropriate directory path with the file name. For example, `Include \dir1\dir2\filename.doc`. The new line after the Include command is disregarded by the formatter.
KeepFollowing	Prevents a page break between this line and the next line. This is the same as Layout/Page Breaks/Keep with Following Text.

Command	Description
Message *text*	Prints the specified message on the screen during formatting.
Modify *command, fields...*	Modifies a previously defined format as specified. You cannot modify an format after it has been used.
NewColumn	Same as ColumnBreak.
NewPage	Begins a new page. This is the same as **Layout/Page Breaks/Insert** (unconditional).
NoTCT *text*	Formats the enclosed text with all TCT (character translation) commands temporarily disabled.
O *text*	Overprints all the enclosed letters (a maximum of 12 letters is allowed). You can use a superscript command and other commands here to move accents to print in the right place. To print good-looking accent marks, always put the accent mark first.
OVP *text*	Formats the text, then backs up and continues formatting right on top of it.
PageRef *expression* {*,template "string"*}	Prints the page number of the page where the *expression* variable was set using the **D**efine a Tag command. This is the same as choosing **R**eference a Tag from the **S**tyle/**X**-Reference menu, and then choosing **R**eference By/**P**age Number.
Parent *var1=var2*	Makes *var1* the parent of *var2*. *Var2*'s parent remains unchanged. You can do this only once for each *var1*.
PgBlank *number*	The next time the formatter begins a new page, it will print *number* blank pages. (To have this command executed immediately, insert an unconditional page break command right before it.) Note that a blank page is not immediately printed, but the formatter waits until the end of the page it is currently formatting. If the number is not included, one blank page is printed.

Command	Description
PgBreak	A page break can appear here. Has priority over all possible ways the formatter might prevent a page break (for example, group), except for the KeepFollowing command. This is the same as Layout/Page Breaks/ Conditional Page Break.
Place *command*	Prints an "after" format (specifically Endnotes) here.
Printer *name*	Prints on the printer <*name*> (defined in a file called <*NAME*>.SPP). The file <*NAME*>.SPP must be on the disk. This is the same as choosing Print/Current Printer and then choosing the desired printer name.
ReadEPS *filename*	Checks *filename* to see if it's a true encapsulated PostScript file.
Ref *expression* {*,template "string"*}	Prints the value of *expression* that is assigned by the Define a Tag command somewhere in the text. The main difference between this command and the Value command is that it doesn't matter if a variable is set yet or not. This is the same as the Reference a Tag command on the Style/X-Reference menu.
Reserve *dimension*	Leaves the specified amount of space blank. For example, `RESERVE 3 inches` leaves 3 inches of blank space immediately following the command. If the blank space won't fit on the current page, a new page is started, and the entire blank space is put at the top of the new page.
Ruler *text*	Does a ruler line. The input parser converts a ruler line ^K into Ruler(*text*) where *"text"* is the text on the ruler line. Ruler is exactly the same as Style except that is clears the "ruler" tab stops.
Set *variable* = *expression*	Assigns a new value to a variable. The variable's value can be changed throughout the document.
String *variable* = *string*	Assigns a string of values to the variable. This is the same as the Define Text Variable command on the Insert menu.

Table B.1: Built-In Formatting Commands, continued

Command	Description
StringInput {*"message",*} *variable*	Asks the user to input a string from the terminal.
Style *fields...*	Changes the global formatting specifications, and makes changes to the current format. You can put as many Style commands throughout your document as is necessary. However, to avoid formatting problems during the two different printing passes, you should set global style specifications only once.
Tab *dimension*	Acts just like pressing *Tab*, except that it moves to the given horizontal location from the left margin.
TabDivide *dimension*	Sets *n*-1 formatter tab stops evenly spaced across the line.
TabSet *dimension*	Sets a formatter tab stop at the given horizontal position.
Tag *name* {*=expression*}	Sets a tag for the variable *name*. You can then use the Ref *tagname* command to print out the expression. If the expression part of the command is missing, then the current format's value will be assigned to *name*. For example, if you placed the command Tag *test* in section 2.1 of your document, Ref *test* would have a value of 2. 1 This is the same as the Style/X-Reference/Define a Tag command.
TagString *name* = *"string"*	Sets a tag to a string value (in quotes).
TCT *"string1"* = *"string2"*	Changes the value of *string1* to the value of *string2; string2*'s value remains unchanged.
Template *variable* = *"string"*	Permanently sets how a variable prints. You can execute this command only once for each variable.

Table B.1: Built-In Formatting Commands, continued

Command	Description
Under *text*	The main use of this command is to allow an index entry to appear somewhere other than its normal alphabetical location within the index. For example, you could have the index print *20* where *twenty* would normally appear (numbers are ordinarily indexed together at the beginning of the index). This command must be used within a regular Index command. This is the same as Style/Index/Index Under.
Value *expression* {,*template "string"*}	Prints the value of the specified expression. The difference between this command and the Eval command is that Eval is done immediately when encountered (except in a format), while Value is not done until the text it is in is being formatted. Value is the same as the Insert/Variable command.
VUnits *number*	Moves the print head the specified number of units down (if the number is positive) or up (if the number is negative).
Word *text*	Doesn't allow any line breaks in this text. The whole text should be regarded as a single "word."

APPENDIX

C

Style Sheet Commands

Sprint has a number of commands that are meant to be used exclusively (or at least primarily) when modifying style sheets (that is, files with the .FMT extension that contain command definitions).

These .FMT-specific commands are therefore a special breed of Sprint formats since you virtually never need them unless you are creating your own format (.FMT) files. And because format files normally use @-sign commands (and not commands accessed through the menus), you have to type in the commands using the @-sign method.

For a full list of Sprint's more frequently used commands, see the *Reference Guide.*

AtEnd

Keystrokes @AtEnd[Command /c *DOS commands*]

Function Gives Sprint a DOS command to carry out after completing its formatting of a document.

You can have your .FMT file always carry out a particular DOS command after every formatting cycle (but before actual printing) by including this command at the end of it.

This would be handy if, for example, you needed to run a particular spooling program before printing. In that case you could add the command

```
@AtEnd[spool @Value(Manuscript)]
```

If you omit the "/c", Sprint will wait at the DOS shell until you type `exit` to return to Sprint.

Some DOS commands need to be preceded with the words *Command /c* in order to work. This is because certain DOS commands actually get interpreted by the COMMAND.COM program. As a result, you have to call COMMAND before running DOS standbys such as DIR, DEL, ERASE, COPY, and the like.

If you don't want a particular DOS command run after every formatting cycle, you can include the command at the end of any document for which you want the command carried out.

Define

Keystrokes

@Define[*FormatName, attribute1, attribute2,...*]

@Define[*NewFormatName=OldFormatName, attribute1, attribute2,...*]

Function

Creates a new format.

This powerful command lets you create new formats. After you create a new format command with the Define command, you can use it just like any of the predefined format commands.

New (format) commands can be modeled after existing commands. You can make the new command equal to an existing one, and then list the attributes that make it different from the model. See the "Modifying Formats" entry in Chapter 1 of the Sprint *Reference Guide* for the "Format Parameters" table that lists all valid parameters you can include in a command definition.

For a detailed list of parameters that can be included in a command definition, and for examples of several customized commands, refer to Appendix D.

How To

To define a command, give it a name and a set of attributes. If the new command has several attributes in common with an existing command, include an equal sign followed by the name of the other command.

The following example creates a format called *RightColumn*. This format is defined such that it prints double-spaced text on only the right half of a page; since it has a lot in common with the Address format, we included this format in the definition, and added a few other attributes.

You should insert the Define command in a copy of the STANDARD.FMT file (or some other .FMT file that you've created). Note that you *could* insert the command only at the top of the documents that need it, thereby guarding against accidentally introducing an error into STANDARD.FMT, which must remain error-free when printing and formatting. In either case, we recommend you use the @-sign method of entering the Define command.

Type

```
@Define[RightColumn=Address, spacing 2, justify yes, above 0,
below 0]
```

Now save your .FMT file that has this new definition.

When you want text to print in this format, make sure your document is set up to recognize your special .FMT file (if you created one) by choosing **L**ayout/**D**ocument-Wide/**S**tyle Sheet. Then choose **O**ther Formats from the **S**tyle menu and type `RightColumn`. Here's how RightColumn will look when printed:

> This is an example of *RightColumn*. As you can
>
> see, the text begins to the right of the current
>
> left margin and is aligned at the right margin.
>
> The text is also double-spaced.

Global Change
First decide whether you want to permanently change a format definition. If you think you might want to use a format as it's defined in STANDARD.FMT, we recommend that you create a new command, rather than modify the existing definition. (See the Define entry in this menu encyclopedia for details.)

If you want to permanently change a format definition, you can edit the STANDARD.FMT line that contains the format definition (for example, @Define(Bullets,...)) and make the desired changes. Be careful with this method, since you are making a change that will affect *all* files containing this format command.

A safer way to modify a format in STANDARD.FMT is to:

1. Copy the format definition within STANDARD.FMT (mark the current definition and then copy it).
2. Move the cursor to the line below the current format definition, and paste the definition (with the **Paste** command).
3. Change @Define to @Modify.
4. Add to, change, or delete the format attributes to create the effect you want.
5. Save the STANDARD.FMT file before printing a document.

For example,

```
@Define(Hyphens=Numbered, numbered "%<-%;*%;o%]",
        BeforeEach "@>()@value(counter)@hsp(.2in)@\")
```

```
@Modify(Hyphens,numbered "%<x%;*%]")
```

This example modifies the Itemize format so that it prints a lowercase *x* instead of a hyphen at the beginning of each new paragraph. We only had to change @Define to @Modify, change the numbering template, and remove the parameters following the template.

(**Note:** If you're using a PostScript printer, you'll have to modify the printing template for the Hyphens format in POSTSCR.TCT instead.)

When you modify a format, any attributes that you omit are left unchanged. The formatter recognizes everything contained in the format definition, except the parameters that you override with the Modify command. For example, if you type the command:

```
MODIFY Numbered, indent -5
```

you are only changing the indent value (the point at which each new paragraph begins within an Numbered format). The rest of the parameters contained in its definition (the line that begins *@Define(Numbered,...)*) remain in effect.

Tips

Commands and formats are typically defined in .FMT files (like the STANDARD.FMT file). You can create your own specialized .FMT files by copying all of STANDARD.FMT into a new file, modifying it, and giving it a new name. You then tell Sprint to format using this .FMT file by using the Style Sheet command.

If the command is only going to affect one document, you can insert it near the top of this document. Make sure you define the new format before you actually use it.

For a complete detailed list of parameters that can be included in a command definition, refer to the Sprint *Reference Guide* in the "Menu Encyclopedia" entry called Modifying Formats. Note that several parameters are valid *only* when used with Define. These are noted in the list. For an extended example of several customized commands, refer to Chapter 4.

Error

Keystrokes

@Error[*error message text*]

Function

Reports an error in user-defined formatter macros.

The Error command lets you generate error messages in your formatter macros. The command produces an error message just like the built-in Sprint error messages, complete with the file name and line number containing the error. As with all error messages, the formatter will abort the Print/Printer command after the first pass.

If you create a macro that requires the user to do something (for example, assign a value to a string), you can include the *Error* command as part of your macro. For example,

```
@IfDef[name,N "@Error<You must do @string(name =
"BookTitle")>"]
```

If the user forgets to do @string[name= "The manual"], whatever command actually uses the string can test if it's been set, and force a more friendly, informative message.

The following *NameInFooter* macro will cause the formatter to put whatever string given as the BookTitle in the footer of a document. Since the macro requires a string value for the variable *BookTitle,* the file will have to contain an Define Text Variable or @String command that defines the book title. If you try to use the macro without inserting the string (that is, BookTitle = "My diary"), the formatter will generate the informative error message *You must do @string(booktitle = "name").* Note the correct sequence of commands, @string, then @macro, then the new command itself:

```
@string[booktitle="My Diary"]

@macro(Nameinfooter = "@ifdef(booktitle,N "
@error(You must do @string(booktitle = "name"))",
Y"@string(pfe = booktitle)@string(pfo = booktitle)")")

@nameinfooter
```

Tips

If you want to generate a warning instead of an error, use the Warn command instead (which is also described in this chapter).

Eval

Keystrokes

@Eval

Function

Represents the value of a variable.

@Eval is the value of a Sprint-defined variable (such as *Page, Counter, Chapter,* and so on), or a variable that you create with the Define Text Variable or @String command.

For example, if you place @Eval(page) in the definition of a string, the number of the current page (that is, the page the @Eval command appears on) will become part of the string.

It is important to understand the difference between this command, which refers only to the *present* value of the variable, and @Value (which equals Insert/Variable from the menus), which refers to whatever value the variable later takes on.

The value of an Eval command is filled in immediately, but the value of a Value command is not filled in until the last possible moment before printing.

(For programmers, the difference between Eval and Value is that between "lexical" and "dynamic." Eval computes a value when it is first defined, while Value computes the value when it is used.)

The following illustrates the difference between @Eval and @Value:

```
@b(Passage One)@set(passage = 1)
Medieval philosophy buffs will recognize the
difference between "Eval" and "Value" as an example
of the distinction between @i<intentio prima>,
wherein a concept refers directly to an object, and
@i<intentio secunda>, wherein a concept refers to
another concept.

@String[PrimaRef = "I mentioned @i(intentio prima) in
passage @eval(passage), and you are now reading
passage @value(passage)."]

@b(Passage Forty-six)@set(passage = 46)
@value(PrimaRef). See?
```

results in:

> **Passage One**
> Medieval philosophy buffs will recognize the difference between "Eval" and "Value" as an example of the distinction between *intentio prima,* wherein a concept refers directly to an object, and *intentio secunda,* wherein a concept refers to another concept.
>
> **Passage Forty-six**
> I mentioned *intentio prima* in passage 1, and you are now reading passage 46. See?

Eval is also used to place macro arguments in the "macro expansion" (see the chapter on macros in this manual); you can see many examples of this in the file STANDARD.FMT. In this context, if you don't give a variable name to Eval, the default variable *text* is used. In macros, Eval is not given a value when the macro is defined, but when it is used in the text.

For example, the command Label is merely a Tag (Define a Tag) command evaluated for the current value of the variable *SectionNumber*:

```
@macro(label() = "@tag(@eval SectionNumber)")
```

Eval can also take two other parameters: a new template for printing the variable differently, and a default value to be used if the variable is not defined. For example:

```
@Eval(page,template = "%0")
```

See the Template entry in the *Sprint Reference Guide* for a list of all available templates.

HUnits

Keystrokes

@HUnits[*n*]

Function

Moves the print head *n* units to the right (if the number is positive) or to the left (if the number is negative).

This command uses printer units, which are the printer's primitive unit of measurement. For this reason, a unit on one printer might not be the same as a unit on another.

How To

Use this command to construct sophisticated (but printer-dependent) commands. STANDARD.FMT uses @HUnits in its special EPS commands (for use with PostScript printers).

Tips

HUnits is often used in conjunction with VUnits.

If

Keystrokes

@If[*value, thenClause, elseClause*]
@If[*value, yesClause, noClause*]

Function

Creates macros that have decision-making capabilities.

The If commands let you specify cases within your macros.

How To

You can use If commands throughout your macros for changing the way things print. For example, you could use the If command to control printing on different printers, depending on the value of a variable called *draft*. For example:

```
@If(draft=1, y "@printer[draft.spp]", n
"@printer[laser]")
```

If a file contains the formatter command *set draft=1,* Sprint will format the file for printing on the printer you installed with the name *draft*. If you set the variable *draft* to a value other than 1 or if you don't define it at all, Sprint will format the file for output to a printer called *laser*.

The If command is also useful for printing special page footings in your normal Sprint documents. For example:

```
BEGIN FOOTER
@if(draft)=1, "Draft Copy", else "Final Version"))
END FOOTER
```

would print *Draft Copy* in your footing if *draft* was set to 1, but would print *Final Version* if the value of *draft* was anything else.

There are also more complex uses for the If command. For example, suppose you have a database with a customer list, and this customer list contains a field called "credit" that contains the type of credit extended to the customer, encoded as follows:

```
1=COD 2=Net 10 3=Net 20 4=Net 30 5=Net 60
```

You can create a macro that returns a string when given the values 1-5. Using such a string, you could send each customer a customized version of a form letter specifying payment terms.

To accomplish this, you would enter the following macro:

```
@macro(terms="@If(credit=1, "COD", else @If(credit=2, "NET
10",
   else @If(credit=3, "NET 20", else @If(credit=4, "NET
30",
   else "NET 60"))))")
```

For each customer you would enter the following:

```
Your current credit with us is @terms.
```

By using such a macro, you can universally change the credit limits for each customer by adding a field or changing the text in the macro. When times get tough, changing the command to

```
@macro(terms="COD")
```

could save a lot of money.

Using If commands can get rather confusing when several are in a string. To avoid this confusion, you can use the Case command. Think of the Case command as a table of input, and the If command as an action, similar to the inital description of the encoding of the credit example.

Using Case instead of several If's, the macro becomes

```
@macro(terms="@case( credit, 1 "COD", 2 "NET 10", 3
"NET 20", 4 "NET 30", 5 "NET 60")"
```

IfDef

Keystrokes

@IfDef[*CommandName,* Y *'text or command(s)',* N *'text or commands(s)'*]

Function

This command instructs the formatter to make a decision based on whether a particular command has been defined up to that point

The command takes a "Y" (yes, the command is defined) and an "N" (no, the command isn't defined) case.

How To

Enter the @IfDef command wherever you need to have the formatter do one of several actions depending on the existence of a particular command. IfDef takes as parameters the name of the command being tested, and a Yes and No clause.

For example, the following bit of code forces documents to include the date as a text variable to be printed in the header of odd pages (that's what the *pho* variable does):

```
@IfDef(date, y "@String(pho=date)",
     n "@Error(You must define "Date" to print this file)")
```

IfOdd

Keystrokes

@IfOdd[*VariableName,* Y *'text or command(s)',* N *'text or commands(s)'*]

Function

Instructs the formatter to make a decision based on whether the value of a numeric variable (like *page*) is odd or even.

The command takes a "Y" (yes, the value is odd) and an "N" (no, the value isn't odd) case: If the value is odd (for example, the current page number is odd), Sprint automatically executes the "Y" case; otherwise, Sprint executes the "N" case.

How To

Use the IfOdd command when a formatter function should depend on whether a variable is odd or even. You ordinarily use the @-sign command version of this command, since you usually use the command in .FMT files.

You can, however, also use it in your normal Sprint documents. For example, let's say you want a table to begin on an odd page. If the text before the table ends on an odd page, you want the formatter to center the text *This page intentionally left blank* on the following page (an even-numbered page), and then start the table on the following (odd-numbered) page. You could use the following IfOdd command:

```
@IfOdd[page, N "@NewPage @CenterPage[.5 page]
@Center<This page intentionally left blank.>
@NewPage"]
```

Insert the IfOdd command before the

```
BEGIN TABLE
```

command in your file. When the formatter sees the IfOdd command, it looks at the value of the current page. If the page number is odd, it skips to the next page, centers the text *This page intentionally left blank* and then skips to the next (an odd-numbered) page and begins printing the table. If the text before the table ends on an even-numbered page, the formatter ignores the command.

Tips

If you want the formatter to print text based on the value of a *non-numeric* variable, use the Case command.

You will find the IfOdd command defined in the STANDARD.FMT file.

Macro

Keystrokes

@Macro[*Name="string"*]

Function

Creates a Sprint formatter macro.

This command lets you create a macro that the formatter will execute. It is often used as a substitution command, when you want to type a short string of text, and have the formatter substitute a longer string during printing. In other words, a macro can let you combine several variables into one string and assign a single word to recall that string.

How To

When you have repetitive text in one or more documents, use a Macro command. For example,

```
@Macro[ShortString="Longer, Replacement String"]
```

The text entered in your @Macro command may be as simple as an abbreviation; for example,

```
@Macro[Sprint = "Sprint: The Professional Word Processor"]
```

This makes *Sprint* a formatter command that you can enter with the Style/Other Formats command. When the formatter sees the *Sprint* command during formatting, it will substitute the command text *Sprint* with the longer string *Sprint: The Professional Word Processor.* In this example, the Macro command lets you type the short string *Sprint,* and instructs the formatter to print the longer string.

You can also build on existing macros to create more complex ones. For example, you could create a macro to print your company name in several places throughout your document:

```
@macro(BI = "Borland International, Inc.")
```

You could then build on this macro, taking advantage of the "BI" definition by adding your address:

```
@macro(BIA = "@BI@*4585 Scotts Valley Dr. @*SV, CA")
```

The macro *BIA* calls up and runs the macro *BI,* and continues with the execution of the new macro. (The @* commands are formatter commands that tell the formatter to break the line.)

Tips

If the macro will be used only in a single file, you can insert the Macro command in the file, before you want to execute the macro. If the macro will be used in several files, you should insert the macro command in a copy of the STANDARD.FMT file (or the .FMT file you typically use to format your documents).

See Also

See Chapter 4 for an in-depth discussion of the Macro command.

MakeOdd

Keystrokes

@MakeOdd

Function

Tells the formatter to start printing on an odd-numbered page.

Use this command when you want to force text to appear on an odd page. In STANDARD.FMT, MakeOdd is part of the Chapter command definition, which is why chapters in Sprint documents always begin on an odd page. The MakeOdd macro looks like this:

```
@macro(makeodd() = "@newpage@if(page&1,else
   "@blankpage()")")
```

This says to start a new page and then decide if the page number is odd or even. If the page number is odd, the formatter begins printing text; if the page number is even, the formatter will insert a blank page and begin printing on the following (odd) page.

How To

Ordinarily you use the @-sign command in .FMT files. You can, however, also include the command in your normal Sprint documents. For example, to force text to print on an odd page, choose Style/Other Formats, and type `MakeOdd`. Press `C` to indicate that this is a command. Used with the Table command, it would look like this in a Sprint document:

```
MAKEODD
BEGIN TABLE
```

In the example, MakeOdd ensures that the table will begin on an odd-numbered page.

Merge_Init_

Keystrokes

none

Function

Tells the formatter to carry out certain commands at the start of every SprintMerge record.

This command is defined in STANDARD.FMT to start each record on a new page and to reset all variables. Whenever Sprint starts a new record, it *automatically* invokes the Merge_Init_ command (that's why there are no "keystrokes" listed for you to press).

If your merged records require some special handling (like needing a custom footer printed on every other page), you can customize the Merge_Init_ command by changing its definition in your .FMT file.

For example, to create a special footer for records appearing on odd-numbered pages, you could enter a Merge_Init_ command like this in your .FMT file:

```
@macro(Merge_Init_ =
"@place(toc)@place(tof)@newpage@Reset@PassInit@FooterO["You
May Already Be a Winner!"]")
```

NoFloats

Keystrokes @NoFloats

Function Prevents any more "floating" formats (for example, figures) from being printed on the current page.

A "floating" format is any command that's defined to print at the top or bottom of a page (that is, if it has the *above* or *below* parameter in its definition) or before or after some other part of the document has printed. The page heading commands (like Chapter) all float to the top of the page and automatically include a NoFloats command in their definition (so you can't get two commands vying for the top of the page).

If you define a command that floats, you can use the NoFloats command to prevent figures from floating to a particular page.

PageInit

Keystrokes @PageInit

Function Allows you to define a command string that is carried out by the formatter at the start of every new page.

The PageInit command is executed as soon as the formatter knows it is starting a new page. PageInit can be executed on the second or third line that will be printed on that page, and this can happen anywhere—generally right between two words in the regular text. If you define it to print something, for example,

```
@macro(PageInit = "ding-dong")
```

you will get the interesting but useless effect of having "ding-dong" stuck into your text somewhere at the top of every page.

It is far more practical to use this command to format text into a so-called floating format. STANDARD.FMT defines two floating formats for this, PageHead and PageFoot. These are one-column formats defined as "above page" and "below page," respectively. The PageInit command formats text into both of these formats. It also prints the current page number on the screen with an @Message command.

So that you can modify the page headings and footings inside a document, their actual text is saved in the string variables *phe, pho, pfe, phf, pff,* and *pfo.* These strings are initialized with the PassInit command (also in STANDARD.FMT) to print the page number on the bottom and nothing on top. Then, the @Header and @Footer commands are used to set these variables to various concatenations of the left, right, center, and line arguments.

Parent

Keystrokes @Parent{*variable=numeric variable*}

Function Prints two-tiered numbers.

Since Sprint can reference the values of variables during formatting, the Parent command lets you include the value of a numeric variable (such as chapter) when printing the value of another variable (such as page, figure, table). A typical use for this command is to include the chapter number as part of the page number (for example, page 1-1, 1-2, and so on). In this example, *chapter* is the *parent* for *page.*

The *subsection* variable also helps to explain the function of the Parent command. The subsection number (the value of the variable *subsection*) is dependent on the section number (the current value of the variable *section*). For example, if the value of *section* is 1, the first SubSection command causes the formatter to print 1.1. The next subsection is numbered 1.2. Each time you choose a SubSection command, the formatter includes the value of its *parent* (the value of *section*).

Note: Every time you start a new section, subsection numbering starts over again at 1. Likewise, with every new *chapter,* section numbering starts over at 1. The Parent command provides this capability. *The overall rule* *is that whenever a parent is changed, its "children" are automatically reset to zero.*

Look at the STANDARD.FMT file for further examples of the Parent command at work.

How To

To include the value of a *parent* variable when printing another variable, use the Parent command. You can do this either in an .FMT file or in a Sprint document. For example,

```
@PARENT(figure=chapter)
```

This example appears in the STANDARD.FMT file. It tells the formatter that whenever it prints a figure number, it should also include the chapter number. Figures in chapter 1 will begin with the number 1, followed by the number of the figure (for example, 1.1, 1.2, 1.3, and so on); figure numbers in chapter 2 will begin again at 1, and will be prefaced with the new chapter number (for example, 2.1, 2.2, 2.3, and so on).

The STANDARD.FMT file includes a Parent command for the *Figure, Table, Paragraph, Section,* and *Subsection* variables. *Chapter* is defined as the parent variable for *Figure, Table,* and *Section. Section* is the parent of *Subsection;* and *Subsection* is the parent of *Paragraph.*

If you don't want variables to print with their defined parent, edit the STANDARD.FMT file and delete the appropriate Parent command.

By default, page numbers *do not have a parent assigned* to them. This means that pages are numbered sequentially, beginning with the number 1. You may prefer to include the current chapter number as part of the page number. If you want to print the chapter number as part of the page number, type the following command in the STANDARD.FMT file:

```
@PARENT[page = chapter]
```

Page numbers will now print as 1.1, 1.2, 1.3, 2.1, and so forth (and as usual, they will have dashes before and after them). If you want the chapter and page numbers separated by a dash, rather than the default period, you need to add the Template command, which tells the formatter *how* to print the page numbers:

```
@TEMPLATE(page="%#-%]%d")
```

The above command consists of two parts, *%#-%]* and *%d*. The first part (`"%#-%]`) tells the formatter to print the parent of the variable being defined (*page*), followed by a "-". The second part tells it to then print the value of the variable *page*.

After the two commands above, Sprint would change its page number format to *Chapter-Page* (for example, *2-14*). See the Template entry in this chapter for details on numbering templates.

ReadEPS

Keystrokes

@ReadEPS[*filename*]

Function

Checks *filename* to see whether it is a true encapsulated PostScript file.

This command makes sure the named file starts with the characters %! and has a "BoundingBox" comment in the header of the file. If the file doesn't have these two things, the command generates an error message telling you the file is not a legal EPS file.

If the file satisfies these criteria, the command reads the dimensions found in the BoundingBox comment into the variables *llx, lly, urx,* and *ury.*

If all this is Greek to you, don't use the ReadEPS command.

Reset

Keystrokes

@Reset

Function

Releases all variables except tags.

The Reset command makes all non-tag variables undefined, as though they had never been set. This command is primarily used by the Merge_Init_ command to clear the slate after each merged document is printed.

VUnits

Keystrokes @VUnits[*n*]

Function Moves the print head *n* units down (if the number is positive) or up (if the number is negative).

This command uses printer units, which are the printer's primitive unit of measurement. For this reason, a unit on one printer might not be the same as a unit on another.

How To Use this command to construct sophisticated (but printer-dependent) commands. STANDARD.FMT uses @VUnits in its special EPS commands (for use with PostScript printers).

VUnits is often used in conjunction with HUnits.

Warn

Keystrokes @Warn[*warning message text*]

Function Reports a warning in user-defined macros.

The Warn command works just like the Error command except that it generate a warning message instead of an error. Warning messages do not abort the formatting passes, so documents that generate warnings (but no errors) still print.

You use this command to generate warning messages in your macros just like the built-in Sprint warning messages, complete with the file name and line number containing the problem.

Tips Refer to the Error command entry in this chapter for details and examples on using this command.

* (Asterisk)

Keystrokes @*

Function This command forces a hard return to be executed.

How To

Insert this command wherever you want a hard return (the equivalent of pressing *Enter*). For example, if you're defining a format that automatically skips three lines before printing the word *Warning* in large, bold letters, you would include this in the definition:

```
...@*@*@*@large[Warning]...
```

The result would be three blank lines (three hard returns in a row), followed by

Warning

, (Comma)

Keystrokes

@,

Function

Produces a comma when two of these commands appear next to each other.

You use this command to create a comma to separate conditional references in macros. Note that a single @, command does nothing.

For example,

```
   @+_(@,@ref(footnote)@,)
```

This line is taken from the STANDARD.FMT definition of the Footnote command. If two footnotes are placed together in the text referring to the same word, the two footnote numbers appear with a comma between them (because there are then two @, commands in a row).

The command is also responsible for Sprint's knowing to put a comma between page numbers in an index entry but not after the last page number.

; (Semicolon)

Keystrokes

@;

Function

Tells Sprint to "do nothing," which means it halts the current activity.

If you use this command after @~ or @′, it prevents them from removing whitespace.

Tips

You can also use this command (in normal documents) to prevent Sprint from splitting a word at a hyphen.

For example,

```
co-@;op
```

keeps Sprint from ending a line at this hyphen no matter what.

‘(Single Close Quote)

Keystrokes

@’

Function

Ignores all whitespace immediately following.

How To

To remove a line break, tabs, or space characters, enter @’. For example:

```
Here is a line @’
   and another that will be on the same line, @’

and a third that won't have a blank line above it.
```

Result:

Here is a line and another that will be on the same line, and a third that won’t have a blank line above it.

Exception: When an @‘ command (ignore preceding whitespace) precedes an @’ command, and if @‘ *isn’t* at the beginning of a line, Sprint will not execute the @‘ command and will instead perform the @’ command. This is useful for removing whitespace within macros. For instance, the command @DoSomething could be defined as follows:

```
@form(DoSomething() = "@‘@message<do one thing and
another>@’")
```

The @’ and @‘commands around the macro cause the following text:

```
This is a sample @DoSomething(sample) of
"DoSomething."
```

to print correctly:

This is a sample of "DoSomething".

If the @’ and @‘commands weren’t there, there would be two spaces in the output between the words "sample" and "of." However, if @DoSomething is entered this way:

```
text text text text text text text text text text
@DoSomething(text) text text text text
```

@’ will be executed

‘ (Single Open Quote)

Keystrokes @‘

Function Ignores whitespace preceding the command.

This command ignores all space characters or tabs (whitespace) that appear before the command.

How To To eliminate leading whitespace in a line, enter @‘. For example:

```
this       @‘ that
```

prints as:

this that

Typically, you use this command at the start of formatter macros, so that the macro can be set off by a space in the document without distorting the printout. The following example shows how @‘ is used in the TEXT.MAK definition of the Index command. Index is defined as:

```
@macro(Index() = "@‘@TheIndex(e=text,v="@, @eval(page)@,")@’")
```

Let’s say your text includes the text:

> ...various chemicals in the urban drinking water @index{chemicals, in water}, and the pollutants introduced by various industries, have an adverse affect on the taste, as well as the nutritional value, of this vital resource.

> The following table lists the chemicals found in our water, and their effects.

Notice that there is a space before the Index command. This is an *extra* space, since you don't want it to print before the comma following the index entry. If the Index command definition didn't contain the @‘ command, the line of text would print like this:

> ...various chemicals in the urban drinking water , and the pollutants introduced by...

Instead, the formatter removes the extra space before printing the comma.

Tips

This command is useful at the start of formatter macros, which allow macros to be set off by a space in the document without distorting the printout.

If you're a Borland SuperKey user, and have this program loaded, you'll need to type the open quote twice to get it to print on the screen.

@~ (Tilde)

Keystrokes

@~

Function

Ignores all whitespace.

This command eliminates all whitespace (spaces, tabs, and blank lines) up to the next printing character.

How To

You sometimes need this command in .FMT files when you need the formatter to ignore any spaces that intervening commands may have introduced.

APPENDIX

D

Format Parameters

A command or format can include a variety of *parameters,* or modifiers. These parameters affect the text within the format; they take effect as soon as you begin the format, and end when you end the format. If you've started a format, and before ending it start another format, the second format not only has its own set of parameters, but also inherits the parameters of the *parent* format (the format you already started but haven't ended).

These parameters can be used with virtually any format—assuming it makes sense to do so. For example, *Group* could logically be used with the RESERVE command, but the *Underline* parameter makes no sense there.

The one exception to this rule is the Style command: There are several parameters that make sense *only* with the Style command because they make global changes. These are *BottomMargin, FormFeed, LeftMargin, Paper, PaperWidth, RightMargin, TabSize,* and *TopMargin.*

You also can use any of the parameters when you define your own format using the Define command in .FMT files. If you use these parameters to modify existing commands in .FMT files (or to create brand-new ones there), you'll be using the @-sign versions of the commands. If you use these parameters to modify formats in your Sprint files, you should use the menus to insert the command and then choose Style/Modify to add the parameter.

Here are some examples of parameters at work:

In Sprint documents:

```
HYPHENS, group, font Helvetica

BEGIN HEADER, size 6 points, linelength 5 inches

STYLE, notct, spread 1.5 lines, fill on, leftindent + 3 picas

TEXT, above 2 lines, below 2 lines, centered, font elite,
      ifnotfound pica, leadingspaces kept
```

In .FMT files:

```
@Define(myfont, font Times, size 12 points, script + 3 picas,
        spacing 13 points)

@modify(PageHead, linelength 6 inches)
```

Table D.1 lists all the parameters that Sprint recognizes.

Table D.1: Format Parameters

Above
AbovePage
After
AfterEntry
AfterExit
Before
BeforeEach
BeforeExit
Below
BelowPage
BlankLines
BottomMargin
Centered
Column
Columns
Comments
Counter
Divider
Fill
Flushleft
Flushright
Font
FormFeed
Free
Group
Gutter
IfNotFound
Increment
Indent
Index
Initialize
Inline
Invisible
Justify
LeadingSpaces
LeftIndent
LineLength
Margins
Numbered
Offset
OverStruck
Paper
PaperLength
PaperWidth
RightIndent
Script
Size
Spacing
Spread
Strikeout
TabSize
TCT
TopMargin
Underline
VerticalJustify
WidestBlank
WithEach
Within

Most of these parameters has its own section in this appendix, and all of them have brief descriptions in the complete list that starts on page 429.

It's often useful to categorize parameters; that is, some parameters affect the typeface of text, others affect where on the page the text will appear. Table D.2 lists the various categories of format parameters, and briefly describes the function of each.

Table D.2: Parameter Categories

Typestyle Parameters	*Font, IfNotFound, Invisible, OverStruck, Script, Size, StrikeOut, Underline* These are parameters that can be used with typeface commands. If you specify any parameters other than those listed, the command is *not* considered a typeface command, and will therefore start on a new line. If you don't specify new parameters for the command, the format *inherits* these parameters from the parent format. (The parent format is the format enclosing the format you invoke.) Therefore, if the parent format is already printing in bold type, the new format will also.
Formatting Parameters	*Above, AbovePage, Below, BelowPage, BlankLines, BottomMargin, Centered, ColumnOffset, Columns, Fill, FlushLeft, FlushRight, FormFeed, Group, Indent, Justification, Justify, LeadingSpaces, LeftIndent, Margins, Paper, RightIndent, Spacing, Spread, TabSize, TopMargin, VerticalJustify, WidestBlank* These parameters change the text layout. If any of them have non-null values, text within the format will begin on a new line; text following the END command will also start on a new line.
Floating Formats	*Above, After, Before, Below, Free, Index, Inline* These parameters create different types of floating formats (those which are forced to appear at the top or bottom of a page, or at the beginning or end of the document). You can only use one of them in a given format definition.
Enumeration Parameters	*Counter,* Increment, Numbered, Within Sprint can *count* the paragraphs within a format. This count is stored in variables, and all the commands to set or print the value of variables work on these parameters.
Macro Parameters	*AfterEntry,* AfterExit, BeforeEach, BeforeExit, Divider, Initialize, WithEach These parameters define command strings that are automatically executed at various places in the format. This gives the format *macro* capabilities, and allows for special effects.

When an entry says it uses a *dimension,* this means you can use any appropriate Sprint dimension (like inches, picas, characters, or lines). Refer to the "Dimensions" entry in the *Reference Guide* for a full list.

Starting on the next page are detailed descriptions of most of the parameters along with examples of how to use that parameter in a command definition. Many of the examples use @-sign commands because it's assumed that you'll often be using them in your .FMT files (style sheets).

Following the detailed descriptions is Table D.6 (on page 429), which is a complete list of all the parameters recognized by Sprint and short descriptions of each.

Above

Syntax: Above *dimension*

This formatting parameter defines how much extra blank space (*leading*) will be inserted before the first line of the format. The specified blank space will appear between the preceding text and the first line of text within this format. If you insert blank lines at the start of a format, the formatter will ignore them unless they exceed the *Above* dimension. For example,

```
BEGIN QUOTATION margins +.5 in, above 1, below 1, spacing 1
```

The Quotation command automatically inserts a blank line before beginning the text within the format. If a blank line precedes the BEGIN QUOTATION command, the formatter will ignore it; however, if there are two or more blank lines preceding the command, the formatter will insert this blank space in the printed text.

If you don't include a dimension, but instead follow the *Above* parameter with a comma or the name of another format, the meaning of this parameter changes. See the following *Above* section.

Above

Syntax: Above

If you don't include a dimension with the *Above* parameter, *Above* becomes a *floating format* parameter. Floating formats are those formats that the formatter automatically moves (floats or sinks) to either the top or the bottom of the page. Page headers and footers are examples of floating formats; page headers always float to the top and page footers always sink to the bottom of the page.

Formats that include the *Above* parameter with no dimension specified always print at the top of the page. If you use a format that includes the *Above* parameter within multi-column text, it will float the text to the top of the column instead of the page. For example:

```
BEGIN TEXT, COLUMNS 2
Here's some text that will print in two columns. Before ending this
format, we'll introduce a format that uses the Above parameter. Note
that the following text will actually print at the top of the column.

BEGIN QUOTATION, above, font bold
This text appears at the top of the current column, since the Text,
Columns 2 format is still in effect.
END QUOTATION

Here's the rest of the text to finish out the Text format.
END TEXT
```

The example prints like this:

This text appears at the top of the current column, since the *Text, Columns 2* format is still in effect.

Here's some text that will print in two columns. Before ending this format, we'll introduce a format that uses the *Above* parameter. Note that the following text will actually print at the top of the column.

Here's the rest of the text to finish out the *Text* format.

If you specify *Above Page,* the format will take the full page width. For example,

```
BEGIN TEXT, COLUMNS 2
Here's some text that will print in two columns. Before ending this
format, we'll introduce a format that uses the Above parameter. Note
that the following text will actually print at the top of the page.

BEGIN QUOTATION, above page, font bold
This text appears at the top of the page and across the width of the
page, even though the Text, Columns 2 format is still in effect.
END QUOTATION

Here's the rest of the text to finish out the Text format.
END TEXT
```

Look at the following page to see how the example now prints.

This text appears at the top of the page and across the width of the page, even though the *Text, Columns* 2 format is still in in effect.

Here's some text that will print in two columns. Before ending this format, we'll introduce a format that uses the *Above* parameter. Note that the following text will actually print at the top of the page.

Here's the rest of the text to finish out the *Text* format.

If the formatter encounters a format that includes the *Divider* parameter, the *Divider* text, if any, will print below the *Above* format. For example, if you added the *Above* parameter to the Figure format definition (to force figures to appear at the top of the page) and also added the *Divider* parameter to the definition (to create a divider between the figure and the text following the figure), the divider text (an underline, perhaps) would appear *below* the figure. If the same format is used more than once on the page (e.g., two or more Figure formats), the divider text is printed only once, below all of them. If the format is too big to fit on the page above the text that invoked it, and if the format is not grouped, the formatter splits the text of the format and continues on the next page; the divider then appears below both sections. You can force such splits by choosing Insert (Uncoditional) Page Break command.

If you invoke more than one *Above* format on a single page, the formatter stacks them in the order in which they were invoked.

Also refer to the *Below* parameter for related information.

After

Syntax: After

This floating-format parameter saves the text in a floating format for later execution, instead of printing the text immediately. The formatter fills in the Variable commands in the text, but doesn't print anything until after all other text in the document prints. During formatting, the formatter appends the *After* text in memory. If you specify more than one *Before* or *After* format in a document, the text of these formats is printed in the order in which it was defined.

When the formatter reaches the end of the document (or encounters the formatter command Place), it then invokes the *After* format, and formats all the text in it at this place in the file. This can be used for endnotes, bibliographies, tables of authority, and other pieces of text that are referenced throughout the document, but printed together at the end. For example, the *Notes* definition looks like this:

```
@Define(Notes, after, above 3, spacing 1, indent 5, initialize
        "@Heading(Notes)")
```

This definition instructs the formatter to print the text of the endnotes *after* the text of the document completes printing.

AfterEntry

Syntax: AfterEntry *"string"*

This macro parameter executes the specified string at the start of the format. It acts as though it's right after the opening BEGIN command. For example, you might create a format that displays a line of asterisks before printing any text. This format could be defined as follows:

```
@Define(StarScreen = Verbatim, group, afterentry "@>{*}@*",
        beforeexit "@*@>{*}")

BEGIN STARSCREEN
                    Here's a starry screen display.
END STARSCREEN
```

The example prints like this:

```
****************************************************************************************************
                              Here's a starry screen display.
****************************************************************************************************
```

Note: The parameters *AfterEntry* and *Initialize* have the same effect.

AfterExit

Syntax: AfterExit *"string"*

The string in this macro parameter is special, in that it's not executed *within* the format. Instead, it is saved until after the format ends, and then executed in the enclosing format. It acts as though it is placed immediately after the END command. The primary use for this parameter is to print reference numbers in *floating* formats, such as footnotes. For example:

```
@Define(Foot, font small, ifnotfound, size .707, spacing 1,
        below, counter footnote, above .5,
        divider "@u(@>)",
        AfterEntry "@ref(Footnote). @[",
        AfterExit "@+(@,@ref(Footnote)@,)")
```

The *AfterExit* string in this example says that immediately after the Footnote command ends, the formatter must superscript (print above the baseline) the value (number) of the footnote (the @, is used to force multiple references to include commas between them).

Before

Syntax: Before

Before is like the *After* parameter, except that the *Before* format text is formatted for placement at the *start* of the document. For example, the table of contents is a *Before* format, as shown in the partial *TOC* definition below:

```
@Define(TOC, before, indent -3, justify no, margins .5in, ...)
```

The pages for the *Before* format are numbered starting with the Roman numeral *i*. The variable *FirstPage* is set to the last page used by the *Before* format. If you start the document with the command *Set Page=FirstPage,* the page numbering will continue from the last page of the printed *Before* format.

Note: The text of a *Before* format is actually printed after the document is printed; you need to move the printed pages to the front of the printed document.

BeforeEach

Syntax: BeforeEach *"string"*

This parameter executes the specified string at the start of each paragraph. For instance, to print a dash before each paragraph in the format, use *BeforeEach "- "*. A paragraph must be separated from the previous one by a blank line and *cannot* be indented. If a blank line doesn't separate two paragraphs, the second paragraph is considered a *sub-paragraph* (part of the previous one). This is the same way paragraphs are counted with the *Numbered* parameter.

The Numbered format definition includes the *BeforeEach* parameter to number each paragraph within the format:

```
@Define(Numbered, indent -6,above 1,below 1,
        numbered "%&.%]%<%u%;%a%;%i%]",
        BeforeEach"@>()@value(counter).@hsp(.2in)@\")
```

The *BeforeEach* parameter in the Numbered example tells the formatter to begin each paragraph with the value (number) of the counter, followed by a period. After the formatter prints the number and period, it performs the Hsp (horizontal space) command, which moves the print head to the right 0.2 inch. Finally, the *BeforeEach* string instructs the formatter to insert an @\ command to set the wrap margin for the rest of the paragraph.

The Hyphens command definition includes this parameter to print a dash at the beginning of each paragraph in this format, and an asterisk at the beginning of nested Hyphens paragraphs:

```
@Define(Hyphens=Numbered,numbered "%<-%;*%]",BeforeEach
        "@>()@value(counter)@hsp(.2in)@\")
```

Note: This parameter is similar to *WithEach* (see page 426); only one of these two parameters may be specified within a single format definition.

BeforeExit

Syntax: BeforeExit *"string"*

This macro parameter executes the specified string at the end of the format. It acts as though it is placed immediately before the END command.

For example, you might create a format that displays a line of asterisks after printing the text of the format. This format could be defined as follows:

```
@Define(StarScreen = Verbatim, group, afterentry "@>{*}@*",
        beforeexit "@*@>{*}")

BEGIN STARSCREEN
                    Here's a starry screen display.
END STARSCREEN
```

The example prints like this:

```
*******************************************************************************************
                    Here's a starry screen display.
*******************************************************************************************
```

Below

Syntax: Below *dimension*

This formatting attriubte defines how much extra blank space (*leading*) that will be inserted below the last line in the format. It works like the *Above* parameter at the end of a format.

If you don't include a dimension, but instead follow the *Below* parameter with a comma or the name of another format, the meaning of this parameter changes. See the following *Below* section for details.

Below

Syntax: Below

This floating-format parameter is just like *Above* (no dimension), except these formats appear below the main text. This is used for footnotes. Formats that specify *Below* (no dimension) will appear at the bottom of the current column.

BlankLines

Syntax: BlankLines *hinge/break/kept*

This parameter can be set to *one* of those listed above. *Break* causes multiple blank lines to be ignored (the formatter will print only the *spread* value). *Kept* just turns off *break*. Hinge is like *Break,* but causes the formatter to automatically perform a PGBREAK command (see the "Page Breaks" entry in the *Reference Guide*) on each blank line; you should also add the *group* parameter to make this useful. For example,

```
@Define(Center, centered, indent 0, group, blanklines hinge)
```

Kept is the default.

Centered

Syntax: Centered
        Justify Center

Centered

This formatting parameter centers text between the current left and right margins. For example,

```
@Define(Center, centered, indent 0, group, blanklines hinge)
```

This is the definition for the Center format, which centers text between the current left and right margins. For example,

```
BEGIN CENTER
Caution!
Slow children on board!
END CENTER
```

The example prints like this:

Caution!
Slow children on board!

By default, the *Centered* parameter also turns fill mode off, but you can turn it back on by following the *Centered* parameter with the *Fill* parameter. For example,

```
@Define{nonsense=center, fill on}
```

If you formatted the earlier example with the Nonsense format, the text would print like this:

Caution! Slow children on board!

Columns

Syntax: Columns *number*

This formatting parameter specifies the number of columns for the format. If you don't include this parameter, or give it a value of zero, the defined format will print in as many columns as defined by the parent format. If you include a number with this parameter (including *1*), the formatter will end the current column(s), balance the column(s) at the top of the page, and then divide the page into the specified number of columns for the duration of this format. After you exit this format, the columns are ended and balanced, and the previously specified number of columns resumes. For example,

```
@Define{MultiColumn=text, columns 4}
BEGIN TEXT, COLUMNS 2
Some years ago, it was impossible to buy a package of one particular
chicken part--breasts, wings, drumsticks, or thighs. As consumption of
chicken increased because of nutritional considerations and cost,
poultry packers had an incentive to produce pre-packaged, one-of-a-kind
chicken parts.

After these appeared on supermarket shelves, health-conscious cooks
proved willing to pay a little extra for boneless, skinless chicken
breasts. Now, boneless, skinless, TASTELESS chicken is available at
many markets.

BEGIN MULTICOLUMN
This text is totally unrelated to chicken. In fact, many of us are so
tired of chicken, we'd rather not read anything about it. Hamburgers,
though high in fat, calories, and numerous other dietary nightmares,
are a much more interesting topic, especially for dieters. A so-called
chicken sandwich just can't compare with a juicy, messy, incredibly
large burger. And if you're really into blowing a diet, you can add
guacamole, extra cheese, salsa, bacon, mushrooms, barbeque sauce,
ketchup, mayonnaise, mustard, and relish, as well as the standard
lettuce, tomato, onion, and pickles. Jimmy Buffet once wrote a song
entitled "Cheeseburger in Paradise" the lyrics of which lead me to
believe he'd been on a diet for far too long.
END MULTICOLUMN

Now we return to a two-column format, so it makes sense that we return
to the subject of chicken. The only food one might find more boring
than this is smelly fish!
END TEXT
```

The example prints like this:

Some years ago, it was impossible to buy a package of one particular chicken part—breasts, wings, drumsticks, or thighs. As consumption of chicken increased because of nutritional considerations and cost, poultry packers had an incentive to produce pre-packaged, one-of-a-kind chicken parts.

After these appeared on supermarket shelves, health-conscious cooks proved willing to pay a little extra for boneless, skinless chicken breasts. Now, boneless, skinless, TASTELESS chicken is available at many markets.

This text is totally unrelated to chicken. In fact, many of us are so tired of chicken, we'd rather not read anything about it. Hamburgers, though high in fat, calories, and numerous other dietary nightmares, are a much more interesting topic, especially for dieters. A so-called chicken sandwich just can't compare with a juicy, messy, incredibly large burger. And if you're really into blowing a diet, you can add guacamole, extra cheese, salsa, bacon, mushrooms, barbeque sauce, ketchup, mayonnaise, mustard, and relish, as well as the standard lettuce, tomato, onion, and pickles. Jimmy Buffet once wrote a song entitled "Cheeseburger in Paradise" the lyrics of which lead me to believe he'd been on a diet for far too long.

Now we return to a two-column format, so it makes sense that we return to the subject of chicken. The only food one might find more boring than this is smelly fish!

Counter

Syntax: Counter *{variable}*

This enumeration parameter names a global variable that acts as the *counter*, and is incremented each time you invoke the format. You can print the value of this variable by referencing *Counter*, as well as the name of the global variable you've selected. The Footnote command definition provides an example of this:

```
@Define(Foot, font small, ifnotfound, size .707, spacing 1,
        below, counter footnote, above .5,
        divider "@u(@>)",
        AfterEntry "@ref(Footnote). @[",
        AfterExit "@+(@,@ref(Footnote)@,)")
```

The *counter footnote* parameter tells the formatter to keep track of the number of Footnote commands entered, and to increment the counter each time a Footnote command appears in the file. If you want to print the current footnote number, you could choose the Variable command and type `footnote`.

If you don't name a variable in the definition, the formatter will use the *local* counter (the counter currently in effect).

Divider

Syntax: Divider *"string"*

This macro parameter produces a kind of visual separation, e.g., the bar that appears above footnotes at the bottom of the page. The text produced by this string will be printed between the invocations of this format and the running text. The Footnote command includes the *Divider* parameter in its definition:

```
@Define(Foot, font small, ifnotfound, size .707, spacing 1,
        Below, counter footnote, above .5,
        divider "@u(@>)",
        AfterEntry "@ref(Footnote). @[",
        AfterExit "@+(@,@ref(Footnote)@,)")
```

The parameter *divider "@u(@>)"*, specifies a solid underline character repeated to the right margin. For example,

> Here's a footnote referenced in text.[1] Look at the bottom of this page for an example of how the *Divider* parameter works.

Note: You can include *Divider* only in a *floating* format definition (a definition that contains one of the following parameters: *Above* (no dimension), *After*, *Below* (no dimension), *Before*, *Free*, or *Index*).

Fill

Syntax: Fill *Yes/No*, *On/Off*, or *1/0*

This formatting parameter specifies whether the formatter should fill out the line length of paragraphs. The formatter recognizes a paragraph as text followed by either a blank line (two *Enter* characters in a row), the formatter command @*, or a line that's indented. *Fill On* means that if a line in the file ends without one of these, the formatter replaces the soft or hard return with a space, and the paragraph continues. For example,

```
BEGIN TEXT, Fill On
Here's an example
of how the Fill parameter
works. Regardless of where the hard
return characters are in the file,
the
formatter fills up the
lines
between the left
and right margins.
END TEXT
```

[1] This is the text of the footnote

The printed example:

> Here's an example of how the Fill parameter works. Regardless of where the hard return characters are in the file, the formatter fills up the lines between the left and right margins.

Fill On means that you want the lines to end wherever you have a hard return, and forces *verbatim* output. To show the difference, here's the same example text affected by the *Fill Off* parameter.

```
BEGIN TEXT, FILL NO
Here's an example
of how the Fill parameter
works. Regardless of where the hard
return characters are in the file,
the
formatter fills up the
lines
between the left
and right margins.
END TEXT
```

Prints like this:

> Here's an example
> of how the Fill parameter
> works. Regardless of where the hard
> return characters are in the file,
> the
> formatter fills up the
> lines
> between the left
> and right margins.

Fill Off also turns off justification. You can turn it back on by following the *Fill* parameter with the *Justify* parameter.

FlushLeft

Syntax: FlushLeft
Justify Left

This formatting parameter justifies text at the current left margin. As each line completes, or a tab appears, the formatter aligns the text at the left margin. For example,

```
@Define{PushLeft, font italic, flushleft, fill off}
BEGIN PUSHLEFT
The PushLeft command is a
command that we made up to demonstrate
how the FlushLeft/Justify Left parameter works.
It could be used to format addresses in letters.
END PUSHLEFT
```

This example creates a command called PushLeft that prints text in an italic font, aligns text at the current left margin, and does not fill lines. The example prints like this:

The PushLeft command is a
command that we made up to demonstrate
how the FlushLeft/Justify Left parameter works.
It could be used to format addresses in letters.

If you don't specify a *Justify* parameter in your format definition, its setting is inherited from the parent format. The default for the outermost page format is *On;* the formatter justifies text at both the left *and* right margins.

FlushRight

Syntax: FlushRight
Justify Right

This formatting parameter justifies text at the current right margin. As each line completes, or a tab appears, the formatter aligns the text at the right margin. For example,

```
@Define{PushRight, font italic, flushright, fill off}
BEGIN PUSHRIGHT
The PushRight command is a
command that we made up to demonstrate
how the FlushRight/Justify Right parameter works.
It could be used to format return addresses in letters.
END PUSHRIGHT
```

This example creates a command called PushRight that prints text in an italic font, aligns text at the current right margin, and does not fill lines. The example prints like this:

The PushRight command is a command that we made up to demonstrate how the FlushRight/Justify Right parameter works. It could be used to format return addresses in letters.

Font

Syntax: Font *fontname(s)*

This typeface parameter specifies the font(s) to be used in this format. For example,

```
@Define(Example = Display, font elite pica courier, size 10 pt,
        ifnotfound)
```

If the printer has a font or parameter that matches one of the font names listed, it uses it for the duration of the format. If there is more than one match, the formatter uses the first one, and the other font names are ignored. The formatter will generate a warning during formatting if none of the specified font names exist
for the current printer, unless the *IfNotFound* parameter follows, as shown above.

The Example definition shown above is based on the Display format, but includes the *Font* parameter. This means that the formatter should format the text as if it were in a Display format, but with a different font. The formatter will first check to see if the printer can print an *elite* font. If it can, the text will print with this font; if it can't, the formatter will determine whether the print can print in a *pica* font, and so on. (See also *Size* and *IfNotFound* in this appendix).

If you specify a single font like *Font bold,* and your printer doesn't have this capability, it may attempt to double-strike the characters to make them darker. If the printer doesn't have an alternative to match (or approximate) the font you specify, the formatter will display a warning message indicating the specified font cannot be found. The document will print, but the text will print in the printer's default font.

Free

Syntax: Free

Free formats begin in the current column, immediately after the line in which they are invoked, and before the next line. This can be used for *local* floating figures, quotations, and notes. They differ from regular formats in that text after them may be added to the preceding line by the format to fill it to the right margin. As soon as the formatter finishes the line, the text of the free format will be printed.

Group

Syntax: Group *Yes/No*

This formatting parameter prevents page breaks anywhere inside the format. This is the same as enclosing the entire format in a Group format. If you don't specify this parameter in your format/command definition, its setting is inherited from the parent format.

The Figure command uses the *Group* parameter, to prevent figures from being split across pages. Its definition looks like this:

```
@Define(Figure, above, group, columns 1, spacing 1,
        fill n, linelength 0, notct, below 1)
```

This definition specifies text centered between the current left and right margins, with indentation (if specified by the parent format) turned off. All text within the Center format will appear together on the same page (as specified by the *Group* parameter). If the text contains multiple blank lines, the formatter will ignore them and insert a single blank line; if the text cannot fit on a page, the formatter will break the page at a blank line.

Gutter

Syntax: Gutter *distance*

This formatting parameter specifies the gap (amount of blank space) between columns in a multi-column format. The Index format, for example, specifies a .75 inch gutter between the columns in the index, as shown in the partial definition below:

```
@Define(TheIndex, index, columns 2, spread .5, spacing 1,
        justify no, margins +.75 inch, gutter .75 inch, indent -5, ...
```

If you don't specify the *Gutter* parameter, its setting is inherited from the parent format. The default is three characters.

IfNotFound

Syntax: IfNotFound

This typeface parameter is meaningful only if preceded by a typeface parameter (*Font, Invisible, Overstruck, Script, Size, Strikeout,* or *Underline*). If you follow a typeface parameter with the *IfNotFound* parameter, and the printer can't print the typeface specified, the formatter will not give a warning. For example,

```
@Define(Example = Display, font elite pica courier, size 10 pt,
        ifnotfound)
```

This format tells the formatter to print the text in the Display format, but use either an elite, pica, or courier font. In addition, the formatter should print the text in 10-point type, but if the printer doesn't have this capability, it should ignore the command (but not give any message). Without the *IfNotFound,* an error mesage would be generated.

The other use of *IfNotFound* is to provide a substitute for a typeface. The example *Font italic, IfNotFound bold* tells the formatter to print text in italics; if the printer doesn't have an italic font, the *IfNotFound bold* parameter tells the formatter to use bold for the text within this format instead.

If you don't include the *IfNotFound* parameter, and specify a font that your printer can't print, you'll get a warning message during formatting, and the text will print in the printer's default typeface.

You can also follow *IfNotFound* with an alternate typeface, to be used only if the printer can't print with the typeface you specified prior to the *IfNotFound* parameter. For example,

```
font bold, ifnotfound, overstruck
```

will overstrike only if the printer does not have a "bold" font. In essence, you're giving the formatter an alternative, rather than having the printer try to simulate the desired typeface.

Increment

Syntax: Increment *variable*

Same as *Counter.*

Indent

Syntax: Indent *(+/–) distance*

This formatting parameter specifies the indent or outdent value for the first line of each paragraph in the format. This indent value is relative to the current left margin. If the distance is positive, it specifies by how much the first line of each paragraph should be indented. All paragraphs will start at this column, even if they don't look indented on the screen. For example,

```
@Define(Notes, after, above 3, spacing 1, indent 5,
        initialize "@Heading(Notes)")
```

The definition of the Notes command (which equals the Endnote command on the References menu) indents the text of the notes five characters from the global left margin.

If you don't want a paragraph to be indented, begin the paragraph with a single tab. Press *Tab* at the beginning of the paragraph, and then type the text of the paragraph.

If you want a paragraph within the format to be indented more than the specified indent value, you can begin the paragraph with two or more tabs; the text will be indented to the specified tab stop on the printed copy.

If the *Indent* distance is negative, the format is an *outdented* format. In this case, the *wrap margin* is set to the specified distance. Paragraphs that are not indented in the file will start printing at the left margin, but all continuing lines in the paragraph will start at the wrap margin. The Description format uses a negative indent distance of *–.25 line,* which means the first part of each paragraph will appear 1/4 of the line length to the left of the remaining text in the paragraph. To begin printing text at the wrap margin, press *Tab.* The following example shows you the definition of the Description format, followed by text formatted with Description.

```
@Define(Description, indent -.25 line, WithEach "@b(@eval)
        @\",above 1,below 1)
BEGIN DESCRIPTION
Milk Tab A nutritious beverage that's full of vitamins, calcium, and
protein.

Cookies Tab A not-so-nutritious snack that tastes great with milk.

Tab Homemade cookies are a favorite among toddlers, school children,
and adults alike.
END DESCRIPTION
```

The example prints like this:

Milk	A nutritious beverage that's full of vitamins, calcium, and protein.
Cookies	A not-so-nutritious snack that tastes great with milk.
	Homemade cookies are a favorite among toddlers, school children, and adults alike.

As shown in the previous example, if you indent a paragraph with the *Tab* key, the line will also start at the wrap margin.

Indentation is *not* inherited from the enclosing format. It is set to zero unless otherwise specified.

Index

Syntax: Index

The index is printed using a specialized *After* format. This type of format cannot be used with the formatter command Place, and it is called using a special command form.

A call to an index type format looks like this:

```
FormatName entry string, value string...
```

or

```
FormatName e string, v string
```

where the strings are quoted strings or the names of string variables.

The entries are alphabetized, and the values for each entry are appended together with them to make a string like:

```
EntryValueValueValue...
```

for each unique entry. Sprint ignores case and formatting commands when comparing entries. When Sprint finishes formatting the document, it formats the index. These strings are printed in the Index format, with a double hard return after each.

In these formats, each time the initial letter changes, the variable *Counter* is set to that letter (A is 1) and the *BeginEach* command is executed; this lets you title each letter.

Commas in the entry string are used to make multi-level indexes. The text before the comma locates the primary entry, and the rest of the text describes an entry in a *sub-index* which is printed after the entry. The sub-index formats just like the main index, except each line is printed with a tab in front of it. Sub-indexes can be nested any number of times.

Commas inside commands in the entry string are not used for this, so you can use the command *word text,text* to put commas in an index entry.

The typical user avoids all this complexity by using macros. The Index macro, defined in STANDARD.FMT, lets you make index entries containing the page number by choosing an Index command from the menus or using one of the equivalent @-sign index commands. It expands to:

```
@'@TheIndex(e = "@eval",v = "@, @eval (page)@,")@'
```

Notice the use of @, to place commas between the values provided for the page numbers.

Initialize

Syntax: Initialize *"string"*

This macro parameter executes the specified string at the start of the format. It acts as though it's right after the opening BEGIN command. The definition of the formatter command TCapt uses the *Initialize* parameter, as follows:

```
@Define(tcapt = center, size 8 point, font AvantGarde,
        counter table, Initialize="@*Table @value(table): ",
        BeforeExit="@*@ux{@>}@*@*@NoHinge")
```

This *Initialize* parameter in this definition states that immediately following a TCapt command, the formatter should begin a new line, print the word *Table* followed by the number (value) of the current table, and then print a colon and a space.

Also refer to the entry on the *AfterEntry* parameter for more information.

Invisible

Syntax: Invisible *Yes/No*

This typeface parameter lets you hide the characters inside a format; the formatter instead prints a blank space of equal size. Underlining and strikeout, if included in the command definition, will occur as though the characters were actually printed.

Invisible can be useful for table alignment.

Justification

Same as *Justify*.

Justify

Syntax: Justify *Left, No, Off,* 0
Justify *Right*
Justify *Center*
Justify *Both, Yes, On,* 1

This formatting parameter specifies how text should be aligned. As each line completes, or a tab appears, the formatter decides what to do with the text formatted thus far (align it at the left or right, between both margins, or centered between the margins). If you don't specify this parameter, its setting is inherited from the parent format. The default for the outermost page format is *On*; the formatter justifies text at the left and right margins. *Justify Yes, Justify On, Justify Both,* and *Justify 1* have the same effect.

Justify No, Justify Off, Justify Left, and *Justify 0* have the same effect; text is justified only at the left margin.

Justify Center centers text between the margins and turns off filling.

Justify Right aligns text at the right margin and also turns off filling.

If you specify *Justify Center, Right, or Left* and want Sprint to fill paragraphs, you can turn filling back on by following the *Justify* parameter with the *Fill* parameter.

For examples of the *Justify* parameter, see *Center, FlushLeft,* and *FlushRight* in this section.

LeadingSpaces

Syntax: LeadingSpaces *kept* or *ignored*

If this parameter specifies *ignored,* the formatter will ignore any indentation (tabs or space characters) at the start of a paragraph.

LeftIndent

Syntax: LeftIndent (+/–) *distance*

This formatting atribute sets the left edge of the text in a format. You can either set it *absolutely* from the left margin, or (more commonly) *relative* to the enclosing format's left indent by typing a + or – before the dimension. (The left indent can be different than the left margin; the Description format is an example of left indent vs. left margin.)

The Address command definition sets an *absolute* left indent one-half of the entire line length (the left indent is set half-way across the page, measuring from edge to edge):

```
@Define(Address, leftindent.5 line, above 2, below 2,fill
        n,group,initialize="@nohinge")
```

This means that regardless of the left indent set by the parent format (if one exists), the left indent of text in the Address format will be one-half line from the left edge of the paper. Similarly, if the definition included the parameter *LeftIndent .5 inch,* the left indent for text in this format would be 0.5 inch from the left edge of the paper, regardless of the parent format's left indent setting.

The Display format definition sets the left indent *relative* to the current format's left indent. The left indent for text in the Display format is 0.5 inch from the current left indent:

```
@Define(Display = Verbatim, leftindent +.5 in, above 1, below 1,
        group, blanklines hinge)
```

When you set a left indent relative to the current left indent, you don't have to know what the current left indent is. You just specify how far to the left or right of the current indent you want text to begin; + moves the left indent in (to the right) the specified distance, and – moves the left indent out (to the left) the specified distance.

LineLength

Syntax: LineLength *distance*

This formatting parameter specifies the length of a printed line, measured from the current left margin. For example, *LineLength 32 picas* means each line (except the last line of a paragraph) will be 32 picas in length; the right margin will be 32 picas from the left margin.

You can use the parameter *LineLength 0* to turn off wordwrap. For example,

```
@Define(Verbatim, indent 0, spacing 1, fill n, linelength 0, notct)
```

Text typed within the Verbatim format is printed as a single line, until the formatter sees a hard return (*Enter*) character.

Margins

Syntax: Margins (+/–) *distance*

This formatting parameter sets both *LeftIndent* and *RightIndent* at once, to the same absolute or relative values. The Quotation format definition uses the *Margins* parameter to move both the left and right indents in (toward the center of the page) by 0.5 inch:

```
@Define(Quotation, margins +.5 in, above 1, below 1, spacing 1)
```

See the *LeftIndent* and *RightIndent* sections for details on these parameters.

NoTCT

Same as *TCT No.*

Numbered

Syntax: Numbered *Template*

This enumeration parameter specifies that this is a numbered format, which means the formatter will increment the *Counter* each time a paragraph begins. Each paragraph that is separated from the previous one by a blank line (and not indented) will be counted.

If you don't specify a global variable with *Counter*, the formatter uses a *local* variable, which can be accessed by the name *Counter* while you're in the format. This local variable disappears when you end the format. In this case, a string argument to *Numbered* can supply a numbering template. The Numbered and Hyphens format definitions provide an example of this concept:

```
@Define(Numbered, indent -6,above 1,below 1,
        numbered "%&.%]%<%u%;%a%;%i%]",
        BeforeEach "@>()@value(counter).@hsp(.2in)@\")

@Define(Hyphens=Numbered,numbered "%<-%;*%]",BeforeEach
        "@>()@value(counter)@hsp(.2in)@\")
```

Note: If a definition includes the *Numbered* parameter as well as a *Before* or *After* parameter, the formatter won't increment the counter until the text is actually formatted.

Overstruck

Syntax: Overstruck

This typeface parameter is usually included to allow for printers that can't print in bold type. *Overstruck* specifies printing characters twice, in order to make them appear bold. The printer definition may tell Sprint to offset these overstrikes, in which case the resulting text will be wider. You might use this parameter if you sometimes print on a printer that doesn't have a bold typeface. For example,

```
@Define(B, font bold, ifnotfound, overstruck)
```

This is the command definition for *bold,* which tells the formatter to print in bold, and if the formatter doesn't find a bold typeface, it should instruct the printer to overstrike the text to make it darker.

RightIndent

Syntax: RightIndent (+/–) *distance*

This formatting parameter sets the right edge of the text either *absolutely* from the right edge of the paper or *relative* to the enclosing format's right indent. Note that positive numbers move the right indent to the left. For example,

RightIndent +.5 inch

increases the right margin by 0.5 inch; the right indent for text will be 0.5 inch to the left of the parent format's right indent.

RightIndent –.5 inch

moves the right indent 0.5 inch to the right (the right indent is 0.5 inch closer to the right edge than the parent format's right indent).

When you set a right indent relative to the current right indent, you don't have to know what the current right indent is. You just specify how far to the left or right of the current indent you want text to begin; + makes the right indent larger (moves it to the left the specified distance), and – shrinks the right indent (extends the line length by the specified distance).

Script

Syntax: Script *(+/–) distance*

This typeface parameter specifies where the formatter should print text in relation to the baseline of text. For example, when you choose the + Superscript command on the Typestyle menu, the formatter knows that it should print the text .35 of a line above the baseline (the bottom of the current line). The formatter gets this information from the definition of "S" (superscript):

```
@Define(S, font super, ifnotfound, script .35, font small, ifnotfound,
        size .707)
```

The number entered as part of the *Script* parameter can be positive or negative, and is expressed in relation to the baseline. For example, *Script +2* raises and prints text two full lines above the baseline; *Script –.5* instructs the formatter to print the text one-half line below the current baseline. This offset is measured using the height of the font in the *parent* format. Positive is *up*; a negative number is *down*. The default *Script* values for superscript and subscript use +*.35* and –*.25* respectively.

The signed (+ or –) number will be rounded to the nearest vertical printer unit, except that it will always use at least one unit, even if the rounded value is zero.

If the offset (after rounding) exceeds +*.66* or –*.33*, the neighboring lines will be moved away to make room for the script characters. Otherwise, the lines will remain where they are, even though the letters might touch.

Note that the distance used with the *Script* parameter is always in terms of lines. For this reason, you should not include any dimension abbreviation except "line" or "lines."

Size

Syntax: Size *n*

This typeface parameter specifies the point size of characters within the format, if the printer supports font scaling. You can specify size in absolute units, such as inches or points, or you can use lines, scaling relative to the parent format's point size.

The Big format provides an example of the *Size* parameter:

```
@Define(Big=B, font large dwidth, ifnotfound, size 1.414,
        font bold, ifnotfound, overstruck, afterexit "@nohinge")
```

This example means the formatter should try to find a font called *large*, or look up the current font name (in the .SPP file for the printer) and see if there's a font that ends with *.large* (e.g., *Helvetica.large*). If the formatter finds such a font, it uses this font to print the text within the format. If it doesn't find it, it tries the same thing with the *dwidth* font (*large* is a common font on laser printers; *dwidth* is more common among dot matrix printers). If neither font is available, the formatter determines whether the printer supports font scaling. If so, it sets the character size to be 1.414 times larger than it is now, and prints the text in the *bold* font. (In typesetting lore, 1.414 (the square root of 2) is said to be an appropriate size for characters in headings, titles, etc.)

Spacing

Syntax: Spacing *dimension*

This formatting parameter specifies line spacing. This is the space (calculated by font height) between lines in the printed output. The number *2* specifies double spacing (2 lines), *3* specifies triple spacing, 1.5 specifies one and one-half spacing, etc. You can use any valid vertical dimension to specify line spacing; *points* is the typical unit of measure for most desktop publishing applications (see Table 2.1 on page 72 for a complete list of dimensions). There are 72 points per vertical inch, so if you want 6 lines per inch, specify *Spacing 12 points* (72 divided by 6); if you want 8 lines per inch, specify *Spacing 9 points*. If you don't specify the *Spacing* parameter, its setting is inherited from the parent format.

This parameter is useful if your overall page format specifies line spacing other than single. If you want certain formats to be single-spaced, you could include this parameter in the format definition. For example, let's say that you set up a document to have .75 spacing, but you want the page headings to be single spaced. The following definition shows you how PageHead is defined in the STANDARD.FMT file.

```
@Define(PageHead,above page,columns 1,below 2,spacing 1)
```

This definition "floats" the page head (a page heading that affects only the current page) to the top of the page, prints the heading in a single column, inserts two blank lines below the heading, and sets up single-spaced output.

Spread

Syntax: Spread *distance*

This formatting parameter specifies the depth that appears whenever the formatter sees a single blank line. You can use this parameter to create more or less space between paragraphs. If the *Spacing* is 1 and *Spread* is 1, there is a single blank line between the paragraphs. If *Spread* is 2, there are two blank lines between paragraphs; if *Spread* is .5, there is one-half blank line between paragraphs; if *Spread* is 0, there are no blank lines between paragraphs. *Spread 0* is useful for enumerated formats. You must insert blank lines in the file to separate the paragraphs, but if you don't want the blank lines to print, use the *Spread 0* parameter. If you don't specify this parameter, its setting is inherited from the parent format.

The following example shows part of the Index definition:

```
@Define(TheIndex, index, columns 2, spread .5, spacing 1, ...
```

The beginning of this definition specifies a two-column format for index entries. The entries will be single-spaced, but each blank line will be replaced with one-half of one line, rather than the full blank line.

The *Spread* parameter only affects single blank lines. Blank lines that appear after the first one produce full blank lines in the printed file. That is, *N* blank input lines in a row will produce *N-1* plus the *Spread* number in the printed copy. For example,

```
@Define{SillySpace=Numbered, spread 0, spacing 2}
BEGIN SILLYSPACE
Text of item 1.

Text of item 2. This will be a longer item to demonstrate the Spacing 2
parameter. This item will print double-spaced.

Text of item 3.


Text of item 4. We inserted two blank lines between item 3 and item 4.
When the example prints, there will be one blank line (2 minus 1, plus
the spread (0) equals 1 blank line in the output).
END SILLYSPACE
```

This nonsense format is similar to Numbered, except that the output is double-spaced, and the spread (number of blank lines between paragraphs) is zero. The printed example looks like this:

1. Text of item 1.
2. Text of item 2. This will be a longer item to demonstrate the Spacing 2 parameter. This item will print double-spaced.
3. Text of item 3.

4. Text of item 4. We inserted two blank lines between item 3 and item 4. When the example prints, there will be one blank line (2 minus 1, plus the spread (0), equals 1 blank line in the output).

StrikeOut

Syntax: StrikeOut *Off/Alphanumeric/NonBlank/All*

This typeface parameter specifies what should be struck out by printing dashes though it. It works just like *Underline* (see page 426). For example,

```
DEFINE Xout=Example, StrikeOut alphanumeric
BEGIN XOUT
Strikeout this sample text.
END XOUT
```

Prints like this:

~~Strikeout this sample text.~~

TCT

Syntax: TCT *Yes/No*

This macro parameter determines whether *TCT* commands should affect text within this format. For example,

```
@Define(Verbatim, indent 0, spacing 1, fill n, linelength 0, TCT No)
```

This definition tells the formatter to ignore any character translation (TCT) commands for the duration of this format. Once the format ends, the TCT commands resume their effect. For more information on the *TCT* command, see the "TCT" entry in the *Reference Guide.*

Underline

Syntax: Underline *Off/Alphanumeric/NonBlank/All*

This typeface parameter specifies how text should be underlined when printed in this format. For example,

```
@Define(E, font italic uns, ifnotfound, underline nonblank)
```

This is the definition used by the **Italic** command on the **Typestyle** menu. It says to print the text in an italic typeface, and if the formatter can't find this typeface, to underline everything except the blank spaces in the text.

Underline Nonblank tells the formatter to underline everything that's not a blank space; *Underline All* underlines all text, including spaces between words; *Underline Alphanumeric* underlines all letters and numbers, but not punctuation symbols or blank spaces; *Underline Off* turns the underline font off, if the parent format has turned it on. If you don't specify one of the various types of underlining, the formatter will assume *Underline All.*

WithEach

Syntax: WithEach *"string"*

The specified string enables a formatter macro at the start of each paragraph, using the text up to the first tab or hard return, as an argument to the macro. Use the Eval command in the string where you want the text to appear. For example, you can make the Description format print the item in bold, and set the wrap margin right after the item by using the following Modify command:

```
@Modify(Description, WithEach "@B{@eval) @$"})
```

Note: A format definition can contain either one *WithEach* or one *BeforeEach* parameter; these two parameters are mutually exclusive.

Within

Syntax: Within *variable*

This enumeration parameter works in conjunction with the *Numbered* parameter. If you specify *Numbered* in your format definition, the *Within variable* parameter can name a variable that acts as the *parent* to the *Counter* variable. If you don't use this to name a parent, the formatter uses the counter for the enclosing format as the parent.

List of Parameters

Sprint has over 60 different parameters available, all of which are listed in Table D.6.

Note that the Style command is unique in that it sets global settings in a document. Because of this, there are certain parameters that can *only* be used with the Style command. These parameters are listed in Table D.4.

There are also some parameters that can *only* be used with formats that affect regions of text. These commands include any format that starts with the command BEGIN, any that is editable with the **Style/Modify** command, or any command you create yourself using the Define or Modify commands. (You enter the Define and Modify commands using the **Style/Other Format** menu command.) Parameters valid only for these formats are in Table D.3.

A third group of parameters are those that are valid anywhere: both in Style commands and in all other commands. This group of parameters appears in Table D.5.

After the three short tables comes a complete alphabetical list (Table D.6) of all the parameters with short descriptions about each.

Table D.3: Parameters Used Only with Formats Affecting Regions

- Above
- AbovePage
- After
- AfterEntry
- AfterExit
- Before
- BeforeEach
- BeforeExit
- Below
- BelowPage
- BlankLines
- Centered
- Column
- Columns
- Divider
- FlushLeft
- FlushRight
- Free
- Group
- IfNotFound
- Index
- Initialize
- Inline
- Invisible
- LeadingSpaces
- Margins
- Numbered
- Overstruck
- Script
- Strikeout
- Underline
- WithEach
- Within

Table D.4: Parameters Used Only with Style Commands

- BindingOffset
- BottomMargin
- Comments
- Counter
- FormFeed
- Increment
- LeftMargin
- Offset
- Paper
- PaperWidth
- RightMargin
- TabSize
- TopMargin
- WidowPrevent
- WordSpacing

Table D.5: Parameters Used Anywhere

- Fill
- Font
- Gutter
- Indent
- Justification
- Justify
- LeftIndent
- LineLength
- NoTCT
- RightIndent
- Size
- Spacing
- Spread
- TCT

Table D.6: Format Parameters (Complete List)

Field	Description
Typeface Parameters	
Font *name(s)*	Uses the *name* font. The *Font* parameter can be one font name or a list of names. For example, *Font courier pica elite* allows any of these fonts to be used. The first match is the one used.
IfNotFound	Ignores any script, size, overstruck, underline, strikeout, or invisible fields if the most recent font was matched by something from the printer. For example, *Font bold, IfNotFound, Overstruck* will overstrike only if the printer does not have a *bold* font. This command, if at the end of a definition, also prevents the error message that is normally printed if a specified font is not supported by the printer; for example, *@Define(Typewriter, Font courier, IfNotFound).*
Invisible	Does not print the specified text. However, the text still takes up space and gets underscored and struck out if appropriate.
Overstruck	Prints the text once, offsets slightly, and prints again. It is similar to the bold format.
Script {+/–} *dimension*	Moves up or down by the given dimension. (The dimension must be in lines.)
Size *dimension*	Specifies the point size. Size can be given in absolute units such as inches or points, or it can be given in lines (the width of the line depends on the current font's point size). If the dimension is in lines, nesting such formats will cause the point size to grow or shrink geometrically.
Strikeout *type*	The specified text will be struck out. For example, ~~STRIKE-SOMETHING OUT...~~ For a list of acceptable types, refer to *Underline.*
Underline *type*	Underlines the specified text. There are four different types of underline formats: *all* — Everything will be underlined. *alphanumeric* — All letters and numbers will be underlined. *nonblank* — Everything except blanks will be underlined. *off* — No underlining will occur. If no *type* is specified, then *all* will be used.

Table D.6: Format Parameters, continued

Field	Description
Formatting Parameters	
Above *dimension*	At least this much blank space will be put above the format.
AbovePage	"Floats" this format to the top of the page.
Below *dimension*	At least this much blank space will be put below the format.
BelowPage	"Sinks" this format to the bottom of the page.
BlankLines *type*	Defines how the formatter will view blank lines entered in the text. Valid types are *break* Multiple blank lines are ignored. Together they will result in a single *spread* line. *kept* Formats each blank line (opposite of *break*). *Kept* is the default. *hinge* Similar to *break*, but an automatic Hinge command is inserted on each blank line.
BottomMargin *dimension*	This is the space between the end of the text area (which includes the footer if there is one) and the bottom edge of the paper. Default value is 1 inch. Set this value only once, at the beginning of the document, and only as a modifier to the Style command.
Centered	Centers the text within the defined margins (same as *Justify center*.
Column	Sets up a format to print parallel (not snaking) columns. The lines that come after this format start printing at exactly the same spot as the lines in it. (For this reason, you need to set a new left indent after the command that has the *Column* parameter.)
Columns *n*	Divides the page up into this many columns (maximum is 6). If you set *n* equal to 0 (the default value), the text will be formatted into one column.
Comments {*yes/no*}	Tells the formatter whether to hide comments (that is, any line starting with a semicolon) entered in your file. Yes means to omit the comments from the printed version of the file; no means to print the comments. See the Comments entry for instructions on entering comments. The default value of this parameter is *Comments no* and is set on the last line of STANDARD.FMT. Used only as a modifier to the Style command.

Field	Description
Fill {*on/off*}, {*yes/no*}	Turns *Fill* mode on or off for this format. Turning *Fill* on causes the formatter to wordwrap a paragraph, ignoring single hard returns if necessary. Setting *Fill* to off means Sprint always starts a new line if it encounters a single hard return character. (If you do not wish to left justify, you can override this by inserting another justify command after the fill command.)
FlushLeft	Forces all lines to begin at the left margin (same as *Justify left*).
FlushRight	Forces the end of all lines to the right margin (same as *Justify right*).
FooterSpacing *dimension*	Defines the distance between the bottom of the page and the place where the footer begins printing.
FormFeed {*on/off*}	Defines whether the formatter will send form feed characters to the printer to advance the printer to the top of the next page. If you specify *off*, the formatter sends line feed characters to the printer instead of form feeds. You may need to used this parameter if your paper length is not 11 inches (the usual default paper length). Default is *FormFeed on*. Used only as a modifier to the Style command.
Group {*yes/no*}	Groups this format. You can also disable grouping with *Group no*.
Gutter *dimension*	Defines the distance between columns in a multi-column format. Default value is .5 inch.
HeaderSpacing *dimension*	Defines the distance between the top of the page and the place where the header begins printing.
Indent {+/–} *dimension*	Defines the amount of space the first line of a paragraph will be indented (or outdented) relative to the left margin. Default value is 0 (no indent). If this parameter is a positive number (for example, 3 picas), the formatter indents the first line of *every* paragraph by this amount. If you specify a negative number, the first line of every paragraph will be outdented by the specified amount (printed to the left of the remaining text in the paragraph). The *Indent* parameter has the same effect as the Layout/Ruler/Precise Settings/Initial (First Line) Indent command. If an area of text is affected by a command that indents by default, or if you specify an indent value in a Style command and don't want an area of your file indented, modify the command affecting the text that shouldn't be indented.
Justification *type*	Same as *Justify*.

Table D.6: Format Parameters, continued

Field	Description
Justify *type*	Defines the type of justification. Valid types are *left, right, no, yes, off, on, both,* and *center. Right* and *center* types also set the fill command to off. The default value is *Yes;* all paragraphs justified to the left and right margins. If you set this parameter to *No,* Sprint prints with ragged-right margins.
LeadingSpaces *type*	Defines how the formatter will treat indentation by tabs or spaces at the start of a paragraph. Valid types are *kept* Formats each blank space. *ignored* Formats two or more blank spaces as a single blank space. Manual indentation by tabs or spaces at the start of a paragraph is ignored.
LeftIndent {+} *dimension*	Defines the new left margin relative to the current format's left margin. For example, *LeftIndent 1 inch* starts the left margin 1 inch from the previous format's left margin. You cannot use negative numbers with *LeftIndent.*
LeftMargin	Defines the left margin. You can use this parameter only once in a document and only at the top of the file modifying a Style command. The default value is 1 inch.
LineLength {+/–} *dimension*	Defines the length of a line of text (that is, the placement of the right margin relative to the left margin). When the dimension is set to 0, paragraph wordwrap ("fill") does not occur (so long lines can go off the right side of the page).
Margins {+/–} *dimension*	Simultaneously defines both the left and the right margin. For example, *margin .5 inch* creates a left and right margin of 1/2 inch.
NoTCT	Same as *TCT no.*
Offset *dimension*	Adds this much space to the *inner* margin (that is, alternating between the left and right margins) to facilitate binding. Use this parameter only once in a document and only as the first parameter modifying a Style command.
Paper *dimension*	Changes the paper length as defined in the selected printer's printer definition (typically 11 inches). Be sure this parameter matches the form length of the paper in the printer, or the text will "drift" over the pages. **Warning:** Many printers take a *form feed command* and will not adjust to the longer or shorter paper unless the form length switch on the printer is changed to the correct position. If this is necessary with your printer, you can tell Sprint not to use the form feed command by running SP-SETUP and answering *N* to the question *Use Form Feed (Control-L)?* You can also insert a Style command with the *Formfeed No* parameter. Default value is read from the printer definition, usually set at 11 inches.

Field	Description
	Note: Set this value only once, at the beginning of your document, as a modifier to the Style command.
PaperWidth *dimension*	Changes the paper width, as specified in the selected printer's printer definition (default is typically 8-1/2 inches), thus moving the right edge of the text further right or left. The length of the lines increases or decreases without changing the margins. Default value is read from the printer definition, usually set at 8-1/2 inches. Used only as a modifier to the Style command.
RightIndent {+/–} *dimension*	Defines the new right margin relative to the current format's right margin. A positive number moves the indent to the left of the right margin; a negative margin moves it to the right.
RightMargin	Defines the right margin. You can use this parameter only once in a document and only at the top of a file modifying a Style command.
Spacing *dimension*	Changes the distance between each line of the text. Spacing can be given in absolute units such as inches or points, or it can be specified in lines (the depth of the line depends on the current font's point size). Default is 1 line.
Spread *dimension*	Defines the depth of a single blank line. If this value is equal to the *spacing* value, then blank lines in the input look just like blank lines in the output. The default is 1 line. Often the printout looks better if this value is set to less than one line.
TabSize *dimension*	Determines the distance between ASCII tabs. You should only use this value if you're creating an ASCII file and don't have any ruler lines in your file. Default value is 8 characters. Used only as a modifier to the Style command.
TCT {*yes/no*}	Disables/enables TCT translations inside this format.
TopMargin	Defines the top margin. You can use this parameter only once in a document and only at the top of a file modifying a Style command. The default is 1 inch.
WordSpacing *dimension*	Determines the maximum extra space the formatter can insert between words during justification. When justifying text, Sprint stretches the spaces between words first; if this stretching has reached a maximum, and the line is still not justified, Sprint spreads out the letters of individual words. If your printer can handle microspacing between each letter without slowing down considerably, you may want to set this to 2 or 3, so that words will also be stretched. Some people even like to set *WordSpacing* to 1, which inserts space evenly across all the letters and spaces in a line with no special consideration for stretching word spacing first. Default value is *WordSpacing 10,000,* which in effect disables this feature. Used exclusively as a modifier to the Style command.

Field	Description
WidowPrevent {*on/off/N*}	Prevents widows (a partial paragraph at the bottom of a page) and orphans (a partial paragraph at the top of a page). *N* is the minimum number of lines permissible at the bottom or top of a page. *On* is the same as entering 1; *off* is the same as entering 0. Setting this parameter to a large number (like 100) is a good way to prevent paragraphs from ever being split across pages. Used exclusively as a modifier to the Style command.
Enumeration Parameters	
Counter *variable*	Uses the *variable* as the counter (if *variable* is not specified, a "local" counter will be used, which can be referenced by the name *counter*). Used exclusively as a modifier to the Style command. Sets the built-in variable *Counter* to the specified variable for this format only. This command is used mainly to affect the number the formatter assigns to text affected by the **D**efine a Tag and **R**eference a Tag commands. The STANDARD.FMT file, by default, sets this counter to *SectionNumber*, but you can use this parameter to override this *Counter* setting. Note, however, that the formatter doesn't increment each paragraph if you set the counter with a Style command. Basically, *Counter* tells tags what variable to save for later reference. Therefore, *Style counter SectionNumber* causes tags to save the current section number for later reference. Similarly, *Style counter Figure* causes tag to reference the *Figure* variable.
Increment *variable*	Same as *Counter*.
Numbered *"string"*	Increments the counter for each paragraph. *"String"* is an optional template for the counter.
Within *variable*	Sets a global variable to be the "parent" of the counter. (If *variable* is not specified, the enclosing format's counter is used.)
Macro Parameters	
AfterEntry *"text"*	Defines a macro to be executed immediately after the format is started.
AfterExit *"text"*	Defines the macro to be executed in the format that called this one. This command is usually used to print reference numbers to floating environments.
BeforeEach *"text"*	Defines the macro to be executed before each paragraph (as long as the paragraph is separated from the previous paragraph by a blank line and is not indented).

Table D.6: Format Parameters, continued

Field	Description
BeforeExit *"text"*	Defines a macro to be executed at the end of the format.
Divider *"text"*	Executes a macro to produce the "divider line" between a footnote or figure and the regular text. *Divider* is used only by float and sink formats.
Initialize *"text"*	Same as *AfterEntry.*
WithEach *"text"*	Similar to *BeforeEach,* except all leading text up to the first tab is read in and placed in the variable *"text"*. The macro is then executed.
Floating Parameters	
Above	"Floats" this format to the top of the column.
After/Before	Saves text to be printed either at the end of the document or at the very start. The text saved is exactly what was placed in the format call. You can imagine this as a file into which all text is written, then that file is reread by the formatter and formatted. New lines are not appended after each item. *Initialize* and *BeforeExit* are done when the command is formatted.
Below	"Sinks" this format to the bottom of the column.
Free	Formats a "free" format in the column after the current line is finished.
Inline	Turns off any floating switches (*Free, Above, Below,* etc.).
Index	Used in creating indexes. *Initialize* and *BeforeExit* are executed when the format is executed. Refer to the explanation in the *Advanced User's Guide* for details.

APPENDIX

E

Key Codes

There are 512 different key codes (entries in the **key** macro table) available in Sprint. This appendix describes and lists all possible key codes.

Three Types of Key Codes

There are three types of key codes.

The first 256 codes correspond to characters that are actually placed into the buffer. There are three effective subdivisions of these. The first 32 are commonly called *control codes.*. These do not have any printed graphic (although a caret with letter conventially used to represent them, for example,^C), and Sprint and most other programs give them defined imbedded functions (for instance, ^J is a hard return). The 32 (20H) scan code is the space character, and all the other codes are printing characters. Exactly what these codes print is up to the device they are printed on, although there is agreement on the codes from 20H through 7EH, which make up the standard ASCII character set.

The next 128 codes (256-383, or hex 100 to 17F) are *function keys.* Except for putting the numbered function keys on N+256, the assignments here are mostly based on the IBM scan codes.

The last 128 key codes (384-512, hex 180 to 1FF) are the *meta keys.* These are gotten by holding down *Alt* and typing one of the first 128 codes. These should also be produced if there is any "alternative" way to produce an ASCII code. For instance the *Ctrl-M* produces a meta-^M, because the *Enter* key produces the normal ^M. And the keypad numbers and symbols produce meta-numbers, because the top-row numbers produce regular ones. Unless they must make the distinction, programs should pretend these are the regular ASCII codes by subtracting 180H from them.

Modifier Keys

There are three modifier keys on the keyboard:

Shift Used to switch between the printed legends on the keys. If there is only one legend on the key, *Shift* should do nothing (because during rapid typing it is often held down accidentally).

Ctrl Changes the function of a key. It turns X into ^X, makes the *Backspace* and *Enter* keys produce ^? and ^J, and changes the function keys such as the arrows into their *Ctrl* versions. *Ctrl* always overrides *Shift.* If there is no *Ctrl* alternative to a key, *Ctrl* acts like *Alt.*

Alt "Meta-izes" a key if it is in the range 0-127 by adding 180H to it. If the code is not 0-127, *Alt* acts like *Ctrl. Alt* also "shifts" certain keys, so that *Alt-a* is an uppercase Meta-A and *Alt*+numbers is Meta-symbol to avoid conflict with a numeric keypad.

CapsLock simply inverts the *Shift* effect for the A-Z keys, and is not considered a modifier key. *NumLock* is also ignored here.

Key Code Functions

The keys can be divided into these functional areas:

Normal ASCII. A thru Z, @, [, \,], ^, and _ (the *Ctrl*-able characters): these produce the obvious code when typed. If *Ctrl* is held down, the uppercase code XOR'd with 40H is produced, except for ^@, ^H, ^I, ^J, ^M, and ^[, which are automatically "meta'd" to avoid conflict with the *Spacebar, Backspace, Tab, Enter,* and *Esc* keys (so *Alt* has no effect on these).

Special Characters. All the other ASCII letters, symbols, and foreign letters. These produce the obvious code when typed. (*Ctrl* should act like *Alt* on these, but doesn't because the BIOS instead makes them "dead," which is very user unfriendly but unavoidable.)

Control Characters. Keys that produce the correct ASCII control codes: *Esc, Backspace, Del* (*Ctrl-Backspace*), *Enter,* line feed (*Ctrl-Enter*), *Spacebar,* and nul (*Ctrl-Spacebar. Alt* or *Ctrl* (if *Ctrl* is not used to produce the normal code) will produce the Meta-version.

Numbered Function Keys. Numbered function keys produce codes of N+256 (or Function N). The *Shift* keys effectivly add constants to the function number to make many more numbered function keys (a total of 40 on a PC keyboard, and the potential for 64 with the addition of an F0, F11-F15 keys). *Shift* adds 10H, *Ctrl* adds 20H, and *Alt* adds 30H.

Other Function Keys. These include arrow keys, *Ins, Del, Home, End,* and the like. These produce 256 plus the IBM scan code number. Thus the 8-bit stream looks exactly like that read from DOS input. *Shift* has no effect on these keys (unless there is another legend, such as numbers, on them). *Ctrl* and *Alt* both add 40H to them.

Keypad Keys. These are the numbers and symbols (0-9, period, +, –, and any other symbol keys not on the main keyboard. They produce Meta+symbol. *Ctrl, Shift,* and *Alt* have no effect on these keys (unless they have multiple legends on them).

Key Code Table

In the table that follows, the first column is the key code in hex, the second is the name you must give for it in SP.SPM. The third column indicates the keys you should press to generate that character. Aside from the standard ASCII set (up to scan code 7F), a character with nothing in the third column means there is no way to generate that character from the normal IBM keyboard.

An asterisk (*) indicates the IBM BIOS does not produce the code (but it would be nice if it did). A plus sign (+) indicates the code is only produced by the "new BIOS call" that works with the newer 110-key keyboards.

There is some overlap in these key codes; in these cases the keys are indistinguishable as far as key rebinding and the menu reports go. However, the scan code of the last keypress will be available, and the macro for the code can check this to differentiate the identical cases. The scan code for the less common case is in angle brackets; you should check for equality to this code (*do not check for equality with the more common code*). This should only be used if necessary to emulate another word processor.

Table E.1: Key Code Table

NORMAL ASCII

0	^@	(not produced, see 100)
1	^A	*Ctrl-A*
2	^B	*Ctrl-B*
3	^C	*Ctrl-C*
4	^D	*Ctrl-D*
5	^E	*Ctrl-E*
6	^F	*Ctrl-F*
7	^G	*Ctrl-G*
8	^H	*Backspace*
9	^I	*Tab*
A	^J	*Ctrl-Enter*
B	^K	*Ctrl-K*
C	^L	*Ctrl-L*
D	^M	*Enter*
E	^N	*Ctrl-N*
F	^O	*Ctrl-O*
10	^P	*Ctrl-P*
11	^Q	*Ctrl-Q*
12	^R	*Ctrl-R*
13	^S	*Ctrl-S*
14	^T	*Ctrl-T*
15	^U	*Ctrl-U*
16	^V	*Ctrl-V*
17	^W	*Ctrl-W*
18	^X	*Ctrl-X*
19	^Y	*Ctrl-Y*
1A	^Z	*Ctrl-Z*
1B	^[	*Esc*
1C	^\	*Ctrl-*
1D	^]	*Ctrl-]*
1E	^^	*Ctrl-6*
1F	^_	*Ctrl—*
20		*Spacebar*
21	!	
22	"	
23	#	
24	$	
25	%	
26	&	
27	'	
28	(	
29	)	
2A	*	
2B	+	
2C	,	
2D	–	
2E	.	
2F	/	
30	0	
31	1	
32	2	
33	3	
34	4	
35	5	
36	6	
37	7	
38	8	
39	9	
3A	:	
3B	;	
3C	<	
3D	=	
3E	>	
3F	?	
40	@*	
41	A	
42	B	
43	C	
44	D	
45	E	
46	F	
47	G	
48	H	
49	I	
4A	J	
4B	K	
4C	L	
4D	M	
4E	N	
4F	O	
50	P	
51	Q	
52	R	
53	S	
54	T	
55	U	
56	V	
57	W	
58	X	
59	Y	
5A	Z	
5B	[	
5C	\	
5D	]	
5E	^	
5F	_	
60	‘	
61	a	
62	b	
63	c	
64	d	

Table E.1: Key Codes, continued

65	e	
66	f	
67	g	
68	h	
69	i	
6A	j	
6B	k	
6C	l	
6D	m	
6E	n	
6F	o	
70	p	
71	q	
72	r	
73	s	
74	t	
75	u	
76	v	
77	w	
78	x	
79	y	
7A	z	
7B	{	
7C	|	
7D	}	
7E	~	
7F	^?	

FOREIGN LETTERS

80	Ç	*Alt-128*
81	ü	*Alt-129*
82	é	*Alt-130*
83	â	*Alt-131*
84	ä	*Alt-132*
85	à	*Alt-133*
86	å	*Alt-134*
87	ç	*Alt-135*
88	ê	*Alt-136*
89	ë	*Alt-137*
8A	è	*Alt-138*
8B	ï	*Alt-139*
8C	î	*Alt-140*
8D	ì	*Alt-141*
8E	Ä	*Alt-142*
8F	Å	*Alt-143*
90	E	*Alt-144*
91	æ	*Alt-145*
92	Æ	*Alt-146*
93	ô	*Alt-147*
94	ö	*Alt-148*
95	ò	*Alt-149*
96	û	*Alt-150*
97	ù	*Alt-151*
98	ÿ	*Alt-152*
99	Ö	*Alt-153*
9A	Ü	*Alt-154*
9B	¢	*Alt-155*
9C	£	*Alt-156*
9D	¥	*Alt-157*
9E	P_t	*Alt-158*
9F	ƒ	*Alt-159*
A0	á	*Alt-160*
A1	í	*Alt-161*
A2	ó	*Alt-162*
A3	ú	*Alt-163*
A4	ñ	*Alt-164*
A5	Ñ	*Alt-165*
A6	ª	*Alt-166*
A7	º	*Alt-167*
A8	¿	*Alt-168*
A9		*Alt-169*
AA	¬	*Alt-170*
AB	1/2	*Alt-171*
AC	1/4	*Alt-172*
AD	¡	*Alt-173*
AE	«	*Alt-174*
AF	»	*Alt-175*
B0		*Alt-176*
B1		*Alt-177*
B2		*Alt-178*
B3		*Alt-179*
B4		*Alt-180*
B5		*Alt-181*
B6		*Alt-182*
B7		*Alt-183*
B8		*Alt-184*
B9		*Alt-185*
BA		*Alt-186*
BB		*Alt-187*
BC		*Alt-188*
BD		*Alt-189*
BE		*Alt-190*
BF		*Alt-191*
C0		*Alt-192*
C1		*Alt-192*
C2		*Alt-194*
C3		*Alt-195*
C4		*Alt-196*
C5		*Alt-197*
C6		*Alt-198*
C7		*Alt-199*
C8		*Alt-200*

Table E.1: Key Codes, continued

C9		*Alt-201*
CA		*Alt-202*
CB		*Alt-203*
CC		*Alt-204*
CD		*Alt-205*
CE		*Alt-206*
CF		*Alt-207*
D0		*Alt-208*
D1		*Alt-209*
D2		*Alt-210*
D3		*Alt-211*
D4		*Alt-212*
D5		*Alt-213*
D6		*Alt-214*
D7		*Alt-215*
D8		*Alt-216*
D9		*Alt-217*
DA		*Alt-218*
DB		*Alt-219*
DC		*Alt-220*
DD		*Alt-221*
DE		*Alt-222*
DF		*Alt-223*
E0	α	*Alt-224*
E1	β	*Alt-225*
E2	Γ	*Alt-226*
E3	π	*Alt-227*
E4	Σ	*Alt-228*
E5	σ	*Alt-229*
E6	µ	*Alt-230*
E7	τ	*Alt-231*
E8	Φ	*Alt-232*
E9	Θ	*Alt-233*
EA	Ω	*Alt-234*
EB	δ	*Alt-235*
EC	∞	*Alt-236*
ED	∅	*Alt-237*
EE	∈	*Alt-238*
EF	∩	*Alt-239*
F0	≡	*Alt-240*
F1	±	*Alt-241*
F2	≥	*Alt-242*
F3	≤	*Alt-243*
F4	¶	*Alt-244*
F5	§	*Alt-245*
F6	÷	*Alt-246*
F7	≈	*Alt-247*
F8	°	*Alt-248*
F9	•	*Alt-249*
FA	·	*Alt-250*
FB	√	*Alt-251*
FC	^{n}	*Alt-252*
FD	^{2}	*Alt-253*
FE		*Alt-254*
FF		*Alt-255*

FUNCTION KEYS

100	F0	*Ctrl-space*
101	F1	F1
102	F2	F2
103	F3	F3
104	F4	F4
105	F5	F5
106	F6	F6
107	F7	F7
108	F8	F8
109	F9	F9
10A	F10	F10
10B	F11	F11^{+}
10C	F12	F12^{+}
10D	F13	
10E	F14	
10F	F15	
110	F16	
111	F17	*Shift-F1*
112	F18	*Shift-F2*
113	F19	*Shift-F3*
114	F20	*Shift-F4*
115	F21	*Shift-F5*
116	F22	*Shift-F6*
117	F23	*Shift-F7*
118	F24	*Shift-F8*
119	F25	*Shift-F9*
11A	F26	*Shift-F10*
11B	F27	*Shift-F11*$^{+}$
11C	F28	*Shift-F12*$^{+}$
11D	F29	
11E	F30	
11F	F31	
120	F32	
121	F33	*Ctrl-F1*
122	F34	*Ctrl-F2*
123	F35	*Ctrl-F3*
124	F36	*Ctrl-F4*
125	F37	*Ctrl-F5*
126	F38	*Ctrl-F6*
127	F39	*Ctrl-F7*
128	F40	*Ctrl-F8*
129	F41	*Ctrl-F9*
12A	F42	*Ctrl-F10*
12B	F43	*Ctrl-F11*$^{+}$
12C	F44	*Ctrl-F12*$^{+}$

Table E.1: Key Codes, continued

12D	F45	
12E	F46	
12F	F47	
130	F48	
131	F49	*Alt-F1*
132	F50	*Alt-F2*
133	F51	*Alt-F3*
134	F52	*Alt-F4*
135	F53	*Alt-F5*
136	F54	*Alt-F6*
137	F55	*Alt-F7*
138	F56	*Alt-F8*
139	F57	*Alt-F9*
13A	F58	*Alt-F10*
13B	F59	*Alt-F11*[+]
13C	F60	*Alt-F12*[+]
13D	F61	
13E	F62	
13F	F63	
140	F40H	*mouse left*
141	F41H	*mouse right*
142	F42H	*mouse middle*
143	F43H	*release of any mouse button*
144	F44H	*mouse left dbl*
145	F45H	*mouse right dbl*
146	F46H	*mouse middle dbl*
147	F47H	*Home*
148	F48H	*Up arrow, mouse up*
149	F49H	*PgUp*
14A	F4AH	*mouse left*
14B	F4BH	*Left arrow*
14C	F4CH	*Center*[+]
14D	F4DH	*Right arrow*
14E	F4EH	*mouse right*
14F	F4FH	*End*
150	F50H	*Down arrow, mouse down*
151	F51H	*PgDn*
152	F52H	*Ins*
153	F53H	*Del*
154	F54H	
155	F55H	
156	F56H	
157	F57H	
158	F58H	
159	F59H	
15A	F5AH	
15B	F5BH	
15C	F5CH	
15D	F5DH	
15E	F5EH	
15F	F5FH	
160	F60H	*Ctrl-Alt-mouse left*
161	F61H	*Ctrl-Alt-mouse right*
162	F62H	*Ctrl-Alt-mouse middle*
163	F63H	
164	F64H	*Ctrl-Alt-mouse left dbl*
165	F65H	*Ctrl-Alt-mouse right dbl*
166	F66H	*Ctrl-Alt-mouse middle dbl*
167	F67H	*Ctrl-Alt-Home*
168	F68H	*Ctrl-Alt-Up*[+]
169	F69H	*Ctrl-Alt-PgUp*
16A	F6AH	
16B	F6BH	*Ctrl-Alt-Left*
16C	F6CH	*Ctrl-Alt-Center*[+]
16D	F6DH	*Ctrl-Alt-Right*
16E	F6EH	
16F	F6FH	*Ctrl-Alt-End*
170	F70H	*Ctrl-Alt-Down*[+]
171	F71H	*Ctrl-Alt-PgDn*
172	F72H	*Ctrl-Alt-Ins*[+]
173	F73H	*Ctrl-Alt-Del*[+]
174	F74H	
175	F75H	
176	F76H	
177	F77H	
178	F78H	
179	F79H	
17A	F7AH	
17B	F7BH	
17C	F7CH	
17D	F7DH	
17E	F7EH	
17F	F7FH	
		META KEYS
180	~^@	*Ctrl-2<3>, Ctrl-Alt-Space*[+]
181	~^A	*Ctrl-Alt-a*[+]
182	~^B	*Ctrl-Alt-b*[+]
183	~^C	*Ctrl-Alt-c*[+]
184	~^D	*Ctrl-Alt-d*[+]
185	~^E	*Ctrl-Alt-e*[+]
186	~^F	*Ctrl-Alt-f*[+]
187	~^G	*Ctrl-Alt-g*[+]
188	~^H	*Ctrl-h<23>, Alt-Backspace*[+]
189	~^I	*Ctrl-i<17>, Ctrl*[+]*/Shift/Alt*[+]*-Tab*

18A	~^J	Ctrl-j<24>, Ctrl-Alt-enter*
18B	~^K	Ctrl-Alt-k$^+$
18C	~^L	Ctrl-Alt-l$^+$
18D	~^M	Ctrl-m<32>, Alt-Enter<1C>$^+$, keypad Enter$^+$
18E	~^N	Ctrl-Alt-n$^+$
18F	~^O	Ctrl-Alt-o$^+$

190	~^P	Ctrl-Alt-p$^+$
191	~^Q	Ctrl-Alt-q$^+$
192	~^R	Ctrl-Alt-r$^+$
193	~^S	Ctrl-Alt-s$^+$
194	~^T	Ctrl-Alt-t$^+$
195	~^U	Ctrl-Alt-u$^+$
196	~^V	Ctrl-Alt-v$^+$
197	~^W	Ctrl-Alt-w$^+$
198	~^X	Ctrl-Alt-x$^+$
199	~^Y	Ctrl-Alt-y$^+$
19A	~^Z	Ctrl-Alt-z$^+$
19B	~^[	Ctrl-[<1A>, Ctrl-Alt-Esc
19C	~^\	Ctrl-Alt-\$^+$
19D	~^]	Ctrl-Alt-]$^+$
19E	~^^	Ctrl-Alt-6$^+$
19F	~^_	Ctrl-Alt--$^+$
1A0	~space	Alt-space
1A1	~!	Alt-1
1A2	~"	Alt-Shift-'*
1A3	~#	Alt-3
1A4	~$	Alt-4
1A5	~%	Alt-5
1A6	~&	Alt-7
1A7	~'	Alt-'$^+$
1A8	~(	Alt-9
1A9	~)	Alt-0
1AA	~*	keypad *, Alt-8<9>
1AB	~+	keypad +
1AC	~,	keypad ,
1AD	~-	keypad –
1AE	~.	keypad .
1AF	~/	keypad /$^+$
1B0	~0	keypad 0
1B1	~1	keypad 1
1B2	~2	keypad 2
1B3	~3	keypad 3
1B4	~4	keypad 4
1B5	~5	keypad 5
1B6	~6	keypad 6
1B7	~7	keypad 7
1B8	~8	keypad 8
1B9	~9	keypad 9
1BA	~:	Alt-Shift-;*
1BB	~;	Alt-;$^+$
1BC	~<	Alt-,$^+$
1BD	~=	Alt-=$^+$<D>
1BE	~>	Alt-.$^+$
1BF	~?	Alt-/$^+$
1C0	~@	Alt-2
1C1	~A	Alt-a
1C2	~B	Alt-b
1C3	~C	Alt-c
1C4	~D	Alt-d
1C5	~E	Alt-e
1C6	~F	Alt-f
1C7	~G	Alt-g
1C8	~H	Alt-h
1C9	~I	Alt-i
1CA	~J	Alt-j
1CB	~K	Alt-k
1CC	~L	Alt-l
1CD	~M	Alt-m
1CE	~N	Alt-n
1CF	~O	Alt-o
1D0	~P	Alt-p
1D1	~Q	Alt-q
1D2	~R	Alt-r
1D3	~S	Alt-s
1D4	~T	Alt-t
1D5	~U	Alt-u
1D6	~V	Alt-v
1D7	~W	Alt-w
1D8	~X	Alt-x
1D9	~Y	Alt-y
1DA	~Z	Alt-z
1DB	~[	Alt-[$^+$
1DC	~\	Alt-\$^+$
1DD	~]	Alt-]$^+$
1DE	~^	Alt-6
1DF	~_	Alt--
1E0	~'	Alt-'$^+$
1E1	~a	Shift-Alt-a
1E2	~b	Shift-Alt-b
1E3	~c	Shift-Alt-c
1E4	~d	Shift-Alt-d
1E5	~e	Shift-Alt-e
1E6	~f	Shift-Alt-f
1E7	~g	Shift-Alt-g
1E8	~h	Shift-Alt-h
1E9	~i	Shift-Alt-i
1EA	~j	Shift-Alt-j
1EB	~k	Shift-Alt-k
1EC	~l	Shift-Alt-l
1ED	~m	Shift-Alt-m
1EE	~n	Shift-Alt-n

1EF	~o	Shift-Alt-o	1F1	~q	Shift-Alt-q
1F0	~p	Shift-Alt-p	1F2	~r	Shift-Alt-r
1F3	~s	Shift-Alt-s	1FA	~z	Shift-Alt-z
1F4	~t	Shift-Alt-t	1FB	~{	Shift-Alt-[+
1F5	~u	Shift-Alt-u	1FC	~|	Shift-Alt-\+
1F6	~v	Shift-Alt-v	1FD	~}	Shift-Alt-]+
1F7	~w	Shift-Alt-w	1FE	~~	Shift-Alt-‘*
1F8	~x	Shift-Alt-x	1FF	~^?	Ctrl-/* <35>, Ctrl-Alt-Backspace*
1F9	~y	Shift-Alt-y			

APPENDIX

F

Build Your Own Printer and Screen Drivers

Some companies would be satisfied with having a programmable editor, but Borland is expected to have more; and we do. Before you read on, we want to warn you that this chapter has a great deal of technical information—most of which can be ignored. You only need to read this section if you must

- print on a device (printer or typesetter) that Sprint doesn't support
- send output to a port Sprint doesn't support
- use a screen adapter, or even a terminal that Sprint doesn't support

Sprint has an "open architecture." This means that there are facilities built into Sprint to allow adding support for many different types of hardware. You can control this configuration; there is no need to write "device drivers," or to wait for Borland to supply one.

Most programs that can be configured require the company that makes the program to have the same equipment you do and to write an interface program for it. Often this interface program must be linked into their program, resulting in a completely different version of the program.

Sprint, on the other hand, lets the *user* do the configuration—no programming or linking is required. Instead, you can use the Sprint editor to write a description of the device (the *library* file), and then run the SP-SETUP program, which converts this ASCII description into a binary file. The editor and formatter are able to read these binary files, and use them to control the device.

Therefore, if you have an unusual system, you don't have to give up on it just because it isn't "standard," nor do you have to wait until a new version of Sprint is released—you can do it yourself!

Configuration is *not* trivial, but it is not extremely difficult either. Admittedly, it helps to have had some programming experience before you attempt it. This appendix describes how to make your own library files for Sprint.

The SP-SETUP Program

When you installed Sprint for your computer and printer, you used the Sprint setup program, SP-SETUP.EXE. What you didn't see when running SP-SETUP was its ability to automatically adapt itself to new printer, screen, and port definitions as they are added to its database.

When run, SP-SETUP reads the database file (MAIN.SPL) that contains descriptions of devices and how to control them. It then presents a list of the devices it found, and allows the user to choose from the list. Once the user chooses a device, the device's description is used to create a *binary* representation of the description in the database. These binary descriptions are then used by the editor and formatter to control the devices.

The database that SP-SETUP uses is often referred to as a *library* file, and has an extension of .SPL (Sprint Library). It is a text file to which more devices can be added by using Sprint to add the required information.

After the necessary binary files are created using SP-SETUP, SP-SETUP and the library files are not needed to run Sprint, and can be removed from your work disk or hard disk to provide more storage.

There are three types of devices that the user can define for Sprint:

Screens
: Screens are descriptions of display adapters and terminals. The driver for this is contained in the file DEFAULT.SPS. This file contains a description of the screen and the colors (or attributes) to be used. Alternative drivers can exist in other files as *name*.SPS. Using this capability, for example, you can have two drivers for your computer with an EGA card, one supporting the 43-line mode, and another for 25-line mode. You can tell Sprint to use the alternative driver by starting Sprint with "`SP -s=`*`fname`*".

  If no DEFAULT.SPS file is found by the editor on startup, it will automatically detect the type of screen adapter and use its built-in description for it.

Printers

Printers are descriptions of output devices. These output devices may be dot-matrix, daisy wheel, or laser printers, or typsetters. In Sprint, we refer to all these as printers. The printer description includes information about all the fonts the printer can use, the size of the letters in those fonts, and all the attributes (such as boldface) available. The driver for the default printer is written to the file DEFAULT.SPP. Other printer descriptions are written to *name*.SPP. The printer to use can be selected from the editor, the formatter command line, or with the @Device formatter command within a file.

If no DEFAULT.SPP file, is found by the formatter, and no alternate is specified, Sprint will assume a plain printer with no special features.

Ports

Ports are descriptions of how Sprint can talk to an external terminal or printer. Usually, Sprint sends codes to these devices using the normal operating system interface to them. Sometimes DOS may not support the port, or the port requires different handshaking than that provided by DOS. A port describes how to run the actual computer hardware (usually a serial communication chip).

The binary version of a port record, if one is selected, is added to the binary version of the screen or printer record, so this does not create a new file. Ports are selected from the SP-SETUP program.

NOTE: Ports for screens are requested by the install program only if a screen definition does not use the BIOS for cursor positioning. In this case, it assumes that the installation is for an external terminal.

The Library File

SP-SETUP reads its data from a *library* file. These files must have the extension .SPL (for SPrint Library). SP-SETUP reads its definitions from the file MAIN.SPL. As shipped, Sprint comes with many additional library files, including HP.SPL, POSTSCR.SPL, and DIABLO.SPL. These library files contain definitions for the HP Laserjet series, the Apple LaserWriter (and other PostScript printers), and DIABLO (and compatible) printers. These library files are automatically read into SP-SETUP and used to provide definitions for these printers.

Library Records

The library file contains ASCII *records* for each defined device. Every record starts with the type of record, (that is, screen, printer, or port), the name of the device, and finally, specific information describing the device. (There are also special *sub-records* used for printer descriptions. They are *font, attr, tct, and pst*. These records are added to selected printers.)

Because a single manufacturer can have many types of printers, a record can contain a *root name* field (name is optional—if not given, the record name will be used). Root records appear on the device menu, and when chosen, cause another device menu to display all records named *name.subname,* from which the actual device can be chosen:

```
printer Hewlett-Packard,root HP
```

The information used to describe the device is made up of *fields*. Each field has a *name* which must be first, a space, and then a *value*. The value can be a string, a number, or a yes/no value, depending on the type of the field. There must be commas between the fields to separate them. The last field in a record has no comma after it.

A record can contain many lines. All but the first line must be indented. The first line of a record must be at the left margin.

Comments start with *double* semicolons, and extend through the end of the line. We highly recommend that you put lots of comments in your descriptions; they won't be clear otherwise. Comments that follow a record's name, yet appear before the last line of a record, *and* start in the first column, will be displayed when the particular device is selected. This allows important printer-dependent information to be conveyed.

Here is a sample record, that of the Hewlett-Packard ThinkJet printer:

```
printer HP.ThinkJet.IBM Mode,flag2,
;; This driver is for the HP ThinkJet printer, in IBM mode.
   vpi 72,size 12,cr^M,nl ^M^J,ff^L,
   svp^[A%c^[2,mvm 85,hpi 600,
   init^[O^[G,reset^M
   font Elite,width 50,on^R^T
   font Cpi6,width 100,on^R^N,off^T
   font Cpi10,width 56,on^O^N,off^R^T
   font Cpi21,width 28,on^R^O,off^R
   attr bold,on^[E,off^[F
   attr uns,on^[-1,off^[-0
```

The first line indicates that this is a *printer* record, and the name of this printer is HP.ThinkJet.IBM Mode. The second line is a comment which will be written to the screen if the user selects this printer. The next three lines have the fields that describe the character sequences and dimensions used by the ThinkJet printer.

This example also illustrates several *sub-records* for the HP ThinkJet. These sub-records describe the Elite font, several different font sizes, and two font attributes (bold, and underline).

Since there is much overlap among different devices, records can also indicate that they resemble another record of the same type, by using the *as* field. Then any field not specified in this record is copied from the *as* record:

```
printer Okidata.84,as Okidata.92
```

The above example defines a printer called Okidata.84 to be exactly like an Okidata.92.

Records may indicate that they are stored in a different file by using the *in* field. You cannot use any other fields along with *in*. The record must appear (with exactly the same name) in the other file:

```
printer Apple, in apple.spl
```

This example tells SP-SETUP that the definition for Apple printers will be found in the file APPLE.SPL

(Note: You cannot use *as* to reference printers defined in a different file.)

Separate files have two advantages: (1) they are faster to configure from, because you have to search only the other file for font tables, and (2) auxiliary records (such as font tables for a printer) can have names that duplicate the names used in the main file.

Yes/No Fields

If a field takes a yes/no value, it can be followed either by any word starting with "Y" or "N"(we use Yay, nay and YES, NO). Or, if the field is immediately followed by a comma SP-SETUP will default to yes. (Flags can also be followed by a number from 0 to 255; this feature is reserved for future use.

Numeric Fields

If a field takes a numeric value, the number can be given in decimal after the field. The number can also be given in hexadecimal by following it with an *H,* or in octal by following it with the letter *O.* The largest number allowed is 32,767.

Dependent Files

Certain screen or printer drivers are dependent on other files in order to operate. To meet these needs, and ensure the files are copied when the installation calls for it, four dependency file fields exist. They are named DFILE1, DFILE2, DFILE3, and DFILE4. Any driver that requires another file (or files), can list the names as DFILES within the driver, and SP-SETUP will automatically copy the files when the user chooses one of these.

The Hercules InColor Card requires a font file to be downloaded to it. This is normally accomplished by the file SPHERC.COM, which sends the font file SPHERC.FNT. To ensure that the files are installed on the hard disk when the user selects this driver, we made the definition look like:

```
screen Hercules-InColor, as Monochrome, init Cspherc spherc.fnt,
  dfile1 SPHERC.COM, dfile2 SPHERC.FNT...
```

String Fields

String fields are somewhat more complicated. Leading blanks are skipped, and the string is taken to be all the characters up to the comma, a comment, or the end of the line.

Since this does not allow you to put all possible characters in the string, there are special ways to construct the other characters.

You can get any control character by typing a caret (^) followed by a letter. For instance, ^X will put a Control-X (ASCII code 24) into the string. Code 127 (DEL) can be gotten with ^?. *Be careful not to put* real *control codes in the .SPL file. SP-SETUP can't read them.*

You can get characters with decimal values greater than 127 by putting a tilde (~) before them. For instance, ~X is the code 216 (128 + 88), and ~^X is the code 152. See the Appendix G for a decimal number to code letter conversion.

The backslash (\) is used as an *escape* character, to put other codes into the string that can't be put in otherwise. You need it before any character that is interpreted specially, and also to name some common control characters. Users of the C programming language will find this very familiar:

\,	– comma
\^	– caret
\~	– tilde
\\	– back slash
\;	– semicolon (\;; does not start a comment)
\<sp>	– leading or trailing spaces
\e	– escape (^[)
\t	– tab (^I)
\b	– backspace (^H)
\r	– return (^M)
\n	– line feed (^J)
\f	– form feed (^L)
\nnn	– any code in decimal notation
\xnn	– any code in hex notation

You cannot put NUL (^@ or ASCII code 0) into any string (C programmers will know why). If you want to send a null to a device, you must use the %z printf command (which we'll soon describe).

Character Fields

Some fields (especially the printer translation tables) have values that are one character. These are simply one-character strings. If you want to put in a specific decimal code, use the \nnn method.

Printf Strings

The formatter and editor programs send commands to devices using *printf* strings. These strings let you specify how a numerical value is sent to the device.

A printf string contains text and *placeholders* for information to be included within the text. These placeholders describe where information being passed to it gets inserted, and in what form it should be inserted (hex, decimal, character, and so on.)

These strings are an extension of the *printf* string format used in the C programming language. Any character except % in the string is sent to the printer unchanged. % indicates a place where one of the *arguments* to the string is to be used. Each argument is used in order, for instance the *cur* string in a screen definition takes two arguments, the line number and column number, so the first % encountered uses the line number, and the second one uses the column number.

%c — The number is turned into a character and printed.

%*n*c — Print *n*-2 nulls, then an integer as two characters, high byte first (*n* = 2 is most useful).

%*n*w — Print an integer as two characters, low byte, then high byte, followed by n-2 nulls.

%u — The number is turned into an unsigned decimal number, and the digits printed.

%d — The number is turned into signed decimal and printed.

%*n*d — The number is turned into signed decimal and printed in a field *n* characters wide. (Use %0*n* to pad with 0's.)

%*n*u — The number is turned into decimal in a fixed field width of *n*. *N* is a number typed into the string, as in %4u. If the number *n* starts with a zero (e.g %04u), the field is padded with leading zeros, otherwise it is padded with leading spaces.

%(...%) — The text between the %(and the %) is repeated that number of times. Zero also works.

%[...%;...%;
...%:...%] — This is a case statement. Negative numbers and zero print the first case (the part between the %[and the first %;), 1 prints the second case, 2 prints the third, and so on. If there is a %:, this indicates a default case that is printed for all remaining numbers. If there is no %:, a number larger than the number of cases given wraps around to the first, second, etc.

You can put other % commands inside the cases, in which case the same variable is reused for them. Cases cannot be nested.

Each time you do one of the above operations, the *argument pointer* advances. So if there are multiple arguments, each successive % command prints the subsequent one.

An argument can be modified by inserting any number of the following math operations between the % and the above operations: this math is done using signed 16-bit integers, even for %u. In all these cases, *n* is 1 if it is not given.

n+ — Add the constant *n* to the argument.

n- — Subtract *n* from the argument.

n*   Multiply the argument by *n*.

n/   Divide it by *n*.

n\\   Unsigned divide by *n* (result is positive for numbers greater that 32767).

n>   Shift the argument right by *n* (divide it by 2^n). Shifts are trivially faster than multiplications.

n<   Shift the argument left by *n* (multiply by 2^n).

n#   Get the remainder of dividing the argument by *n* (otherwise known as taking the argument *modulus n*).

n&   AND the binary bits in the argument with *n*.

n | OR the bits with *n*.

n^   XOR (exclusive or) the bits with *n*. This toggles all the bits in the argument when there is a one bit in *n*.

~   Complement the argument (toggle all the bits). You can also get the negative value of the argument by using ~+.

Example: %14+25*-c takes the current argument, adds 14, then multiplies the result by 25, then subtracts 1 (because the default for *n* is 1), then prints the result as a single character.

And another example: If X is the argument, and you need to print round(X/4)+1 (a common occurrence on some printers), you would use something like %2+4/+u, which first adds 2 in order that the integer division rounds the result, then divides by 4, then adds 1, then prints the result in ASCII digits.

The following operations don't use any of the arguments. Any math operations before them are ignored.

%%   Print a %.

%ng   This lets you rearrange the arguments as needed, by switching which one is used next. %0g goes to the first argument, %1g goes to the second, etc.

To reverse the order of coordinates passed for a cursor positioning command, you would use: %1g%c%0g%c

%nz   Sends *n* nulls.

Do not use any other % combinations, such as %s or %r. These are reserved for internal use by the formatter, and will cause unpredictable results if they are put in a configuration string.

It is important to realize that the SP-SETUP program does not pay any attention to the % commands. As far as it knows, % is a regular character, and is not treated specially. Errors with the % commands will not be detected until the editor is run, and even then, they will just be detected as odd behavior.

Hardware Control Strings

Hardware control strings are used in some of the screen definitions, and are also interpreted by the **hardware** editor macro. They are designed to allow the editor to perform hardware-level control of the computer. They allow three things to be done:

- access memory locations
- access I/O ports
- perform software interrupts to call ROM functions

Hardware strings are interpreted from left to right, and can contain numbers and operator symbols. Spaces serve to separate words, but are otherwise ignored. At any time, a single "argument" is preserved; some operators set this argument, others use it.

Numbers must start with a digit, and can end with *H* or *O* or *B* to indicate hex, octal, or binary (otherwise they are decimal). If a number is encountered in the string, the argument is set to it.

Operators:

"%" Sets the argument to a code off the "argument list." For the **hardware** editor macro, this can be done once. Commands in hardware descriptions can take this more than once, it depends on which string is being defined.

"* address" Sets the argument to the contents of a given byte of memory. The address can be a single number, indicating something in segment zero, or it may be "number:number" indicating a segment and offset. Don't forget to put H on the end of the numbers if you want hex addresses.

"> address" Sets the given byte to the argument. The argument does not change.

">| address" OR's the argument with the contents of the byte. This can be used to set various bits.

">& address" AND's the argument with the contents of the byte. This turns off bits.

">^ address" XOR's the argument with the contents of the byte. This toggles bits.

"> reg" Sets a given register to the argument. These register values are used during the next interrupt. Legal registers are AH, AL, AX, BH, BL, BX, CH, CL, CX, DH, DL, DX, SI, DI. (Note you can't set the segment or BP registers.)

"in number" Sets the argument to the input from the given i/o port.

"out number" Sends the argument to the given I/O port.

"int number" Does an interrupt. The argument is put in the AH register. The other registers are set as per the most recent "> reg" instructions, then an int instruction is done. The argument is set to whatever is in AX when the interrupt returns.

As an example, you can change the cursor to a dash with:

```
"5>ch 6>cl 1 int 10h"
```

Screen Descriptions

A *screen* record fully describes how Sprint accesses the display screen on the computer. Since all information concerning how to run a screen is in this file, you should be able to get any arrangement of hardware to operate, including larger screens than those normally sold, external terminals, even devices for the handicapped. Sprint will work with screens of up to 120 lines and 255 columns.

Terminals are generally operated by sending *escape sequences* (special code sequences starting with the ^[or ESC character) to them. Unfortunately, escape sequences are not standardized, and the only attempt at a standard (ANSI) is a very poor and slow one, requiring excessive sequence length and computation.

To allow for both screen adapters and external terminals of various kinds, Sprint can use any combination of escape sequences—sent to the MS-DOS standard output in *raw* mode, IBM PC BIOS calls, and "direct memory mapping" of the character output into an in-memory array that matches the screen display (which is faster than the other methods).

Screen descriptions can also reference a output port to send escape sequences to, in case it is impossible to redirect the standard output to an external device.

The following fields are supported in a screen record. Some of them are required, such as *rows* and *cols*, others are not needed, such as *up*, but the screen update will be faster if they are provided.

^A to ^_ (numbers) Entries for attribute 1 through attribute 31. The default for all the printing ones is dash or space, for all the attributes it is the same as *select*.

clreol (string) This string is sent by the editor to clear from the cursor, to the end of line. If not given, the editor will clear using spaces.

cols (number, required) The number of columns on the screen. Most have 80, although a few have 132. The editor has a maximum of 255. 40 is the minimum. If this is a memory-mapped screen, Sprint assumes each row on the screen takes this many locations (that is, each row is this many words apart).

On terminals, if it does *line wrap* and you can't turn it off with an initialization string, you will have to set this number to 1 less than the maximum. Otherwise the screen will scroll when the status line is drawn, destroying the display.

cur (string, two arguments, required) This string is used to position the cursor. The first argument is the row, the second is the column. The upper left corner is 0,0.

If you give a capital *B* for this string, it will attempt to move the cursor using IBM Video BIOS call number 2, and to print all characters using IBM BIOS call number 9 (unless *map* is given as well). This will disable all other strings except *init* and *reset*.

dc (string, one argument) Delete *N* characters at the cursor, moving the characters to the right *N* left. This should add blanks at the right end of the line, and *not* move the cursor. (This string is not used in the current version of the Sprint editor.)

dl (string, one argument) Delete *N* lines at the row the cursor is on, moving all lower lines up, and adding *N* blank lines at the bottom.

down (string, four arguments) This is the string to send to stdout to scroll a region down. Takes the same four arguments as *up*. A capital *B* causes the BIOS call to be used. (Note that some clones have this call broken; Sprint lets you control the use of *up* and *down* calls separately.)

error (number) The attribute number to use for the error messages (attribute –3). The default is the same as *select*.

ic (string, one argument) Insert *N* characters at the cursor, pushing the character the cursor is on and all the others to the right. It doesn't matter what characters are inserted or where this moves the cursor. (This string is not used in the current version of the editor.)

il (string, one argument) Insert *N* blank lines at the row the cursor is on, pushing it and all lower lines down by *N*. It doesn't matter if this moves the cursor. Not used if there is an *up* and *down* string given.

infobox (number) The attribute to use for infoboxes (attribute –5). The default is the same as *select*.

init (string) This string is sent to stdout when the editor is started. It can send any escape sequences wanted to set modes or print messages.

If the screen requires running a program to set a particular mode, that program can be run automatically by using a file name with *init* in the form "C*programname*" (for example, "init Cset.com 55" will run SET.COM, passing it 55 on the command line). **Do not run any programs this way that will terminate and stay resident, that is, memory-resident programs.** The string can start with an 'H' to execute a hardware command.

map (number) Memory map segment address. For increased speed Sprint can directly write to a memory-mapped display. The map is assumed to be an *rows* by *cols* array of words, the high-order byte of which is the attribute, the low-order is the character (IBM display style). This field indicates the segment address (*top 16 bits*) of the screen memory. Sprint will word-address this memory map only (not character-address). Sprint will handle the map correctly even if this map is write-only (although it uses reads to restore the contents behind menus, resulting in temporary garbage that will be cleared by the next redraw).

menu (number) The attribute number to use for the menus (attribute –4). The default is the same as *select.*

plain (number) The attribute number to use for plain text (attribute 0). The default is 0.

reinit (string) This string is sent after the init string, and after every screen redraw of the editor. It is used to clear the screen, but if not given, the editor will use *clreol*'s or fill the screen with spaces. As with *init,* programs can be run using the "C*programname*" or the "H*hardware string*" construct.

reset (string) This string is sent when the editor is exited. As with *init,* you can run programs using *reset* with a file name in the form of the "C*programname*" or the "H*harware string*" construct.

rows (number, required) The number of rows on the screen. Most screens have 25. The editor will work with up to 120 rows.

select (number) The attribute number to use for selected text (attribute –1). The default is 1.

set (string, one argument) This is the string to be sent to stdout to set the color to the argument. Not used if *map* or BIOS is used.

snows If you put in this flag on a memory-mapped device, Sprint will enable its built-in snow prevention. This snow prevention is specific to the IBM CGA and is useless on others. Sprint looks *only* at this flag; it does not check the screen type itself (because it is sometimes wrong). If this flag is not given, you will get snow on a color screen. If it is given, Sprint will check the CGA retrace flag before writing any character, even if this is *not* a CGA. On a non-CGA screen, if snows is on, it will wait forever before writing any characters.

SP (number) Entry for attribute 32 (spaces). The default is a space.

status (number) The attribute number to use for the status line (attribute –2). This attribute is also copied to any unfilled locations in the attribute vector by SP-SETUP. The default is the same as *select.*

up (string, four arguments) This is the string to send to stdout to scroll a region up. Takes four arguments: the top line to scroll, the line *after* the last line to scroll, the amount to scroll, and the difference between the last line and the amount to scroll (which you can usually ignore, but is useful for some schemes). If the string is a capital *B* then the BIOS scroll call is used. Notice that even when *map* is given, Sprint needs a scrolling method, it does *not* scroll the contents of the map itself!

When creating new screen descriptions, be careful to save any working version of DEFAULT.SPS that you have. If not, you may break the editor with a bad screen description, and be unable to edit it to fix it.

Port Descriptions

To allow for changing operating systems and printer features, and to allow for optimal performance, Sprint can directly control the computer's hardware. This control provides a means for sending output to any serial port, parallel port, or memory-mapped output port.

For most systems, Sprint's default formatter output method (sending text to PRN:) is sufficient. But if the printer or screen needs XON/XOFF handshaking, or if you want the output to go "around" the operating system to a different port, you will need to select a port for it.

Note that using a port will prevent print spoolers and shared printers on a network from working.

Port records have the following fields:

imask (number) Input ready mask. This is AND'ed with the value read from the input status port.

init (string) Port initialization string. This string contains values to send to the output port to set things up. Each character is sent (at full speed, with no handshake). You can cause data to go to other ports by placing "%*n*p" in the string, where *n* is the number of the port you want to send to (unfortunately, *N* must be in decimal—refer to Appendix G for help).

io Defines the type of I/O. This can be PORT, which means it accesses machine I/O ports, DMA, which means it accesses direct memory mapped locations, or DEVICE, which means that Sprint will open the port's name (which should be something like *\dev\lpt1*), as a file, and write to it. If DEVICE is given, all other fields are ignored.

iport (number) Input data port number, which describes where to read the input data from.

iseg (number) DMA segment number for both *iport* and *istat*.

iready (number) Input ready value. After AND'ing with the *imask,* if the status byte equals this number the formatter assumes there is an input to be read from the input data port.

istat (number) Input status port. This port is read to check the input status, to see if an input character is ready.

omask (number) Output ready mask. The byte read from the output status port is AND'ed with this.

oport (number) This is the port number to send output characters to.

oready (number) Output ready value. After AND'ing with the *omask,* the status byte is compared with this number. If equal, the port is considered ready for transmission. This should have 0 bits where the *omask* has 0 bits.

oseg (number) If this is a DMA port, this is the segment number (top 16 bits) for both the *oport* and *ostat* addresses.

ostat (number) Output status port. The port number to read to check the output status.

reset (string) Port reset string. This string is sent after all the data to close the port.

send (string) Sends a string. Some ports (such as IBM parallel ports) need a few outputs to send a character. This string works like the *init* string, and is done to send each character. Put %c in the string where you want the character to be.

sync (XON) XON means uses the XON/XOFF (^S/^Q) protocol. This is by far the most popular scheme; in it, the printer sends a ^S to the computer to tell it to stop transmitting data, and a ^Q to tell it to start again. The Sprint formatter will react within one character of the stop signal.

If you don't specify *sync,* there is no synchronization protocol, and the input port is ignored. To do DTR and other line signal protocols, set up the output status port to check the line.

Using an External Terminal

In some cases, it may be useful to use an external terminal device, rather than the screen that is built into the computer. Sprint can support external terminals, plugged into the communication ports on the back of your computer, with up to 120 lines. Assuming you have already added the screen definition to the MAIN.SPL file, here is how the setup can be done:

- Get the necessary cables to plug the external terminal into the COM1 port on the back of your computer, and connect it up. *You do not need the terminal's keyboard (you will be typing on your computer's normal keyboard), so move it out of the way if possible. Place the monitor where you can see the screen.*
- Use your favorite communications program to make sure the connection to the terminal is working. *If possible, turn on the XON/XOFF protocol for the terminal, and set the baud rate as high as possible, such as 9600 baud for a standard IBM PC.*
- Run SP-SETUP, and select the correct terminal type. When SP-SETUP asks for the port to use, select COM1 port.
- Run the Sprint editor as normal. *Your display will come out on the alternate terminal. When you exit, you will go back to the regular display.*
- If you want, before you run Sprint, type "`command >com1`" at the DOS prompt, and all your normal MS-DOS output will also come out on the alternate terminal. *However, programs that use BIOS calls or direct memory-map will come out on your old display. Do not use IBM's* **ctty** *command as this redirects input from the terminal's keyboard as well, and will result in strange effects in many programs.*

There are also some terminals with taller screens that come with a card that you put inside your IBM. These are less versatile, but work with more existing software. If the editor, set to either IBMMono or IBMColor, displays on the top 25 lines of these screens fine, you can make Sprint use the entire screen by modifying the description in DEFAULT.SPL to have the correct number of rows (change the "25" to whatever the correct number is). If now the *initial* display is then fine, but scrolling doesn't work, remove the BIOS scrolling calls by adding "up,down" to the screen description.

If you have one of these larger screens (especially one that is wider than 80 columns), and IBMMono or IBMColor does not seem to work, try IBMBios. If this displays in the upper-left corner of the screen, modify it to have the correct screen dimensions and use it. If IBMBios does not work (or is too slow for your taste), you should write or obtain a "device driver" which interprets escape sequences to work on your screen, and make a screen description using these escape sequences.

Printer Definitions

Printers used with computers—whether they're dot matrix, letter quality, laser, or phototypesetting—are controlled by sending a sequence of data bytes to them. This sequence is sent through either a *serial* or *parallel* port, and through a cable to the printer.

The Sprint formatter is able to generate virtually any sequence of bytes needed to operate a printer. It can even run printers that don't take ASCII code, or ones that don't use a carriage return or line feed to end the line. Even if Sprint can't run a particular printer (a rarity, but it could happen), it can still generate an intermediate file in almost any format you want, which can then be read by a simple translation program to send data to the printer.

Printer Requirements

Sprint does require a few minor things from the printer. In most cases, these things don't prevent any printer from being used, but they can limit the ways in which it can be used.

For example, the formatter assumes there is a *print head* that can be moved around the page, and that remains where it was last placed until another command is sent. On purely mechanical printers such as Diablos, this is of course the actual print head mechanism. On laser printers this is just an abstract idea, stored as state variables in the micro computer that runs the laser. Most printers fall somewhere in between these two extremes, for instance most dot-matrix printers move a physical head vertically, but construct horizontal lines with a virtual print head, and don't actually print the line until a vertical move is required.

The most important printer requirement is that the printer have a *horizontal resolution unit*. This is a fixed-size, but usually very small, horizontal distance that is the *minimum amount the print head can move*. No command or character (including proportionally spaced characters) can move the print head a non-integer number of these. Moreover, there *must* be a method of moving the print head by *one* of these units *without printing*.

On a typical dot matrix printer, the horizontal unit is the same width as a letter. The method used to move the print head on these printers is to print spaces. On more advanced printers, letters are often 10 or 12 horizontal units wide, and on these you can move by these units using special escape-sequence commands. On daisy wheel printers, these units are very fine, usually 1/60 or 1/120 inch, and horizontal motion is usually simple to configure.

An example of a printer that is limited by this requirement is a "dumb" printer that can print both 10 pitch and 12 pitch. If you try to set it up so both fonts can be used in the same document by the formatter, the horizontal resolution unit is 1/30 of an inch (the difference between the 1/ 12 and 1/10 of an inch letter widths). But there is no way to position the print head with this accuracy, so you cannot combine these fonts into a single printer definition.

A far more common case is printers that have a proportionally spaced font but that cannot move by the units that the proportional widths are multiples of. If such a printer has a method of positioning the print head relative to the left edge of the paper, *and* the units used for this positioning do not vary depending on the font *and* if they are sufficiently small, Sprint supports a work-around for this case. You can tell Sprint that the horizontal unit is the unit required for the proportional widths. Then when the formatter wants to move to a location, the printer actually moves to the nearest approximation based on these new units. Otherwise, you are unable to use proportional spacing on such printers.

Another requirement is that there be an equivalent *vertical resolution unit.* It does not need to be the same size as the horizontal unit. Almost all printers fulfill this requirement. On some, an equivalent problem to the 10 pitch/12 pitch problem is that they can be set to 6 or 8 lines per inch, but not to the 1/24th of an inch difference. These printers can only run in one line spacing at a time.

The last requirement is that the formatter be able to *fully* control the printer. The formatter must be able to position the print head anywhere on the page (although it won't print too close to the margins, and will always advance down the page, never up), and print characters there. This only causes trouble with "intelligent" printers that also attempt to be word processors. Watch out for printers that justify lines all by themselves. This is a sure sign that there will be some difficulty getting Sprint to run them.

Printer .SPL Records

A printer is described by a number of records in a library file.

The most important record is the *printer* record. This describes most of the printer, including the horizontal and vertical resolution, how to move the print head and print characters, and global aspects of how the printer works, such as whether to pause after each page. The printer record also names the printer.

After the printer record in the library file, there can be any number of *font* records, with a minimum of one. Each font record describes one of the fonts that the printer can print, including the commands used to turn that font on and off, and the size of the letters in that font, given in horizontal and vertical units.

There can also be *attribute* records (which are just called *attr* in the library). Each attribute record also has a name, and describes the commands used to turn that attribute on and off.

Attributes are different than fonts in that they cannot change the widths of characters. There is always exactly *one* font in effect at all times. There can be any *set* (zero or more) of attributes turned on at any time, but the widths used are the widths of the current font.

Fonts can refer to *proportional spacing table* records (called just *pst* in the library). These provide the widths of each character in a proportionally spaced font. Many fonts can use the same proportional spacing table, and the table can be anywhere in the library file.

Fonts can also refer to *translation character table* (called just *tct* in the library) records. These allow a one-character to one-character translation of ASCII codes to a different code to be sent to the printer. These were originally designed to translate codes for daisy-wheel printers where the letters were arranged differently on the wheel. They are also used to indicate to the formatter which characters (including high-order foreign symbols) the printer can print, and to translate to EBCDIC and other non-standard character codes.

All these records are assembled by SP-SETUP into a single .SPP file. The formatter opens this file and reads most of it into memory. The spacing and translation tables are swapped into memory as needed (it keeps two of each in memory).

Printer Record Fields That Concern the Formatter

This is a list of the fields a *printer* record can have. This section lists the fields that the formatter uses to figure out where to position the text. All the fields in all records have a default value of zero unless otherwise noted.

hpi (number, required) Horizontal units per inch. This field defines the size of a *horizontal printer unit,* which is used for all other measurements for this printer. A printer unit is the smallest possible horizontal motion of the print head. See the earlier discussion on the requirements for horizontal resolution.

Common values are 10 or 12 for most fixed-width devices, and 120, 144, 160, and many other values for microspacing devices. Laser printers are in the range of 300 to 1000.

Watch out for Diablo "compatible" printers, which have commands to move in 120ths, but round these to some other unit, usually 180th of an inch. The real hpi of these is 60 (the greatest common divisor of 120 and 180), and arguments to the horizontal motion commands must all be multiplied by two.

All character widths (including proportional fonts) must be multiples of this unit. There must also be some method of moving the print head by one of these units *without printing anything*. A suprising number of printers fail to fulfill these requirements, and Sprint will be unable to use them fully. There are, however, lots of ways to cheat. The simplest is just to remove fonts until the remaining set is all multiples of a usable *hpi* (for example, remove Pica or Elite if they conflict). Another is to use the second argument of *fwd* to position absolutely using the finest units the printer does accept, and have the *fwd* string do math to round the *hpi* units to these units (for instance if the proportionally spaced font is measured in 360ths of an inch, and there is a command to absolutely position horizontally by 60ths of an inch, a *fwd* string containing "%1g%4/..." and an *hpi* of 360 will do the job with unnoticeable error.

To prevent internal overflows, the page cannot be more than about 32,000 units wide, meaning the maximum for *hpi* is about 3500.

vpi (number, required) Vertical units per inch. This field defines the size of the *vertical printer unit,* which is the vertical analogy of a printer unit. It is the smallest possible vertical motion of the print head.

Common values are 6 or 8 for fixed-feed devices, 12 or 16 for half-line feeding devices, and 48 or 72 for micro-feeding devices. Laser printers generally have the same *vpi* as *hpi.*

To prevent internal overflows, the page cannot be more than about 32,000 units tall, meaning the maximum for *hpi* is about 2500.

Device Control

The following printer record fields control much of the general operation of the printer:

file (yes/no) File output. If this is given, the output will be sent to a file with the name xxxx.PRN, rather than to an output port. This can be used to generate online-readable output, or to generate output that needs further manipulation or transmission, such as to a remote typesetter. Page pausing, paper offset, and wheel changing are ignored for file output.

Normally, this field is specified when the SP-SETUP program is run, and should be left off in the printer description.

init (string) Initialization String. This string is sent to the printer before anything else. Direction, pitch, shift, ans so forth will be automatically initialized *after* this. If *ipo* is zero, this should leave the printer at the top of a blank page.

If you have a two-bin sheet feeder, you can make this string feed from the letterhead bin, and the *ff* string (discussed on page 471) feed from the plain paper bin. Using this strategy, only the first page will be on the letterhead. For special printer support, the *init* string can be given as: "F<filename>" to dump a whole file of initialization data or a downloaded font to the printer, excluding a trailing ^Z.

This can be used to download a "soft font" to a printer, as long as the file being downloaded contains leading data to alert the printer that font data follows.

ipo (number) Initial paper offset. For convenience in tearing off pages of fan-fold paper, Sprint can advance the paper a specified amount past the last page boundary after printing a file. It also assumes the paper has been advanced this distance when printing starts. This can be used to align the perforations with a page cutter or other reference point. The distance is given in 6ths of an inch.

If page pausing is turned on, this is ignored.

leftm (number) Left margin. It is highly recommended that the printer be set up so that column zero is at the left edge of the paper. Unfortunately, some printers cannot move the carriage that far left (or the paper that far right), therefore this field lets you specify (in 10ths of an inch) how far from the left edge of the paper "column zero" is. The formatter will subtract this amount from the horizontal position of everything it wants to print, and won't print anything formatted to the left of this.

If you run SP-SETUP and specify that cut sheets are to be used, you can also specify *leftm* because many sheet feeders insist that the paper be in the middle of the platen rather than the left edge.

pause (yes/no) Page pause. Sets whether, by default, the formatter will pause for the user to insert new pages into the printer. This can be overridden by the –pause formatter option. Normally, this field is set when the SP-SETUP program is run, and should be left off in the printer description.

print (string) Print a string of characters. The string must contain "%s," and is used to enclose and group of printing characters if the printer requires them. Our PostScript driver has the string "(%s)p " to produce a string and send it along with the "p" command to the printer. *so* and *si* are placed inside this string, but *all* other escape sequences, and @Escape commands, are outside of it.

reset (string) Reset string. This is the last thing sent to the printer. Direction, pitch, shift, and so forth will be set to reasonable defaults *before reset* is sent (for this reason, *reset* can change them because it has access to them last). The F string construct also works here, just as in *init*.

scale (number) Indicates the printer can scale the fonts (such as PostScript printers). Sizes are given in vertical units, and the number is the minimum increment between sizes (the formatter will only ask for sizes that are a multiple of the number). Size is set by sending a number with each font's *on* string (see the description on page 476).

topm (number) Top Margin. The printer should also be set so the top of the print head is at the top edge of the paper when each page is started. However, some printers will not allow this (usually because they take cut sheets, and the sheets must be fed in enough to get under the platen rollers). This field tells how far (in 6ths of an inch) the print head starts down the page. Text formatted above this point will not be printed correctly.

Usually, this field is left unspecified, and is supplied by the SP-SETUP program when you indicate that cut sheets are to be used.

Printer Horizontal Movement Control

Sprint must be able to control exactly the position of the print head across the paper, down to the resolution specified by *hpi*. Unfortunately, many modern printers try to be "too smart," handling justification by themselves, which often makes it very difficult to do exact control. You must disable this "feature" and carefully use the following field descriptions to get around any strange effects of it.

back (string, three arguments) Same as *fwd*, but goes in opposite direction.

bs (string) String to move the carriage backwards by the horizontal pitch (usually a ^H will do this). In general, formed-character printers can do this, dot matrix printers can't. On many proportional-space printers the size of a backspace depends on the last character printed. In such case, this field cannot be used.

cm (Width/Pitch) Character motion. This defines the distance the print head moves after it prints any character *except space*. One of the following keywords is allowed:

- *Width* means the print head moves the width of the character. This is by far the most common, and results in the most compact output coding.
- *Pitch* means it moves by the horizontal pitch. A printer that does this should have the *shp* control sequence (although it will work without it, but incredibly inefficiently). This results in much longer output, but has the advantage that the width table is independent of the printer, and can be adjusted by editing the library file. Most daisy wheel printers do this.

cr (string) This string causes the carriage to return to the left margin without advancing the paper. Usually a ^M (carriage return) will do this.

fwd (string, three arguments) Forward by *N* units (first argument), or to the *M*'th column (second argument). This string moves the carriage *N* printer units to the right, or to the *M*'th column. Many printers will do this, but the correct command is hidden cryptically in the printer manual. Look for "variable spacing" commands, or a set of characters that are spaces of different widths. You may have to go into "graphics mode" and send *N* nulls, but only if there is nothing else. (This is what Epson and IBM graphics printers do, and they are very slow at it!)

The current horizontal pitch is passed as the third argument to *fwd,* and is useful if the command changes it and you have to set it back.

mhm (number) The maximum allowable argument for *fwd, back,* and *shp.* If the formatter wants to move further than this amount, it will send multiple commands in a row. If not specified, the formatter assumes any *N* can be sent.

shp (string, one argument) Set horizontal pitch to *N.* The distance the print head moves when it prints a *space* is called the *horizontal pitch* of the printer. Some printers (Diablos and compatibles) have the capability of changing the horizontal pitch to any number of printer units. This is the string sent to cause this change. If this string is not given, the horizontal pitch is assumed to be the width of a space in the current font, and is unchangeable.

tab (string) String to cause the carriage to move to the next tab stop. Tab stops are assumed to be every *tabsize* (default is 8 unless it is overridden by the –t formatter option or by the Tab Expansion menu command) columns. This is usually ^I, and is useful mostly for file output.

Vertical Movement Control

Vertical movement is similar to horizontal movement. Fortunately, Sprint only needs to advance down the page, so there are no backwards commands.

down (string, two arguments) Move down *N* units (first argument), or to the *M*'th unit from the top of the page (second argument). This string feeds the paper *N* printer units up. If you have a printer that can feed paper in half-line increments, make this string do the half-line feed (ignoring the argument) and set *mvm* to 1.

ff (string, one argument) Form feed string. Feeds paper to top of next page, or otherwise does something to indicate the separation of two pages. Usually a ^L will work. If not given, the formatter assumes *lf, nl,* and *down* can be used to move to the next page. Important: The formatter automatically sends the *cr* string first. If the printer has no *cr* string, this string must also return the print head to the left margin.

The previous page number is sent as an argument to this string. This can be used to update a page number display on the device.

lf (string) This string advances the paper by the current vertical pitch without moving the carriage horizontally. A ^J (line feed) will often do this. *Make sure it does not move to the left margin as well.*

mvm (number) Maximum vertical move. This is the maximum argument for *up, down,* and *svp.* If more vertical movement is desired, the formatter will send multiple commands. If not specified, the formatter assumes any *N* can be sent.

nl (string) This string advances the paper by the current vertical pitch (line height), and returns the print head to the first column. A ^M^J will usually do the trick. Whenever possible, the formatter will send this string, even if another string may work.

page (string) String to send at the start of each page. The current page number is supplied as an argument and can be put in with %d (or tested for odd/even to control which side of the paper to print). If this field is given, SP-SETUP will assume this is a laser printer and not ask the user about manual feeding or paper positioning.

rlf (string) This string moves *back* by the current vertical pitch (reverse line feed). Not required, and not used.

svp (string, one argument) Set vertical pitch to *N*. The amount the paper moves vertically when the printer is sent *nl* or *lf* is the *vertical pitch.* This string changes the vertical pitch to any *number* of printer units (commands to set the *lines per inch* cannot be used!). The *nl* or *lf* string will always be sent immediately after this string. If this isn't given, the vertical pitch is assumed to be the *size* of the current font.

up Move up *N* units (first argument), or to the *M*'th unit from the top of the page (second argument). Not required, and not used.

Font/Attribute Controls

shadow (number) Bold shadow. The number of printer units to move right when overstriking bold characters. This makes bold characters wider and they stand out much more. The default is 0, and the maximum is 255.

so,si (strings) Shift out string, shift in string. If your printer has an "uppercase" and "lowercase" that must be toggled to print all the characters, the translation table can control this. The high-order bit of the translated character tells the printer to "shift out." This high-order bit is masked before printing the character.

If *si* is not given, but *so* is, *so* is instead sent with *each* character that has the high bit set. You must put %c in the string to indicate where the character should be printed. For instance, to do the Diablo ESC-x sequence, set *so* to ^[%c, don't set *si*, and place x+128 in the translation table.

If neither *si* or *so* is given, the high bit from the translation table is sent to the printer unchanged.

unc (string, one argument) Print *N* underscored spaces. The formatter usually underscores by backing up and printing underscore characters over the letters. Unfortunately, most dot-matrix printers do not print a respectable underscore when this is done, and this string provides an alternative method.

The result of printing this string should be equivalent to printing *N* underscore characters in a row, but the result should be an unbroken line at the correct height. Usually this string is *"<on>%(%)<off>"* (where <on> and <off> are the correct sequences to enter and exit underscore mode). Be sure the width of an underscore given in the width table is correct for this, the same as a space.

unw (string, one argument) Underscore a width. On some printers doing underscores by single dots is more convenient and looks better. This command should move the print head to the right by *N hpi* units, underscoring the area it passes over. The argument is the width (in printer units) to be underscored.

xnc (string, one argument) Strikeout *N* characters. Normally, the formatter strikes out characters by printing dashes, half a width of a dash apart, over the characters. This string provides an alternative way, and works the same as *unc*, except it is supposed to strikeout *N* times the width of a dash character.

xnw (string, one argument) Strikeout a width. Same as *unw*, but it should draw a horizontal line through the characters for strikeouts.

Special Flags

Special flag variables are available that allow printer descriptions to override certain "features" on printers. If false, they do nothing. If true, they try to circumvent the feature by brute-force methods.

flag1 (yes/no) Indicates that horizontal movement does not work in some fonts. The precise result of *flag1* is that before a *fwd* or *back* is done, the *off* string for the current font is sent, then the movement is done, then the *on* string is sent. The numerical value for *flag1* should be the h-pitch that the printer will have after the *off* string is sent. (Note: any *attribute* with the * field is shut off whether or not *flag1* is given.)

This fixes the common case where the horizontal units vary depending on the font. This is done by sending the *off* string for the current font, but no *on* string until after the horizontal spacing is done.

flag2 (yes/no) Indicates that vertical movement does not work in some fonts or attributes. The precise result of *flag2* is that before a *cr, lf, nl,* or *down* is done, the *off* string for all attributes and the current font is sent. After the movement, the **on** string for the font is sent. The most common reason for this flag is that some of these commands turn things off anyways. This should also be used if the printer definition file has a font attribute called *uns* (underline string), so it does not underscore the leading spaces on the next line.

flag3,flag4 (yes/no) Extra flags. These flags currently do nothing.

Special Notes on Daisy Wheel Printers

Diablos, SpinWriters, and their kin are very popular machines. Therefore, we have designed some special fields around their own peculiarities. You can use these fields for other printers, but it's unlikely they will be useful. Note: Newer Diablos and compatibles that use microprocessors and have an internal proportional spacing mode are much smarter, and you may want to avoid these fields and run that processor directly in proportional mode.

center (yes/no) Centers characters. Most daisy wheel printers center the character graphic at the carriage position rather than put the left edge there. If this is given, the print head is positioned half a character width to the right before printing. This will only work correctly if the printer also has *cm Pitch.*

Some printer manuals get very confusing on this point. Generally, if the printer has built-in widths for proportional characters (often specified as a "left" and "right" half-width that you must add together), the characters are *not* centered.

wheel (yes/no) Change print wheels. This indicates that all font changes require the operator to place a new print wheel in the printer. The formatter will pause and prompt the user as necessary.

Font Defaults/Overrides

The following fields affect the font descriptions that go with the printer:

pst (name) Names a width table to use *instead* of any width table given for a font. If you specify this, all proportionally spaced fonts for the printer will use this table, rather than their own, but fixed-width fonts will still be fixed-width.

size (number) Standard height. The default height for a font, in vertical printer units. In most cases, all the fonts for a printer have the same height, so it is easiest to specify it here.

tct (name) Names a default translation table to use. All fonts use this translation table unless they specify a different one of their own.

Font Descriptions

A *font* is a different form of text. It can change the width and height of characters, and change their appearance. Only one font is "active" at any time.

A printer is assumed to have all the fonts and attributes that are listed in the .SPL file between it and the next printer description. These font descriptions can be interspersed with *pst* and *tct* tables, blank lines, and even comments (contrary to popular belief).

The difference between "fonts" and "attributes" is this: A font can change the width and height of characters, and *exactly one* font can be in force at any time. Before turning on a new font (even if the commands are nested), the formatter will always send the *off* string for the previous font. An attribute, however, *cannot* change the widths of characters (disregarding the * field), *any set* of attributes (zero or more) can be in force, and the formatter does not send the *off* string for the previous one when turning on a new one.

At a higher level, the formatter tries to treat all the fonts and attributes the same. This is why commands use only "font" fields, even though you can freely put the names of either fonts or attributes there. All the font records for a printer should be listed right after the printer record in the library file.

In the library, a font name is a string of words separated by periods, like bold.italic. The formatter splits this name up into separate commands. You cannot give the command @Bold.italic, but you *can* give the commands @Bold and @Italic, nested in either order inside each other, and get the font "bold.italic." (This *only* works if there are also "bold" and "italic" intermediate fonts, and these font records must be given in order, with the simple fonts first, and the "compound" fonts afterwards.)

Font records can have the following fields:

* (number) A multiplier for the *pst* table. This is useful for phototypesetters that do multiple sizes of the same font. It lets you use a single *greatest common divisor* width table for the entire set. Watch out for typesetters that modify the widths (for good reasons) as the type size changes; you will have to make separate tables for each size of type. The maximum value allowed is 255.

off (string) This string is provided to prevent long and complex *on* strings. It is sent immediately before sending the *on* string for the next font, and turns off the effects of this font.

This is also the string that is sent to "turn off" a font when *flag1* or *flag2* is given. Note that in many cases no *off* string is needed.

on (string) String to send to turn the font on. Takes one argument, the size desired (on any device that can't *scale*, this argument will be equal to the *size* field). Notice that the *on* string will be used to change the size of the current font as well as to switch fonts.

pst (name) The proportional spacing table to use. If none is given, the font is fixed-pitch. Either *width* or *pst* can be given for a font, but not both. Any number of fonts can use the same width table.

Width tables are described with *pst* records, and can appear anywhere in the same .SPL file. We recommend you put them immediately after the font. Each field in a *pst* is a letter, with appropriate quoting by backslash, followed by a number giving the width. The best way to make a new one is to copy an old one and edit the numbers. Make sure there is a comma after *every* number except for the last one in the last row, or the configuration parser will throw away the rest of the table without warning you. Any characters not defined will be given the same width as the '2' character.

Characters (such as accent marks) can be given a zero width, but first *make sure* the printer does not move its print head when sent this character, because that's what the formatter will expect.

size (number) Standard height. The height of the font, in vertical printer units. This is the distance between baselines when this font is printed. The vertical pitch is set to this value when the font changes. The default is the *size* value in the printer description, or 1 if that isn't given.

On printers that can *scale*, this field indicates what size the width table is for, to determine how much to multiply it by for the current font size. For best accuracy, make this be 12 points or whatever your most-used size is, so this multiplication and resulting rounding is not done.

width (number) Standard width. This is the width, in horizontal printer units, of a character in a fixed-pitch font. The default for this is 1.

tct (name) The translation table to use. If none is given, the default translation table from the printer record is used. If none is given there, the printer is assumed to print just the regular ASCII characters (from space to tilde). The *tct* table describes what code to actually send to the printer when a certain letter is needed, usually for printers that do not take the ASCII character set. This is a completely separate process than the TCT command, which is handled by the formatter, not the printer driver. Notice that the Char command is translated by this table; that is, it doesn't necessarily send the literal character to the printer. Use the Escape command to send literal characters.

Often a *tct* table is used with the *so* string to translate some characters to multi-character escape sequences. Any code with the high bit set in the tct table is sent with the *so* string. By using the %[case in the *so* string, theoretically you can have up to 128 arbitrary different sequences sent for different characters, but this gets pretty hairy to define and debug.

Translation tables are defined with *tct* records. Like width tables, these can appear anywhere in the .SPL file, but we recommend you put them right after they are referenced. Each field is a character, as in *pst* tables, followed by a code to translate that character to (a *number*, not a character). As with width tables, make sure there is a comma after every field or the table will get truncated without telling you. Any locations that are undefined are translated to themselves.

You cannot translate a character to zero unless you use *so* to do it. A zero entry in the TCT table will cause the formatter to produce the "can't print a 'x'" error message. Translating space (SP) to zero will prevent the driver from sending spaces to do horizontal positioning, if for any reason spaces don't work (a good example of this is Compugraphic typesetters, which have no space in their fonts).

Attribute Descriptions

An attribute changes the appearance of the text, but does not change its size (except to possibly multiply the width by a constant). Any set of the available attributes can be "active" at any time. All the attribute records for a printer should be listed immediately after the printer record in the library file.

Attribute (attr) records take the following fields:

* (number) Size multiplier for the attribute. This is for "double-width" attributes common to dot-matrix printers, although numbers other than 2 will work. The default is 1. Such an attribute will be turned off before the formatter attempts to move horizontally.

on (string) This string turns the attribute on.

off (string) String to turn the attribute off. All *off* strings are sent before any *on* strings for the next attribute or font. Unlike fonts, the *off* string is required for correct attribute operation.

Proportional Spacing Tables

Pst records look much different from other records in the library file, but are actually read using the same rules. Each field name is a character. Control characters and meta-characters are read just as in strings; for instance, you must quote the caret with a backslash. The width value is in printer units.

The special field *SP* is the width of a space character.

It is quite difficult to type in a *pst* table from scratch. We recommend that you copy one of the existing ones, and edit the numbers to the new values you want.

Characters that are not specified are assumed to have the same width as a '2' character. The width of a '2' *must* be specified.

A range of characters can be given the same width by using two letters with a hyphen between them (as in "a-z"). You can also supply an *as* field, in case two tables are similar.

Note: If the printer description has an *unc* string, use the width of a space for the width of an underscore.

Translation Tables

Tct tables in the library look like proportional space tables. The field names are the same, but in these the value is a *character* to translate to. The character can be a Control character or a "meta-character," or a zero can be given. A meta-character indicates the "shift" state, and causes the *si* and *so* strings to be sent (if the printer has them). It is often convenient to place \ *nnn* as the character, to translate to a certain decimal number.

No value or zero indicates that the font *cannot* print the given character, and the character is constructed using the formatter's TCT table.

By default, all ASCII characters from space to tilde translate to themselves. If the printer cannot print one of these characters, you must indicate that fact by entering that character followed by no value into the table.

You can name a range of characters and cause them to be translated to another range by supplying the first value in the range to be translated to. This is useful for translating all the IBM foreign characters to themselves (do ~^@-~~@ ~^@).

Note: The formatter assumes that every font can print an underscore, a hyphen, and a 2. It also ignores the **si/so** state for spaces.

Making Your Own Printer Type

All printers are different. What follows are some general instructions and guidelines to follow to make the formatter run your printer.

Sprint can support *every* printer in the world,[2] even ones we have never seen before! Of course, if we don't yet support your printer, you will have to do a bit of work to drive it, but it will be well worth your effort.

- You need a complete understanding of the structure and fields of a printer description. *Read the previous sections thoroughly.* Print out a copy of our library files and study it. Despite any initial impressions, you do not need to be a computer scientist to do this, but a good understanding of how printers and ASCII code work would be helpful.

[2] Well, almost. Sprint won't support "Fortran Carriage Control" line printers. Generally, there must be *some* relationship to ASCII, even if the codes are completely different.

- Next, make sure the Sprint formatter can talk to your printer. This can be checked by using the "Plain" printer definition. It is important to completely debug the serial or parallel connections and synchronization protocols before attempting to use tested printer definitions. Make sure the printer prints many sequential pages of text without losing data.
- You need very accurate knowledge of your printer. Get the manual, and read the section on "programming" or "control codes." Use a communication program and try sending each escape sequence to your printer to see *exactly* what it does. Too often, the manuals are wrong, leave out some important information, or are unbelievably obtuse.
- Then write your new .SPL entry. It may be helpful to copy one of ours, then edit it to your specifications. Change only one field at a time, so when disaster strikes, you'll have an idea of what caused it. Put lots of comments in to say what you are doing. If your printer does multiple fonts, don't try to define them all at once. Instead, pick a simple fixed-pitch font for your initial tests.

Printer manuals often give the escape sequences in BASIC, or list the characters to be sent in other obscure ways. The ASCII conversion chart on page 483 provides a way to convert it to standard notation. For example, `chr$(27)` in BASIC means character 27 decimal. Look up the number in the *Decimal* column of the chart. Three letter codes such as `ESC` indicate the ASCII names of the control characters. These are also listed in the table.

Test it, using very short files first. It is quite possible to crash the formatter with an error in the printer description. Don't worry if this happens, just reboot and try to figure out what you did wrong. The formatter will crash (in an infinite loop) if you do not provide the necessary escape sequences (such as *fwd*) to move the print head to any unit. If you get a "divide by zero" error, a field such as *vpi* that must be non-zero was set to zero or left unspecified.

If your printer does proportional spacing, you must make a *pst* table for it. Sometimes the widths of the characters are listed in the back of the printer's manual. Make sure these are listed in the same units you used for the *hpi*. If they are, you can use them. If not, you will have to convert them first.

If the manual does not have such a table, you can come very close by measuring the character widths yourself. First, you must have run the following macro that creates the test file *SPACING.TXT*:

```
open "SPACING.TXT" clear 32->x 224 repeat(50 repeat (x insert) ++x 10
                                          insert)
```

Next, add the new proportionally spaced font to the printer definition, but just give it a *sw* field to indicate that it is fixed-width. Then format

SPACING.TXT with the –plain switch and the –font xxx switch to select the font.

There are 50 characters in each line in SPACING.TXT. If you measure the length of each line, and then divide by 50, you will then have the width of each character in the font. Next, you must convert the result into whatever units you used for the *hpi* field when writing the printer driver.

Share Your Configurations!

If you have configured Sprint to work on a new printer, computer, or other device, no matter how obscure, we would enjoy seeing your work, and adding it to the distributed Sprint database.

Once you have perfected a new configuration, send the new library entries on disk (IBM-readable format) to us. Please don't edit the MAIN.SPL and send that; make a *new* xxxx.SPL with *just* your additions in it. Also send a letter or text on disk describing what you did and interesting points about the new hardware. Our address is

Borland International
4585 Scotts Valley Drive
P.O. Box 660001
Scotts Valley, California 95066-0001
United States

APPENDIX

G

ASCII Character Set

The American Standard Code for Information Interchange (ASCII) is a code that translates alphabetic and numeric characters and symbols and control instructions into 7-bit binary code. Table G.1 shows both printable characters and standard control characters.

You'll especially need these ASCII numbers if you're creating your own hardware drivers. You also need the ASCII numbers in Sprint when you're using the Char command. You also need to note the order of the characters, since the Utilities/Arrange-Sort command arranges according to ASCII order.

IBM Extended ASCII Character Set

DEC	HEX	CHAR			DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	0	^@		NUL	32	20		64	40	@	96	60	‘
1	1	^A	☺	SOH	33	21	!	65	41	A	97	61	a
2	2	^B	☻	STX	34	22	"	66	42	B	98	62	b
3	3	^C	♥	ETX	35	23	#	67	43	C	99	63	c
4	4	^D	♦	EOT	36	24	$	68	44	D	100	64	d
5	5	^E	♣	ENQ	37	25	%	69	45	E	101	65	e
6	6	^F	♠	ACK	38	26	&	70	46	F	102	66	f
7	7	^G	•	BEL	39	27	’	71	47	G	103	67	g
8	8	^H	◘	BS	40	28	(	72	48	H	104	68	h
9	9	^I	○	TAB	41	29	)	73	49	I	105	69	i
10	A	^J	◙	LF	42	2A	*	74	4A	J	106	6A	j
11	B	^K	♂	VT	43	2B	+	75	4B	K	107	6B	k
12	C	^L	♀	FF	44	2C	,	76	4C	L	108	6C	l
13	D	^M	♪	CR	45	2D	-	77	4D	M	109	6D	m
14	E	^N	♫	SO	46	2E	.	78	4E	N	110	6E	n
15	F	^O	¤	SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P	►	DLE	48	30	0	80	50	P	112	70	p
17	11	^Q	◄	DC1	49	31	1	81	51	Q	113	71	q
18	12	^R	↕	DC2	50	32	2	82	52	R	114	72	r
19	13	^S	‼	DC3	51	33	3	83	53	S	115	73	s
20	14	^T	¶	DC4	52	34	4	84	54	T	116	74	t
21	15	^U	§	NAK	53	35	5	85	55	U	117	75	u
22	16	^V	■	SYN	54	36	6	86	56	V	118	76	v
23	17	^W	↨	ETB	55	37	7	87	57	W	119	77	w
24	18	^X	↑	CAN	56	38	8	88	58	X	120	78	x
25	19	^Y	↓	EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z	→	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	←	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	∟	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^]	↔	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	▲	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	▼	US	63	3F	?	95	5F	_	127	7F	

Table G.1: ASCII Table, continued

IBM Extended ASCII Character Set, continued

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	β
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	∅
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	•
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Index

BORLAND
INTERNATIONAL

BORLAND INTERNATIONAL, INC., 4585 SCOTTS VALLEY DR., P.O. BOX 660001, SCOTTS VALLEY, CA 95066-0001 PART# 22 MN-SPT03 BOR 0821