Spectrum Writer®

User's Guide

Reference Manual

Spectrum Writer Release 3.0.1

While every effort has been made to ensure the accuracy of the material in this publication, Pacific Systems Group shall not be liable for any errors contained herein, or for incidental or consequential damages resulting from the performance, furnishing or use of this manual. If you find technical inaccuracies or typographical errors, we would appreciate your letting us know about them. Other comments concerning the usefulness of this publication are also welcome.

Printed in the United States of America

Pacific Systems Group, LLC
32533 Regents Boulevard
Union City, CA 94587

www.pacsys.com

1-800-572-5517 ● 1-510-471-7111

How to use This Manual

What Should You Read?

It is not necessary to read this entire manual in order to start producing custom reports and PC files with Spectrum Writer. To learn how to use Spectrum Writer, we suggest the following steps:

1. Step 1.Read Chapter 1, "Introduction" to learn just what Spectrum Writer is and what it can do for you.
2. Step 2.If you will be producing custom reports, read Chapter 2, "How to Request a Report." There you will learn the basics of producing reports with Spectrum Writer.
3. Step 3.If you want to produce PC files, read Chapter 3, "How to Request a PC File." That chapter teaches you the basics of producing PC files with Spectrum Writer.
4. Step 4.Start producing your own reports and output files! When questions come up, use the Index at the end of this manual to locate the section that explains how to do what you want.

Note: If you are responsible for initially installing Spectrum Writer and defining your input files, also read Chapter 6, "How to Define Your Input Files" and Chapter 8, "Operating System Considerations."

How This Manual Is Organized

This manual is divided into two major parts.

Part 1 is the **User's Guide**, which explains in non–technical terms how to produce reports and PC files with Spectrum Writer. The User's Guide contains over 100 examples of actual Spectrum Writer runs. It also explains how to define files and setup the JCL needed to execute Spectrum Writer. Just read the parts of the User's Guide that explain what you need to do.

Part 2 is the **Reference Manual**, which provides complete syntax information about each of the Spectrum Writer control statements. You will only need to refer to this portion of the manual when you have specific questions about control statement syntax.

Following the Reference Manual is a section titled **"Updates to This Manual"**. Be sure to file any documentation updates that you receive in this section. And remember to check this section for the latest features available in your shop's current version of Spectrum Writer.

The User's Guide and Reference Manual are divided into 9 chapters, plus Appendices and Index. Following is a brief synopsis of each chapter and appendix.

Chapter 1, "Introduction."
This chapter explains just what Spectrum Writer is, and what it can do to save you time and effort. Everyone should read this chapter.

Chapter 2, "How to Request a Report."
This chapter is a tutorial on producing custom reports. It is divided into nine easy lessons. These lessons show you how to write the control statements that tell Spectrum Writer how to produce a report. Everyone who will be producing reports with Spectrum Writer should read at least some of the lessons in this chapter.

Chapter 3, "How to Request a PC File."
This chapter is a tutorial on producing PC files from your shop's mainframe data. It is divided into seven easy lessons. These lessons show you how to write the control statements that tell Spectrum Writer how to produce a PC file. Everyone who will be producing PC files with Spectrum Writer should read at least some of the lessons in this chapter.

Chapter 4, "Beyond the Basics."
This chapter shows how to use of some of Spectrum Writer's more advanced features to create more complex reports and output files. After you feel comfortable with the basics, scan the headings and examples in this chapter to get an idea of what else Spectrum Writer is capable of doing. You may find that you can use Spectrum Writer to produce reports that you thought were too complicated for a Spectrum Writer.

Chapter 5, "How to Make a Web Report."
This chapter shows how to create custom reports that are specially formatted for viewing on the Worldwide Web. Web reports can take advantage of many formatting options not available on mainframe printers. These include such things as custom fonts, colors, bold, underlined and italicized text, graphics, photographs, and much more.

Chapter 6, "How to Define Your Input Files."
This chapter shows how to define your company's files to Spectrum Writer. This one–time setup is necessary before your company's files can be used in reports or PC files. The analyst or programmer responsible for setting up Spectrum Writer file definitions should read this chapter.

Chapter 7, "Working with Databases."
This chapter shows how to produce reports and PC files using data from special databases (instead of standard files). Read this chapter if you will be using Spectrum Writer with a special database.

Chapter 8, "Operating System Considerations."
This chapter explains what "job control language" (JCL) is necessary to run a Spectrum Writer job under different operating systems. The analyst or programmer responsible for setting up the JCL to run Spectrum Writer should read this chapter.

Chapter 9, "General Syntax Rules."
This chapter explains some of the general rules to follow in writing control statements. For example, it explains: the rules for naming fields; how to split a long control statement into multiple lines; the rules for writing computational expressions; etc. It is not necessary to read through this entire chapter. Rather it is intended to be a *reference* chapter. Refer to the appropriate section whenever you need help writing a control statement.

Chapter 10, "Control Statement Syntax."
: This chapter shows the complete syntax for each of Spectrum Writer's control statements. It is not necessary to read through this entire chapter. It is also a *reference* chapter. Refer to the appropriate section whenever you need help writing a control statement.

Appendix A, "Data Types."
: This appendix lists the types of data that Spectrum Writer supports in input files.

Appendix B, "Display Formats."
: This appendix lists the many ways that Spectrum Writer can format data in your reports and output files.

Appendix C, "Built-In Fields."
: This appendix lists Spectrum Writer's built–in fields which are available for use in your requests.

Appendix D, "Built-In Functions."
: This appendix lists Spectrum Writer's built–in functions which are available for use in the COMPUTE statement.

Appendix E, "Error Indicators."
: This appendix lists Spectrum Writer's error indicators (such as ***I***), explains their meaning, and shows ways that they can be handled.

Appendix F, "Files Used in Examples."
: This appendix shows the Spectrum Writer file definitions (and the raw contents) of the sample files used for the examples in this manual.

Appendix G, "Speed-Up Tips."
: This appendix explains various techniques that can be used to optimize Spectrum Writer's run–time efficiency.

Appendix H, "Sample Data Exit Programs."
: This appendix shows a sample data exit program and a sample run that uses it.

Appendix I, "I/O Exits."
: This appendix explains how to use I/O Exits for special file processing. It includes a sample I/O Exit program.

Table of Contents

How to use This Manual 3
What Should You Read? 3
How This Manual Is Organized 3

List of Figures 15

Chapter 1. Introduction 19
What Is Spectrum Writer? 20
Create Brand–New Reports in Minutes 20
Use Mainframe Data in Any PC Program 21
Create Web Reports and E-Mail Attachments 22
Create Custom Mainframe Files in Minutes 22
Ways that Spectrum Writer Benefits You! 22
Spectrum Writer Pays for Itself Fast! 24
Spectrum Writer Features 25

Chapter 2. How to Request a Report 29
Lesson 1. How to Produce a Report in 5 Minutes 34
How to Use the INPUT Statement 34
How to Use the COLUMNS Statement 35
Another 5–Minute Report Example 37
Using Your Company's Files 37
Lesson 2. How to Specify Which Records to Include in Your Report 40
How to Use the INCLUDEIF Statement 40
How to Write Conditional Expressions 40
Lesson 3. How to Create Your Own Fields 46
Creating Numeric Fields 46
Creating Character Fields 48
Assigning Values to Fields Based on Conditions 50
Lesson 4. How to Make Your Own Report Titles 53
How to Use the TITLE Statement 53
More Date and Time Features 55
How to Align the Title 55
How to Put File Data in the Title 55
Lesson 5. Changing the Format of your Report 58
Using Display Formats 58
Specifying Column Headings 60
Specifying a Column's Width 60
Multiple Overrides 60
Lesson 6. How to Specify the Report Order 62
How to Use the SORT Statement 62
Automatic Sorting 62
Lesson 7. How to Create Control Breaks 65
How to Use the BREAK Statement 65
How to Specify Control Break Spacing 67
How to Print Statistics at a Control Break 67
How to Produce Multiple Control Breaks 69
Lesson 8. How to Create Summary Reports 73
How to Create a Summary Report 73

Lesson 9. How to Use Data from More Than One File. 76
How Auxiliary Input Files Are Processed 76
How to Use the READ Statement. 77
"One-to-Many" Random Reads 79
How to Use Multiple READ Statements 79

Chapter 3. How to Request a PC File 83
Lesson 1. How to Produce a PC File in 5 Minutes 88
Converting a Whole Mainframe File 88
Another 5–Minute Example 90
Using Your Company's Files 90
Lesson 2. How to Include Only Certain Records In Your PC File 93
How to Use the INCLUDEIF Statement 93
How to Write Conditional Expressions 95
Lesson 3. How to Create Your Own Fields 98
Creating Numeric Fields. 98
Creating Character Fields 100
Assigning Values to Fields Based on Conditions. 102
Lesson 4. How to Specify the PC File Order 105
How to Use the SORT Statement 105
Automatic Sorting. 105
Lesson 5. How to Create Control Breaks 108
How to Use the BREAK Statement 108
Customizing the Control Break 108
Lesson 6. How to Create Summary Files 113
How to Create a Summary File. 113
Lesson 7. How to Use Data from More Than One File. 116
How Auxiliary Input Files Are Processed 116
How to Use the READ Statement. 117
"One-to-Many" Random Reads 120
How to Use Multiple READ Statements 120

Chapter 4. Beyond the Basics 123
Additional Features in the COLUMNS Statement 125
Writing Print Expressions 126
How to Change the Column Headings 130
Special Options Related to Column Headings 133
How to Change the Width of a Column 135
How to Change the Way Dates, Times and Numbers Are Formatted 137
Formatting Tips for International Users 140
How to Format Data as ASCII 143
How to Blank Out Repeating Values 144
How to Change the Justification of Data within a Column 146
How to Specify Which Columns to Total 148
How to Produce Multi–Line Reports 151
How to Change the Report Margins 154
How to Print Bar Graphs 154
How to Print Vertical Lines between Report Columns 156
Including All Fields in the COLUMNS Statement 158
What If You Run Out of Room? 160
Why Do I See ****X**** in My Report? 160
Customizing the Report Titles 161
How to Include Data from a File in the Title 161

How to Include the Page Number, Date and Time in a Title . . . 163
How to Change the Appearance of Items in the Title . . . 165
How to Split the Title into Left Aligned, Centered, and Right Aligned Parts . . . 168
Special Options Related to Titles . . . 175
How to Print "Titles" at the Bottom of Each Page . . . 175
Customizing the Control Breaks . . . 177
How to Change the Control Break Spacing . . . 178
How a Default Total Line Looks . . . 180
How to Customize the Total Line at a Control Break . . . 182
How to Suppress the Total Line at a Control Break . . . 185
How to Customize the Statistical Lines at a Control Break . . . 186
How to Print Customized Footing Lines at a Control Break . . . 188
How to Print the Number of Items in a Control Group . . . 198
How to Print Header Lines at the Beginning of a Control Group . . . 200
Computing True Percentages and Ratios at Control Breaks . . . 202
Reports with Multiple Control Breaks . . . 204
How to Customize the Grand Totals . . . 207
How to Produce Summary Reports . . . 209
Printing a "Line Number" in Your Report . . . 211
How to Create "Top 10" Type Reports . . . 212
How to Count "Occurrences" in a File . . . 214
How to Break Totals Down into Categories . . . 217
How to Make “Crosstab” Reports . . . 219
A Simple Crosstab Report . . . 219
Another Crosstab Report . . . 221
Working With Multiple Input Files . . . 224
Using Multiple READ Statements for the Same File . . . 224
How to Chain READ Statements . . . 226
How to Name the Input File Records . . . 228
How Missing Records Are Handled . . . 229
Testing for Missing Records . . . 230
How I/O Errors Are Handled . . . 230
Using Generic and KGE Keys . . . 230
How to Perform "One–to–Many" Reads . . . 232
Working with "Batched" Input Files . . . 234
Working With Arrays . . . 237
Using Normalization to Process Arrays . . . 237
The NORMALIZE Parm . . . 240
File Definition Tips for Records with Arrays . . . 243
Normalizing Nested Arrays . . . 244
Normalizing Multiple, Non-Nested Arrays . . . 245
Normalizing only Certain Records . . . 247
Normalizing an Auxiliary Input File . . . 248
Normalization Errors . . . 248
How to Print a Variable Number of Lines Per Input Record . . . 249
Variable Number of Lines — Strategy 1 . . . 249
Variable Number of Lines — Strategy 2 . . . 254
Putting a Variable Number of Items on a Single Line . . . 257
Creating PC Files from Non-Spectrum Writer Reports . . . 258
Working with SMF Records . . . 263
Working with Date Fields . . . 269
Working with Time Fields . . . 272
Producing Files for Non-Standard PC Programs . . . 275

Producing Files for Mainframe Programs 280
How to "Subset" Mainframe Files 283
How to Sort Mainframe Files 283
Computing Percent of Totals 284
Creating Multiple Reports in a Single Run 289

Chapter 5. How to Make a Web Report 293
How to Create a Web Report 294
Writing Your Own HTML Tags 296
Experimenting with HTML Tags 297
Customizing the Web Report's Titles 298
Customizing the Web Report's Data Columns 301
Customizing Control Breaks and Grand Totals 303
Putting Graphics in Your Web Report 305
Putting Graphics in Your Report Title 305
Putting Graphics in the Body of Your Report 306
Putting Graphics at Control Breaks 308
Putting Hot Links in your Web Report 308
Using HTML Tables in your Web Report 312
Using Dynamic HTML Tags 315
Using the PRESCRIPT and POSTSCRIPT Options 318
Summary of Options for Web Reports 320
Common HTML Tags 321

Chapter 6. How to Define Your Input Files 325
How to Define a File 328
How to Use the FILE Statement — OS/390 328
How to Use the FILE Statement — VSE 331
How to Define a Field 333
How to Define a Character Field 333
How to Define a Numeric Field 335
Should You Define a Field as Character or Numeric? 339
How to Define a Date Field 340
How to Define a Time Field 344
How to Define a Bit Field 347
How to Specify a Field’s Column Heading 350
How to Specify a Field’s Location in a Record 350
Variably Located Record Segments 353
How to Define Arrays 355
How to Specify What File a Field Belongs To 356
How to Define a Field Created by a Data Exit 357
Keeping Your File Definitions in a Copy Library 360
Including the Definition Statements "In–Line" 360
Using the Spectrum Writer Copy Library 363
How to Use a Copy Library Alias 367
Defining One–Time Fields 368
Using Cobol and Assembler Record Layouts 369
Live Runs Using Cobol Record Layouts 370
Live Runs Using Assembler Record Layouts 372
Handling Date and Time Fields in Record Layouts 375
How Spectrum Writer Handles Cobol Arrays 377
Converting Cobol and Assembler Layouts to FIELD Statements 378
How to Copy Cobol and Assembler Record Layouts from Libraries 382

Mixing FIELD Statements with COBOL and ASM Statements . . . 383
The Starting Column of a Cobol or Assembler Layout . . . 384
The "Default Location" After a Cobol or Assembler Layout . . . 384
The Scope of the COBOL and ASM Statements . . . 384
Technical Notes on Cobol Support . . . 385
Technical Notes on Assembler Support . . . 387

Chapter 7. Working with Databases . . . 391
Using Spectrum Writer with DB2 Databases . . . 392
Using DB2 Data in Reports . . . 393
Using DB2 Data in PC Programs . . . 395
What Fields Are in Your DB2 Table? . . . 397
Using the WHERE Parm . . . 397
Using the ORDERBY Parm . . . 399
Using Multiple DB2 Tables . . . 400
Using Data from Three DB2 Tables . . . 403
WHERE Parm Syntax . . . 405
Customizing Your DB2 Fields . . . 407
Saving DB2 File Definitions . . . 408
DB2 Setup . . . 409
DB2 Restrictions . . . 410

Chapter 8. Operating System Considerations . . . 411
OS/390 Operating System Considerations . . . 412
Execution JCL for Reports — OS/390 . . . 412
DD statements used by Spectrum Writer . . . 414
Execution JCL for PC and Mainframe Files — OS/390 . . . 415
Spectrum Writer PROC — OS/390 . . . 417
Output File Options — OS/390 . . . 417
Considerations for Runs with Multiple Outputs — OS/390 . . . 419
Setting Up File Definitions — OS/390 . . . 420
Copy Library DD — OS/390 . . . 423
Input File DDs — OS/390 . . . 423
Specifying Shop–Wide Options — OS/390 . . . 424
Completion Codes — OS/390 . . . 425
VSE Operating System Considerations . . . 425
Execution JCL for Reports — VSE . . . 427
Execution JCL for PC and Mainframe Files — VSE . . . 429
Output File Options — VSE . . . 431
Input File DLBL/TLBLs — VSE . . . 432
The Control Statement Listing — VSE . . . 433
The EXEC Statement’s SIZE Parm — VSE . . . 433
Specifying Sort Work Files — VSE . . . 434
Considerations for Runs with Multiple Outputs — VSE . . . 435
Setting Up File Definitions — VSE . . . 437
Completion Codes — VSE . . . 439

Chapter 9. General Syntax Rules . . . 441
Control Statements . . . 443
What Is a Control Statement? . . . 443
How to Write Control Statements . . . 443
How to Continue a Control Statement Onto Multiple Lines . . . 444
The Order of Control Statements . . . 444
How to Put Comments in Your Control Statements . . . 445

How to Put Page Breaks in the Control Listing . . . 446
Names of Files, Fields, and Records . . . 446
Rules for Assigning Names . . . 446
How to Make Field Names Unique . . . 447
How to Write Literals . . . 448
The Five Types of Data . . . 448
Character Literals . . . 448
Numeric Literals . . . 449
Date Literals . . . 449
Time Literals . . . 450
Bit Literals . . . 450
When Do You Need Quotes Around a Number? . . . 450
PICTURE Display Formats . . . 451
Examples of PICTUREs . . . 452
Showing Scaled Numbers with PICTUREs . . . 453
How PICTUREs Work . . . 455
Time PICTUREs . . . 458
Conditional Expressions . . . 459
How to Specify a Relation Condition . . . 460
Comparing Character Operands of Different Lengths . . . 462
Comparing Fields of Different Data Types . . . 463
Conditions Involving Explicit Literals . . . 464
How to Specify a Bit Field Condition . . . 465
How to Specify Multiple Conditions . . . 465
Conditional Expressions That Use AND . . . 465
Conditional Expressions That Use OR . . . 466
Conditional Expressions That Use Both AND and OR . . . 467
How to Shorten Long Expressions . . . 468
How to Negate Conditions . . . 469
Examples of Conditional Expressions . . . 470
Computational Expressions . . . 472
Operands in Computational Expressions . . . 473
Operators in Computational Expressions . . . 473
Order of Operations . . . 474
Examples of Computational Expressions . . . 474

Chapter 10. Control Statement Syntax . . . 477
Syntax Notation . . . 478
ASM Statement . . . 479
BREAK Statement . . . 481
COBOL Statement . . . 493
COLUMNS Statement . . . 498
COMPUTE Statement . . . 506
COPY Statement . . . 516
FIELD Statement . . . 521
FILE Statement . . . 531
FOOTNOTE Statement . . . 538
INCLUDEIF Statement . . . 540
INPUT Statement . . . 542
NEWOUT Statement . . . 554
OPTIONS Statement . . . 555
READ Statement . . . 578
SORT Statement . . . 595

TITLE Statement 602

Appendix A. Data Types 609
Character Data Types 609
Numeric Data Types 610
Date Data Types 611
Time Data Types 613
Bit Data Types 616

Appendix B. Display Formats 617
Default Display Formats 618
Display Formats for Any Type of Field 618
Numeric Display Formats 619
Date Display Formats 620
Time Display Formats 622

Appendix C. Built-In Fields 624
Character Built–In Fields 625
Numeric Built-In Fields 626
Date Built-In Fields 626
Time Built-In Fields 627

Appendix D. Built-In Functions 628
Functions that Return a Character Value 630
Functions that Return a Numeric Value 635
Functions that Return a Date Value 638
Functions that Return a Time Value 640
Functions that Return a Boolean (or Bit) Value 642

Appendix E. Error Indicators 644
Suppressing Error Indicators 646
Propagation of Error Indicators 647
Determining if a Field Is In Error 647

Appendix F. Files Used in Examples 648
Sample File Definitions 648
Sample Files' Raw Data 650

Appendix G. Speed-Up Tips 652
INCLUDEIF Statement 652
Conditional COMPUTE Statements 655
Compute Statements with RETAIN 656
Intermediate Computational Expressions 657
Intermediate Conditional Expressions 657
Read Statements with the MULTI parm 658
Use the STOPWHEN Parm for Non-Keyed Files 661
Replace an Auxiliary File with a "Table Lookup" 661
Clearing I/O Areas 662
Fine-Tuning the Sort 662
Development Cycle 663
Using Explicit Literals in Conditional Expressions 664

Appendix H. Sample Data Exit Programs 666
Sample Assembler Data Exit Program 666
Sample Cobol Data Exit Program 671

Appendix I. I/O Exits 673

Updates to This Manual **689**
Index **691**

List of Figures

Figure 1. Spectrum Writer Control Statements Used for Making Reports 33
Figure 2. A report produced with just two control statements 36
Figure 3. An employee directory produced with only two control statements 38
Figure 4. Using an INCLUDEIF statement to specify which records to include in a report 41
Figure 5. Including records in a report if either of two conditions is true 43
Figure 6. Using the COMPUTE statement to create numeric fields 47
Figure 7. Using the COMPUTE statement to create character fields. 49
Figure 8. Assigning values to computed fields based on conditions 51
Figure 9. Using the TITLE statement to specify your own titles 54
Figure 10. Using slashes to align the different parts of a title 56
Figure 11. Using override display formats, column headings and column widths 59
Figure 12. Using a SORT statement to specify the sort order of a report 63
Figure 13. Using the BREAK statement to create a control break 66
Figure 14. A BREAK statement that produces a page break 68
Figure 15. A report that prints statistical information at control breaks and the Grand Totals 70
Figure 16. A report with two levels of control breaks 71
Figure 17. Producing a summary report 74
Figure 18. A report that uses only the primary input file 78
Figure 19. A report that uses a READ statement to specify an auxiliary input file 80
Figure 20. A report that uses two READ statements to specify two auxiliary input files 81
Figure 21. Spectrum Writer Control Statements for making PC Files 87
Figure 22. An Excel spreadsheet containing the entire mainframe SALES-FILE took only 3 statements 89
Figure 23. A Quattro Pro employee directory produced with just three control statements 91
Figure 24. Using an INCLUDEIF statement to specify which records to include in a PC file 94
Figure 25. Using the COMPUTE statement to create numeric fields for a PC file 99
Figure 26. Using the COMPUTE statement to create character fields for a PC file 101
Figure 27. Assigning values to computed fields based on conditions 103
Figure 28. Using a SORT statement to specify the sort order of a PC file 106
Figure 29. Using the BREAK statement to create a control break with subtotals in a PC file 109
Figure 30. Using FOOTING parms to customize the total row and create blank rows 111
Figure 31. A Paradox table containing only summary data 114
Figure 32. A spreadsheet that uses only the primary input file 118
Figure 33. A spreadsheet that uses a READ statement to specify an auxiliary input file 119
Figure 34. A spreadsheet that uses two READ statements to specify two auxiliary input files 121
Figure 35. Using spacing factors and literal texts in the COLUMN statement 127
Figure 36. Specifying your own column headings 131
Figure 37. Specifying the width of report columns 136
Figure 38. Customizing the way dates and numbers are formatted in a report 138
Figure 39. A report with international formatting options 142
Figure 40. A report that blanks out repeating values 145
Figure 41. Specifying how to justify data within the report columns 147
Figure 42. Specifying which columns to total 149
Figure 43. Using multiple COLUMN statements to print multi-line reports 152
Figure 44. A report with a bar graph column 155
Figure 45. A report with vertical lines separating the columns 157
Figure 46. A report title that includes data from a file 162
Figure 47. A title that shows the current day of the week, date, time and page number 164

Figure 48. Using width, display format and justification parms in the title 166
Figure 49. A report with left and right title parts 170
Figure 50. A report with left, center, and right title parts 171
Figure 51. Titles with the date, 24-hour time, and page number on the left side of the report 172
Figure 52. A title with date (spelled out), time, and page number on the right side of report 174
Figure 53. Using the FOOTNOTE statement to add footnotes to a report 176
Figure 54. A BREAK statement that requests a page break and resets the page number 179
Figure 55. A report with a customized total line at the control breaks 183
Figure 56. A report that prints statistical lines (average, maximum, minimum) at control breaks 187
Figure 57. Using the FOOTING parm to print a customized line at a control break 190
Figure 58. A report which prints a field's average value in a footing line 195
Figure 59. Printing a field's total, average, and maximum values on a single line 197
Figure 60. A report that prints the number of items in a control group 199
Figure 61. A report that prints control group headings 201
Figure 62. Using the DIVTOTS parm to get accurate percentages at control breaks 203
Figure 63. A report with two levels of control breaks 205
Figure 64. A report with customized Grand Totals 208
Figure 65. A summary report that uses two levels of control breaks 210
Figure 66. "Top 3 Sales in Region" report 213
Figure 67. Counting how many times various values occur in a file 215
Figure 68. Breaking down "count" statistics further 216
Figure 69. Subtotaling fields by a category (such as gender) 218
Figure 70. A Simple Crosstab Report 220
Figure 71. Another sample crosstab report 222
Figure 72. Control statements to produce the crosstab report on page 222 223
Figure 73. A report with multiple READ statements for the same file 225
Figure 74. A report with chained READ statements 227
Figure 75. A "one-to-many" report using the MULTI parm in a READ statement 233
Figure 76. An "batched" input file (with header and detail records) and its definition statements 235
Figure 77. A PC file produced from a batched input file (with header and details records) 236
Figure 78. SALES-HISTORY file containing an array 238
Figure 79. Normalizing the SALES-HISTORY file 239
Figure 80. A report that uses normalization to process an array 241
Figure 35. A sample file containing sales data for up to 6 sales per record 250
Figure 36. A report with "no strategy" to deal with unused array items 251
Figure 37. Strategy 1 — just add the SKIPZERODET option 252
Figure 38. Adding literal identifiers to variable lines 256
Figure 39. A typical mainframe report that has been written to a disk file 259
Figure 40. Spectrum Writer statements to define the "report file" shown above 259
Figure 41. Creating a Lotus 1-2-3 spreadsheet from a mainframe report 261
Figure 42. File definition of selected fields in SMF type 30 records 264
Figure 43. SMF "Daily ABEND" report 267
Figure 44. SMF "TSO Sessions" report 268
Figure 45. A standard comma-delimited PC File 277
Figure 46. An output file created with the MAINFRAME option 281
Figure 47. This step adds "region total records" to the file, and also creates a special sort key 285
Figure 48. Sorting the temporary dataset so that the regional totals come before the detail data 287
Figure 49. A report with "percent of total" columns 288
Figure 35. A basic Web report (viewed on Microsoft's Internet Explorer) 295
Figure 36. A Web report with customized titles 299
Figure 37. A Web report with two bold columns 302
Figure 38. A Web report with customized total lines 304

Figure 39. A Web report containing graphics in the title and body 307
Figure 40. Control statements used to create a Web report with “hot links” 309
Figure 41. Two screens from a Web report with “hot links” 310
Figure 42. the HTML output file for a Web report with “hot links” 311
Figure 43. A Web report that uses “tables” 312
Figure 44. Control statements used to create a Web report with “tables” 313
Figure 45. HTML file for a Web report with "tables" 314
Figure 46. A Web report that uses dynamic HTML tags 317
Figure 47. HTML file with dynamic HTML tags 318
Figure 48. A sample Spectrum Writer file definition 327
Figure 49. A report with FIELD statements that define character fields 334
Figure 50. A report with FIELD statements that define numeric fields 336
Figure 51. A report with FIELD statements that define date fields 342
Figure 52. A report with FIELD statements that define time fields 345
Figure 53. A report with FIELD statements that define bit fields 348
Figure 54. Different ways to define an array depending on how it will be processed 356
Figure 55. A Spectrum Writer report that does not use a copy library — OS/390 361
Figure 56. A Spectrum Writer report that does not use a copy library — VSE 362
Figure 57. A report which uses Spectrum Writer’s Copy Library — OS/390 365
Figure 58. A report which uses Spectrum Writer’s Copy Library — VSE 366
Figure 59. A report produced using a Cobol record layout 371
Figure 60. A report produced using an Assembler record layout 373
Figure 61. Creating true date and time fields from a Cobol layout 376
Figure 62. Converting a Cobol record layout to Spectrum Writer FIELD statements 380
Figure 63. A Spectrum Writer DB2 report 394
Figure 64. Using DB2 data in a Lotus 1-2-3 spreadsheet 396
Figure 65. Using the WHERE and ORDERBY parms 398
Figure 66. A report that uses data from two different DB2 tables 402
Figure 67. A report that uses data from three different DB2 tables 404
Figure 68. Sample Spectrum WriterSpectrum Writer JCL for reports - OS/390 413
Figure 69. Sample Spectrum Writer JCL for PC and Mainframe files — OS/390 416
Figure 70. Sample Spectrum Writer JCL for reports — VSE 428
Figure 71. Sample Spectrum Writer JCL for PC and Mainframe files — VSE 430
Figure 72. A report that uses a data exit program 667
Figure 73. Sample Data Exit Program Written in Assembly Language 668

Part 1.
User's Guide

Chapter 1. Introduction

Chapter Table of Contents

Chapter 1. Introduction 19

What Is Spectrum Writer? 20
Create Brand–New Reports in Minutes 20
Use Mainframe Data in Any PC Program 21
Create Web Reports and E-Mail Attachments 22
Create Custom Mainframe Files in Minutes 22
Ways that Spectrum Writer Benefits You! 22
Spectrum Writer Pays for Itself Fast! 24
Spectrum Writer Features 25

Chapter 1. Introduction

What Is Spectrum Writer?

Spectrum Writer is three powerful programs in one.

- It's an easy–to–use, full function **4GL report writer**.
- It's a powerful **PC–format utility**. Use its 4GL language to easily turn any mainframe data into PC files for use in all popular PC programs. Or create HTML files to display reports on web sites, or to send as e-mail attachments.
- It's also a **mainframe file formatting utility**. It's 4GL language lets you easily create your own custom mainframe output files.

Create Brand–New Reports in Minutes

Spectrum Writer makes it easy to produce custom reports from your company's existing files.

Programmer productivity increases dramatically with Spectrum Writer. To produce a new report without Spectrum Writer, a programmer has to write a new program in a language such as COBOL. The programmer must code all of the I/O routines, the selection logic, the computations, summarization, sorting, formatting, page breaks, titles, column

headings, etc. The process of coding, testing, and debugging takes many days, if not weeks. Then there's the whole cycle all over again when the users need "a few minor changes".

The easy alternative is to use Spectrum Writer. With Spectrum Writer, you no longer need to write detailed programming instructions. You simply describe the desired report to Spectrum Writer with a few simple control statements (much like SQL allows you to do with DB2 data). In fact, you can produce a complete report with Spectrum Writer using only *two* statements. Try that with COBOL! Add a few more statements and you can produce more complex reports.

With Spectrum Writer you'll have your results in *minutes*, instead of days or weeks. And if you need to change something later, modifications are a snap with Spectrum Writer.

Spectrum Writer also lets **end users get the information they need** with less intervention from programmers. Set up a model report for the users once — then let them modify and submit it over and over. If new selection criteria are needed in a report, or a different sort order or different title is wanted, *they* can make the changes themselves, without taking up a programmer's time at all. The end users get their results faster, and the programming staff has fewer interruptions. Everyone benefits with Spectrum Writer.

Use Mainframe Data in Any PC Program

Spectrum Writer's PC–formatting feature makes it easier than ever to use mainframe data in your favorite PC programs (such as Excel, Lotus 1–2–3, Access and Paradox, among many others).

Spectrum Writer is a great help for the PC users in your shop. Are users at your company *manually* keying data from mainframe reports into PC spreadsheets or databases? That's a

tedious, time–consuming process that is highly prone to errors. Spectrum Writer lets you **give accurate mainframe data to your PC users** in a format that's especially designed for their PC program. A few keystrokes is all it takes to "import" the data into their PC program. That means they can begin productive work right away.

Just moving data from the mainframe to a PC is easy. But being able to use that data in your PC software, easily and efficiently, is another matter. That's where Spectrum Writer comes in.

Spectrum Writer **lets you use "non–PC compatible" mainframe data** in your PC. This includes such things as bit fields, Julian dates, packed numbers, binary numbers, hexadecimal fields, etc. PC programs can't handle such data, but Spectrum Writer reformats these fields into standard ASCII data that your PC program can use.

Spectrum Writer lets you **choose the PC program you prefer**. Spectrum Writer knows the quirks of various PC program and automatically formats the data appropriately.

Create Web Reports and E-Mail Attachments

Spectrum Writer can create your report in HTML format. Perfect for uploading to Web sites for easy Intranet or Internet viewing. Or attach the HTML report to an e-mail and send it instantly to whomever you like. You can even customize your HTML file to include special formatting, custom fonts, graphics, hyperlinks and more.

Create Custom Mainframe Files in Minutes

Spectrum Writer creates mainframe output files just as easily as PC files. Use its 4GL language to: select the input records you want; combine data from multiple input files; optionally summarize data; sort data; etc. Then have Spectrum Writer write out the desired data in any format you choose. Use Spectrum Writer to easily convert binary fields to packed fields (or vice versa), to reformat date fields (perhaps changing YY dates to YYYY dates), etc. Add new computed fields to your output; or eliminate unneeded fields. You'll find a thousand and one uses for custom mainframe files once you see how easy it is to create them.

Ways that Spectrum Writer Benefits You!

Here are a few examples of the ways that Spectrum Writer's custom reports, PC files and mainframe files will:

- make **you** more productive!
- delight your **end–users!**
- impress your **boss!**

Easily Make Quality Production Reports

The reports produced by Spectrum Writer look every bit as professional as those produced by individual report programs. Titles are perfectly centered, or flush with the report margins. Column headings are neatly aligned above the data, and underlined. At control

breaks, totals are aligned under the numeric columns, with the name of the break field clearly identified, etc. This attention to detail means you can use Spectrum Writer to quickly produce your regular *production reports*. Its usefulness is not limited to just ad hoc reports.

Fast One–Time Queries

Spectrum Writer is also great for those frequent requests for "one–shot" runs. Now you'll be able to satisfy requests that there just wasn't time for without Spectrum Writer. You'll wonder how you ever got along without it.

Provide Reports for CICS Systems

Spectrum Writer is ideal for handling the batch reporting side of online CICS applications. Use your CICS system for online inquiries and updates. Use Spectrum Writer to produce production reports and custom queries from that system.

Save Money on Special Analyses

Without Spectrum Writer, what happens when a special study is needed? Someone probably ends up manually going through the "closest" existing report, copying the needed data onto paper or into a spreadsheet, performing manual calculations, etc. With Spectrum Writer, you can quickly deliver the exact report that's needed and reduce the amount of expensive manual effort required.

Reduce Your CPU Usage

Some programming tools are real "CPU Hogs." No wonder many systems programmers hesitate to let programmers develop new applications using them. But, because Spectrum Writer is written entirely in efficient assembly language, its reports run amazingly fast.

In many cases, there is no significant difference between Spectrum Writer's run time, and the run time of a COBOL program written to produce the same report. And when you consider the CPU cycles saved in *development* (fewer compiles, test runs, debugging, etc.), Spectrum Writer can actually *lighten* the load on your CPU.

Delight Your PC End–Users

When the users would really prefer to manipulate the mainframe data themselves, Spectrum Writer allows you to give it to them in PC format. The users can then process the mainframe data however they like in their spreadsheet, database or word processing program. And the programmers can get back to programming.

Spectrum Writer delights PC users with many exciting new possibilities. With mainframe data in their PCs, they'll be able to:

- perform "what if" calculations in PC spreadsheets
- maintain their own PC database, for personal access or network use
- print high quality charts
- create color graphics, overhead transparencies and slides for fabulous presentations

Spectrum Writer's PC files also make it easy for you to provide mainframe data to people without access to your mainframe. Copy the PC file to a diskette and send it to other departments in your company. Or, mail it to your offices around the world.

Perfect for Downsizing Applications

Use Spectrum Writer for one–time file conversions needed when downsizing mainframe applications to run on PC systems. Spectrum Writer converts the packed, binary, and bit fields to the kind of ASCII data that is needed on the PC system.

Reduce PC Download Time and Hard Disk Usage

Spectrum Writer reduces download time and hard disk usage by letting you download only the data you actually need (not the entire mainframe file). Why waste time and PC storage downloading records and fields that won't even be used?

Some PC–based products require you to download entire reports to the PC. Then, the PC program must process the entire, gigantic report just to extract the few lines of data that the PC user might actually need. Spectrum Writer lets you do the extraction on the mainframe, before you download the data.

Save Wasted Employee Time Caused by Slow PC Processing

Spectrum Writer eliminates much needless PC processing by moving that processing from the PC to the mainframe. Don’t bother with slow PC sorts. Let your mainframe perform the sort for you at mainframe speed. Then download the sorted file. Instead of summarizing data in your PC, let Spectrum Writer summarize it on the mainframe. Then just download the small summary file to your PC. **W**hy merge data from multiple files on your PC, where disk I/O is slow? Use Spectrum Writer to combine data from multiple mainframe files (or DB2 tables) into a single file before you download it to the PC.

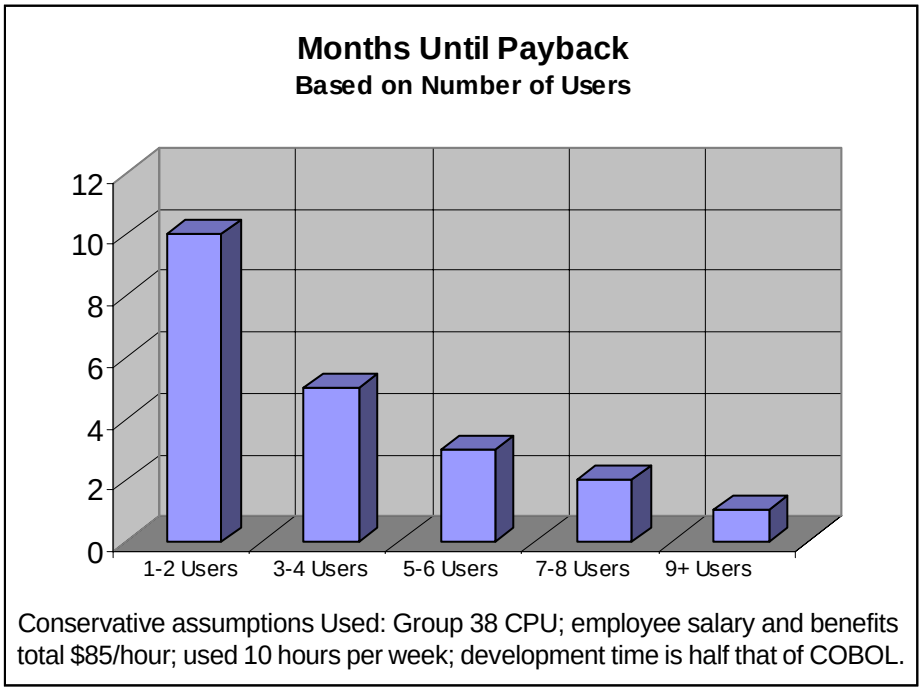
Spectrum Writer Pays for Itself Fast!

Spectrum Writer quickly pays its own way in a shop— maybe even the first time you use it!

Spectrum Writer greatly increases programmers' productivity. It slashes the programming effort required to create reports and PC files by as much as 90%. That means more completed projects, in less time, without an increase in staff. And if Spectrum Writer eliminates the need, *even once*, to bring in contract programmers to help overburdened staff with a project— you could recover its cost on that one project.

Spectrum Writer also increases the productivity of your PC users. If they are manually entering data now, the time savings will be *enormous*. But even if you have an existing download application, Spectrum Writer reduces the "dead–time" associated with it. You'll eliminate the wasted time spent downloading unnecessary data. And you'll shift much of the slower processing from the PC back up to the mainframe. You'll recover all the productivity your shop is losing every day to idle time when PC users are just waiting on their PCs. And with Spectrum Writer, there are no expensive PC components to purchase and maintain. All you need is Spectrum Writer and your existing file transfer facility.

Add together the cumulative value of the hours saved by the programming staff and your end–users. You'll see that it won't take long to recoup your modest investment in Spectrum Writer.



Conservative assumptions Used: Group 38 CPU; employee salary and benefits total $85/hour; used 10 hours per week; development time is half that of COBOL.

Spectrum Writer Features

Here are some of Spectrum Writer's major features:

- control statements use an easy, free format, English–like syntax that's easily learned by non–technical users
- user–friendly field names can be up to 70 characters long (unlike some report writers that restrict you to 8–byte names). This allows full compatibility with existing COBOL, PL/1 and Assembler data names.
- you can easily combine data from flat files, VSAM files and DB2 tables
- use your existing COBOL or Assembler record layouts instead of creating a data dictionary. Or, use Spectrum Writer's simple data dictionary for added functionality.
- no data definition required for DB2 tables — Spectrum Writer accesses the definition from your DB2 system
- automatically creates Web reports for viewing on Web browsers
- produces efficient internal machine code that is easy on your CPU
- can produce multiple reports (or output files) in a single pass of the input file
- produces output files for mainframe or Unix applications
- report lines are not limited to only 132 characters. Spectrum Writer can format a report as wide as your laser printer supports.
- automatically prints bar graphs

- ability to print full–page forms
- ability to skip to a new physical sheet of paper at control breaks (not just the next "page")
- has a logical default for every aspect of the report, from the report titles, to how to format numeric fields, to the layout of the Grand Total line
- allows complete control over formatting of numeric fields, including handling of special cases like telephone numbers, social security numbers, etc.
- formats dates in over 40 ways, including MM/DD/YY, DD/MM/YY, MM/DD/YYYY, etc. Or, with the month name spelled out, or abbreviated, and many more.
- has special numeric, date and time formatting options for international users
- allows complete control over report titles, column headings, and footnotes
- has a "forgiving" error philosophy which produces as much output as it reliably can, even when minor errors are encountered.
- has thorough, clear documentation, including a User's Guide in non–technical language for end–users
- validity–checks numeric data before processing it, to prevent S0C7 abends
- ability to display file data in hexadecimal format, for analyzing invalid data
- translates fields from EBCDIC to ASCII and vice verse
- supports full "boolean logic" (the use of AND, OR and NOT) in conditional expressions
- ability to scan free format fields, to see if a certain text appears anywhere within the field
- comparisons and computations are allowed among *all* numeric fields, (even if some are packed, some are binary, and others are character, etc.)
- comparisons are allowed among *all* date fields (even if some are Julian and some are Gregorian, some packed, others character, etc.)
- supports dates with 2–digit or 4–digit years
- supports century windowing for dates with 2–digit years
- supports every imaginable type of mainframe data, including over 30 kinds of date fields, and over 20 kinds of time fields.
- you can create your own new fields, optionally using different formulas depending on one or more conditions
- full mathematical calculations are supported when creating new fields, including the use of many built–in functions
- supports a full range of functions to manipulate string data, including powerful parsing features
- "compress" formatting features lets you, for example, compress separate city, state and ZIP fields into a normal formatted line format

- lets you use data from existing mainframe *reports* (rather than mainframe files) in PC programs
- handles complicated record layouts, including variably–located fields, fields located by pointer or pointer expressions, etc.
- supports records that contain arrays with varying number of entries
- lets you specify your own spreadsheet column headings, or use defaults
- easily summarizes data
- automatically computes statistics (such as total, average, maximum, minimum)
- allows an unlimited number of input files for a single report or PC file
- allows an unlimited number of control breaks
- allows an unlimited number of print lines per input record
- allows complete customization of control breaks
- allows complete customization of Grand Totals at end of report
- built–in fields provide the system date, time, jobname, etc.
- special features for speedy report development, such as limiting the number of records processed, or the number of report lines printed
- can limit input files to a certain key range to eliminate unnecessary I/O
- can halt input processing when a user-defined condition is met, to eliminate unnecessary I/O
- user exit interfaces for any special data handling required at the field level or record level
- user I/O exit interface allows access to files that use non-standard access methods
- prints end of job statistics, such as how many records read from each input file, and how many records included in report

Chapter 2. How to Request a Report

Chapter Table of Contents

Chapter 2. How to Request a Report 29

Lesson 1. How to Produce a Report in 5 Minutes 34
- How to Use the INPUT Statement 34
- How to Use the COLUMNS Statement 35
- Another 5–Minute Report Example 37
- Using Your Company's Files 37

Lesson 2. How to Specify Which Records to Include in Your Report 40
- How to Use the INCLUDEIF Statement 40
- How to Write Conditional Expressions 40

Lesson 3. How to Create Your Own Fields 46
- Creating Numeric Fields 46
- Creating Character Fields 48
- Assigning Values to Fields Based on Conditions 50

Lesson 4. How to Make Your Own Report Titles 53
- How to Use the TITLE Statement 53
- More Date and Time Features 55
- How to Align the Title 55
- How to Put File Data in the Title 55

Lesson 5. Changing the Format of your Report 58
- Using Display Formats 58
- Specifying Column Headings 60
- Specifying a Column's Width 60
- Multiple Overrides 60

Lesson 6. How to Specify the Report Order 62
- How to Use the SORT Statement 62
- Automatic Sorting 62

Lesson 7. How to Create Control Breaks 65
- How to Use the BREAK Statement 65
- How to Specify Control Break Spacing 67
- How to Print Statistics at a Control Break 67
- How to Produce Multiple Control Breaks 69

Lesson 8. How to Create Summary Reports 73
- How to Create a Summary Report 73

Lesson 9. How to Use Data from More Than One File 76
- How Auxiliary Input Files Are Processed 76
- How to Use the READ Statement 77
- "One-to-Many" Random Reads 79
- How to Use Multiple READ Statements 79

Chapter 2. How to Request a Report

This chapter teaches you how to use Spectrum Writer control statements to request custom reports.

Spectrum Writer's language is non–procedural, which means you just describe the *result* you want, not the programming steps needed to do it. That means you can produce new reports in a matter of minutes, rather than days or weeks.

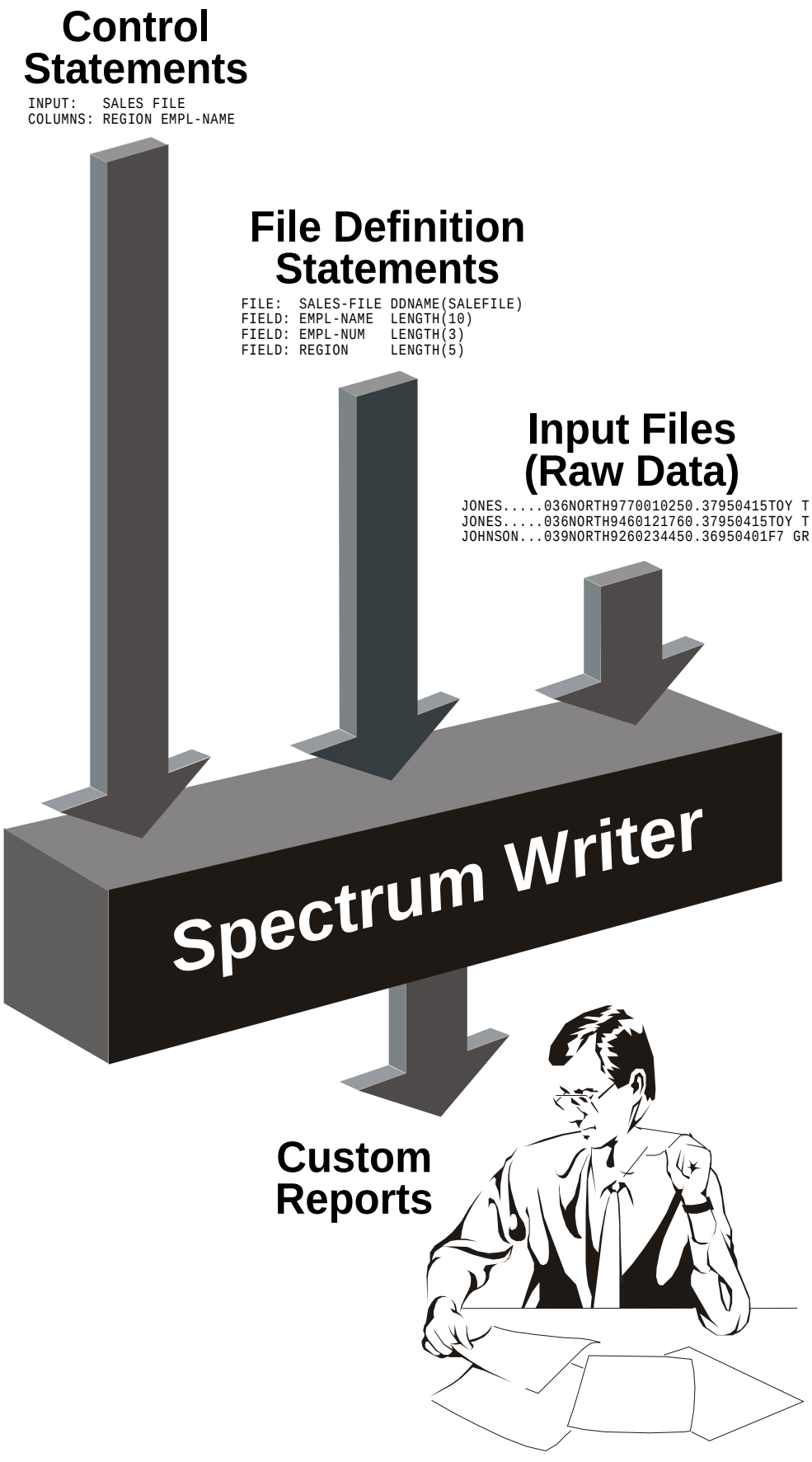
Describe your new report with a few simple "control statements". You can create a report with just *two* control statements. For example:

```
INPUT:    SALES-FILE
COLUMNS:  REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

The above statements are all that is needed to produce a complete report with Spectrum Writer. (See **Figure 2** on page 36.)

The box on page 33 lists all of the Spectrum Writer control statements, and tells you which aspect of the report each one deals with. The lessons in this chapter illustrate how to use these control statements.

Once you've written the necessary control statements, submit a batch job to execute Spectrum Writer. Spectrum Writer examines the control statements describing the report you want. It also automatically reads the appropriate "file definition" statements stored in a copy library. (These statements define the input files needed for your report.) Spectrum Writer then accesses the input file(s) and prepares the desired report.


Control Statements
INPUT: SALES FILE
COLUMNS: REGION EMPL-NAME
File Definition Statements
FILE: SALES-FILE DDNAME(SALEFILE)
FIELD: EMPL-NAME LENGTH(10)
FIELD: EMPL-NUM LENGTH(3)
FIELD: REGION LENGTH(5)
Input Files (Raw Data)
JONES.....036NORTH9770010250.37950415TOY T
JONES.....036NORTH9460121760.37950415TOY T
JOHNSON...039NORTH9260234450.36950401F7 GR
Spectrum Writer
Custom Reports

The remainder of this chapter is divided into nine easy lessons that explain how to use Spectrum Writer's control statements to create custom reports. After reading just the first lesson, you will be able to produce useful reports with Spectrum Writer. The other lessons introduce additional control statements, and explain their roles in producing increasingly sophisticated reports. It is not necessary to read all of the other lessons initially. Nor is it necessary to read the lessons in sequential order. Read the summaries below and decide which lessons you need for the kind of reports you want to produce.

Lesson 1. How to Produce a Report in 5 Minutes.
This lesson shows how to produce reports using just two simple control statements — the INPUT and the COLUMNS statements. You will use these two statements for almost every report you request.

Lesson 2. How to Specify Which Records to Include in Your Report.
This lesson shows how to use the INCLUDEIF statement to select which records will appear in your report.

Lesson 3. How to Create Your Own Fields.
This lesson shows you how to create your own fields by performing computations on existing fields. This is done with the COMPUTE statement.

Lesson 4. How to Make Your Own Report Titles.
This lesson introduces the TITLE statement, and shows how you can specify your own report titles.

Lesson 5. Changing the Format of your Report.
This lesson shows how you can customize the appearance of your report. It introduces some of the parms available in the COLUMNS statement. These parms let you change: column headings; column width; and the way dates and numbers are formatted.

Lesson 6. How to Specify the Report Order.
This lesson shows how to sort your reports into whatever order you want. The use of the SORT statement is explained.

Lesson 7. How to Create Control Breaks.
This lesson shows how to break a report up into sections, printing subtotals for each section. The use of the BREAK statement to request such "control breaks" is explained

Lesson 8. How to Create Summary Reports.
This lesson shows you how to turn a report with subtotals into a "summary report."

Lesson 9. How to Use Data from More Than One File.
This lesson shows how easy it is to read records from additional files when producing a report. By adding a single READ statement, you automatically have access to all of the fields from an additional file.

Keep in mind that these lessons show you the most common use of each control statement. Most control statements also have additional features that are not discussed in these lessons. Additional ways to use these control statements are discussed in Chapter 4, "Beyond the Basics." The complete syntax for each control statement is shown in Chapter 10, "Control Statement Syntax."

SPECTRUM WRITER CONTROL STATEMENTS (GROUPED BY FUNCTION)

Statements that Define How Input Data Looks

FILE	Defines a file
FIELD	Defines a field within a file
ASM	Defines a file using an Assembler record layout
COBOL	Defines a file using a Cobol record layout
COMPUTE	Computes a new user-defined field

Statements that Specify the Input Files to Use for a Report

INPUT	Specifies the primary input file
READ	Specifies an auxiliary input file

Statements that Describe the Body of a Report

INCLUDEIF	Specifies which input records to include in the report
COLUMNS	Specifies the report columns and column headings
TITLE	Specifies the report titles
FOOTNOTE	Specifies footnotes at the bottom of each page

Statements that Define the Report Order and Control Breaks

SORT	Specifies report order and, optionally, specifies control break fields
BREAK	Specifies control break processing

Miscellaneous Statements

OPTIONS	Specifies various special options, such as double spacing, or summary reports
NEWOUT	Indicates that subsequent statements will define a new report
COPY	Copies additional control statements for processing

Figure 1. Spectrum Writer Control Statements Used for Making Reports

Lesson 1. How to Produce a Report in 5 Minutes

This lesson teaches you how to produce a complete report using just two simple control statements. These statements are:

- the INPUT statement
- the COLUMNS statement

You only need these two statements to create a report with Spectrum Writer. For example:

```
INPUT:    SALES-FILE
COLUMNS:  REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Figure 2 shows a report created with just these two statements.

How to Use the INPUT Statement

Your company probably has many files stored on its disk drives and magnetic tapes. For example, the personnel department of your company probably has an employee file, containing information about each employee. The accounting department probably has numerous files, such as an accounts receivable file, an accounts payable file, etc. A sales department might have a sales file, with information about sales that have been made, and so forth.

The very first step in requesting a report is to tell Spectrum Writer which one of your company's files has the data needed for your report. Use the INPUT statement to do this. For example:

```
INPUT:  SALES-FILE
```

The above statement tells Spectrum Writer that you want to use a file named SALES–FILE as the input for your report. SALES–FILE is a sample file that we will use for many examples in this manual. The SALES–FILE contains information about the sales made by the employees of an imaginary company. Each record in this file contains data about one sale.

All Spectrum Writer control statements begin in column 1 with the *name* of the statement (for example, INPUT), followed immediately by a *colon*. What follows next will depend on the particular control statement involved. With an INPUT statement, you simply put the name of the file to be used as the input for the report. In the above example, we named the SALES-FILE.

How to Use the COLUMNS Statement

After identifying the input file to use, the next step is to tell Spectrum Writer which *fields* from that file you want to see in your report. Use the COLUMNS statement to do that. Each field named in this statement will appear as one column of data in the report. For example:

```
INPUT:    SALES-FILE
COLUMNS:  REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

The COLUMNS statement above tells Spectrum Writer that we want columns in our report that show the sales region, the employee name, the sales date, the sales time, the customer's name, the amount of the sale, and the tax amount.

> **Note:** Normally, reports are a maximum of 132 characters wide. You probably won't be able to fit *all* of a file's fields into that much space. Decide, then, which fields you need to see in your particular report, and put them in the COLUMNS statement. You may specify as many fields as there is room for in the report.

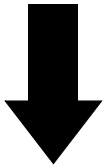
With just the two statements shown above, we have given Spectrum Writer everything it needs to produce a report. The report produced is shown in **Figure 2**.

You now see how easy it is to produce reports with Spectrum Writer. With just two simple statements we have produced an attractive report that has:

- a default **title** containing the name of the input file, as well as the date, time, day of the week, and page number
- the **columns of data** that we requested, appearing in the same order as we requested
- neat, underlined **column headings** identifying each column of data
- date, time and numeric fields that are properly **formatted**
- a **Grand Totals** line which shows totals for each of the numeric columns
- an **item count**, showing the number of records printed in the report

These Control Statements:

```
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
```



Produce this Report:

```
TUE  05/16/95    8:25 AM     DATA FROM SALES-FILE                  PAGE     1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT        TAX

SOUTH   JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL        101.38       6.09
WEST    BAKER      03/26/95 12:09:09 JACKS CAFE            137.00       8.22
EAST    MORRISON   03/29/95 15:30:22 STAR MARKET            44.35       2.66
EAST    MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY         29.65       1.78
EAST    SIMPSON    04/01/95 08:17:57 EUROPEAN DELI          14.99       0.90
NORTH   JOHNSON    04/01/95 17:02:47 VILLA HOTEL           234.45      14.07
NORTH   JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES          9.98       0.60
WEST    BAKER      04/12/95 14:31:12 JACKS CAFE            135.75       8.15
WEST    THOMAS     04/14/95 15:41:38 YOGURT CITY             9.98       0.60
NORTH   JONES      04/15/95 07:58:32 EZ GROCERY             10.25       0.62
NORTH   JONES      04/15/95 08:01:59 TOY TOWN               121.76       7.31
NORTH   JONES      04/15/95 13:52:41 TOY TOWN                10.25       0.62
SOUTH   JOHNSON    04/16/95 11:48:33 ACME BUILDING          500.00      30.00
EAST    SIMPSON    04/30/95 15:30:21 J & S LUMBER            23.87       1.43

*** GRAND TOTAL (14 ITEMS)                               1,383.66      83.05
```

Remarks:

- this report was produced from just two statements: the INPUT and the COLUMNS statements
- the data used in this report comes from the SALES-FILE
- the seven columns of data in the report correspond to the field names in the COLUMNS statement
- the default column headings used are the field names themselves, broken apart at each dash
- the report has a default title which includes the name of the input file
- the report has a Grand Total line showing totals for the two numeric columns
- the number of items listed in the report is shown
- the JCL used to produce this report is shown on page 413 (OS/390) or page 428 (VSE)

Figure 2. A report produced with just two control statements

Another 5–Minute Report Example

Now let's make another report, this time using a different input file. This time we will request a report from the EMPL–FILE. That's a sample employee file. We will print a simple employee directory from this file. We want the report to have columns showing employee number, last name, first name, sex, social security number, date hired, and their city and state. We only need the following two statements:

```
INPUT:   EMPL-FILE
COLUMNS: EMPL-NUM  LAST-NAME  FIRST-NAME  SEX  SOCIAL-SEC-NUM
         HIRE-DATE  CITY  STATE
```

The INPUT statement above specifies that the input file for our report will be the employee file (EMPL–FILE). The COLUMNS statement specifies the columns of data we want our report to have. Notice that we needed two lines for the COLUMNS statement in this example. You can continue a control statement onto as many lines as you like. Just leave at least 1 blank space at the beginning of each continuation line.

The report produced by the above statements is shown in **Figure 3**.

You have now seen two examples showing just how easy it is to request a report with Spectrum Writer. That's all there is to it! You now know enough to request basic reports from the files at your company. Just specify the file you wish to use in your report with an INPUT statement. And then specify the fields that you want to see in the report with a COLUMNS statement.

Using Your Company's Files

You may be wondering how Spectrum Writer knows the names of *your company's* files and fields. The answer is that your company's files are defined to Spectrum Writer by other control statements that are kept in a Spectrum Writer "copy library." For example, the statements used to define the sample files used in the preceding examples are shown in Appendix F, "Files Used in Examples" (page 648).

For a list of the file names and field names available for you to use, ask your programmer. They can print that information from the Spectrum Writer Copy Library, in a format similar to that shown in Appendix F.

If you already know the name of the *file* to use, you can use a "dummy" run to easily get a list of all of its fields. Just use an INPUT statement with the SHOWFLDS(YES) parm, like this:

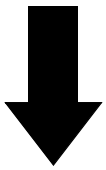
```
INPUT: SALES-FILE  SHOWFLDS(YES)
```

The above statement tells Spectrum Writer to print (in the control statement listing) a list of all of the fields defined for the SALES–FILE.

If a file that you need to use has not yet been defined, see Chapter 6, "How to Define Your Input Files" for information on doing that.

These Control Statements:

```
INPUT:   EMPL-FILE
COLUMNS: EMPL-NUM  LAST-NAME  FIRST-NAME  SEX SOCIAL-SEC-NUM
         HIRE-DATE  CITY  STATE
```



Produce this Report:

```
TUE  05/16/95    8:29 AM        DATA FROM EMPL-FILE                    PAGE     1

                                          SOCIAL
EMPL      LAST            FIRST             SEC       HIRE
NUM       NAME            NAME      SEX     NUM       DATE      CITY          STATE

036  JONES           JERRY            M  012-09-8765 01/31/80 SAN FRANCISCO    CA
037  JOHNSON         THOMAS           M  912-04-0334 06/21/75 SCOTTSDALE       AZ
039  JOHNSON         LINDA            F  004-77-9981 11/25/79 SANTA ROSA       CA
040  MACDONALD       RICHARD          M  889-79-0013 07/04/82 PLEASANTON       CA
041  SIMPSON         TIMOTHY          M  112-05-0456 12/01/82 ARCADIA          CA
042  MORRISON        MICHAEL          M  900-12-0556 11/30/79 GLENDALE         CA
043  CHRISTOPHERSON  MELISSA          F  415-09-0761 08/15/81 PHOENIX          AZ
044  BAKER           VIVIAN           F  878-19-0156 06/04/82 WALNUT CREEK     CA
045  THOMAS          MARTIN           M  776-83-8221 06/04/82 CONCORD          CA

*** GRAND TOTAL(9 ITEMS)
```

Remarks:

- the INPUT statement names the EMPL–FILE as the input file for this report
- the COLUMNS statement specifies which fields to print as columns in the report
- notice that we split the COLUMNS statement onto two lines, with the "continued" line beginning with at least one blank space

Figure 3. An employee directory produced with only two control statements

Summary

Here is a summary of what we learned in this lesson:

- an INPUT statement is needed to tell Spectrum Writer which input file to use for a particular report
- a COLUMNS statement is needed to tell Spectrum Writer what columns of data to print in your report
- by using just these two statements you can produce a complete report

The next lesson will teach you how to limit the records that are included in your report.

To Learn More

To learn more about writing control statements in general, see Chapter 9, "General Syntax Rules." In that chapter you will learn such things as:

- **how long** each control statement can be (page 443)
- how to **continue** control statements onto multiple lines (page 444)

There are some additional features associated with the INPUT and COLUMNS statements which we have not covered in this lesson. Some of these additional features are discussed in Lesson 5, "Changing the Format of your Report" (page 58). Other topics are discussed in Chapter 4, "Beyond the Basics." Some additional features are:

- how to specify your own **column headings** for a report (page 60 and page 130)
- how to make a column in the report **wider or narrower** (page 60 and page 135)
- how to change the way that **numbers, dates and times are formatted** in your report (page 58 and page 137)
- how to make a report column that contains a **literal text** (page 126)
- how to specify the number of **spaces** to leave between columns in your report (page 128)
- how to specify which numeric columns to include in the **Grand Totals** (page 148)
- how to print **multiple report lines** for each input record (page 151)
- how to **print all of the fields** from an input file in your report, without having to name each field individually (see page 158)
- how to produce reports that are **wider than 132 characters** (see page 417 or page 431)

The complete syntax for the INPUT and COLUMNS statements appears in Chapter 10, "Control Statement Syntax" (pages 542 and 498 respectively).

Lesson 2. How to Specify Which Records to Include in Your Report

This lesson teaches you how to select only certain records from the input file for inclusion in your report. The control statement discussed is:

- the INCLUDEIF statement

How to Use the INCLUDEIF Statement

The reports we produced in the previous lesson included all of the records found in the input file. When no INCLUDEIF statement is specified, Spectrum Writer defaults to including every record from the input file. For example, the report on page 36 included all sales from the SALES–FILE. And the report on page 38 listed all of the employees in the EMPL–FILE.

Often you want a report to include only selected records from the input file. Use the INCLUDEIF statement to tell Spectrum Writer to "include" a record in the report only "if" one or more conditions are met.

For example, assume that we want to print another list of sales from the SALES–FILE similar to the one on page 36. But this time we only want to print sales made by the employee named Jones. We would simply add the following INCLUDEIF statement to our other control statements:

```
INCLUDEIF:  EMPL-NAME = 'JONES'
```

The above INCLUDEIF statement tells Spectrum Writer to "include" records from the SALES-FILE "if" the EMPL–NAME field is equal to 'JONES'. Spectrum Writer still reads through the entire SALES–FILE, just like before. But now it examines each record before including it in the report. If the record's EMPL–NAME field contains the value 'JONES', then the record is included in the report. If the EMPL–NAME field contains any other value, then that record is not included in the report. **Figure 4** shows a report produced using the above statement. Only the sales made by Jones appear in that report.

The INCLUDEIF statement may appear anywhere after the INPUT statement. Only one INCLUDEIF statement is allowed per report, but it may contain as many conditions as you like.

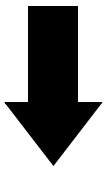
By the way, the INCLUDEIF statement can refer to any of the fields in the input file (as well as any COMPUTE field). You are not limited to just those fields that are listed in the COLUMNS statement.

How to Write Conditional Expressions

The INCLUDEIF statement simply contains a **conditional expression**. The complete rules for writing conditional expressions are explained beginning on page 459. Briefly, a conditional expression contains one or more "conditions," separated with words such as AND and OR. A **condition** usually involves comparing the contents of one field with the

These Control Statements:

```
INPUT:     SALES-FILE
INCLUDEIF: EMPL-NAME = 'JONES'
COLUMNS:   REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
```



Produce this Report:

```
TUE  05/16/95    8:26 AM     DATA FROM SALES-FILE                  PAGE     1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER       AMOUNT        TAX

NORTH  JONES      04/15/95 07:58:32 EZ GROCERY          10.25        0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN           121.76        7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN            10.25        0.62

*** GRAND TOTAL (3 ITEMS)                              142.26        8.55
```

Remarks:

- the report now includes only those records whose EMPL–NAME field is equal to 'JONES'

Figure 4. Using an INCLUDEIF statement to specify which records to include in a report

contents of another field, or with a literal value. Let's look at some more examples of INCLUDEIF statements and their conditional expressions.

> **Note:** If you are a programmer, you will notice that the syntax for conditional expressions is very similar to the syntax used in "IF statements" in COBOL, PL/1, and BASIC. If you are familiar with any of these languages, you should find it especially easy to write INCLUDEIF statements.

You may want your report to include all records which **do not** contain a certain value. Do this by specifying "not equal" in your condition. For example:

```
INCLUDEIF:  EMPL-NAME ¬= 'JONES'
```

The above statement specifies that the report should include all records from the input file whose EMPL–NAME field is not equal to 'JONES'.

> **Note:** In addition to ¬=, you can also use <> to indicate "not equal," like this:
>
> ```
> INCLUDEIF: EMPL-NAME <> 'JONES'
> ```

You may want to include a record in your report if **either of two conditions** is true. To do this, use an INCLUDEIF statement with two conditions, separated by the word OR. Consider the following statement:

```
INCLUDEIF:  EMPL-NAME = 'JONES'  OR  AMOUNT > 100
```

The above statement states that a record should be included in the report "if the EMPL–NAME field is equal to 'JONES' or if the AMOUNT field is greater than 100." The word OR indicates that records from the input file will be included if either one (or both) of the conditions are true. **Figure 5** shows a report that uses the above statement. All sales listed in that report were either made by Jones or were for an amount over $100.

Notice in the above statement that we enclosed 'JONES' in single quotation marks, while we did not use quotation marks around the 100. That is because EMPL–NAME is a character field, while AMOUNT is a numeric field. Character literals (such as 'JONES') must be enclosed in quotation marks. You can use either single (') or double (") quotation marks. But numeric literals (such as 100), as well as date and time literals, are *not* enclosed in quotation marks. Numeric literals also must not contain commas. (The rules for writing literals are thoroughly explained in "How to Write Literals" on page 448.)

As another example, you may want to include records in your report when **both of two conditions** are true. For example, let's say we want a listing only of sales that were made by Jones and that were also for an amount over $100. For this report, two conditions must both be true: the EMPL–NAME field must be equal to 'JONES' *and* the AMOUNT field must be over 100. Use the word AND to specify that both conditions must be true, like this:

```
INCLUDEIF:  EMPL-NAME = 'JONES'  AND  AMOUNT > 100
```

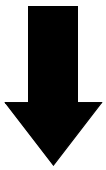
Now as Spectrum Writer reads each record from the input file, it will include a record in the report only "if the EMPL–NAME field is equal to 'JONES' and the AMOUNT field is greater than 100."

Here is an example of including records in a report based on the contents of a **date field**:

```
INCLUDEIF:  SALES-DATE  >  4/15/1995
```

These Control Statements:

```
INPUT:     SALES-FILE
INCLUDEIF: EMPL-NAME = 'JONES'  OR  AMOUNT > 100
COLUMNS:   REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
```



Produce this Report:

```
TUE  05/16/95    8:26 AM     DATA FROM SALES-FILE                  PAGE     1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER        AMOUNT          TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL      101.38         6.09
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE          137.00         8.22
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45        14.07
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE          135.75         8.15
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY           10.25         0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN            121.76         7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN             10.25         0.62
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING       500.00        30.00

*** GRAND TOTAL (8 ITEMS)                            1,250.84        75.08
```

Remarks:

- records are included in the report if either the EMPL–NAME field is equal to 'JONES' or the AMOUNT field is greater than 100

Figure 5. Including records in a report if either of two conditions is true

The above statement specifies that records should be included in the report only if their SALES–DATE field contains a date greater than (after) April 15, 1995.

> **Note:** You may be wondering if you need to use a different format for your date literals when you know that a particular date field is stored in a record as a Julian date (YYDDD format.) The answer is no. All date literals in your control statements should be written as MM/DD/YYYY (or MM/DD/YY). Spectrum Writer automatically takes care of any date conversions that may be required. Thus, you test Julian date fields just like all other date fields:
>
> ```
> INCLUDEIF: JULIAN-START-DATE >= 1/1/2000
> ```

Here is an example of including records in a report based on the contents of a **time field**:

```
INCLUDEIF:  SALES-TIME  <  17:00:00
```

The above statement specifies that records should be included in the report only if their SALES–TIME field contains a time less than (before) 17:00:00 (which is 5 PM).

> **Note:** You are allowed to omit the seconds in your time literals, if you prefer. When the seconds are not specified, zero seconds is assumed. Thus, you could also write the above statement this way:
>
> ```
> INCLUDEIF:  SALES-TIME  <  17:00
> ```

If your INCLUDEIF statement contains both the words OR and AND, you should use parentheses to indicate the order in which to perform the comparisons. Consider the following statement:

```
INCLUDEIF:  EMPL-NAME = 'JONES'  OR
            (SALES-DATE > 4/15/1995  AND  SALES-DATE < 4/30/1995)
```

In the above statement, records will be included if the EMPL–NAME field is equal to 'JONES' or if both of the SALES–DATE comparisons are true. The parentheses cause the two SALES–DATE comparisons to be treated as one condition. That condition is true if the SALES–DATE is greater than April 15, 1995 and is less than April 30, 1995.

> **Note:** In addition to the actual words AND and OR, you can also use the symbols "&" and" |", respectively, in your conditional expressions.

Summary

Here is a summary of what we learned in this lesson:

- use the INCLUDEIF statement when you want to include only certain records from the input file in your report
- the INCLUDEIF statement may contain one or more conditions, separated by the words AND or OR
- groups of conditions can be enclosed in parentheses, to indicate the order in which the comparisons should be performed

The next lesson will show you how to compute your own new fields for use in your report.

To Learn More

There are some additional features associated with the INCLUDEIF statement which we have not covered in this lesson. These additional features are discussed in "Conditional Expressions" (page 459). The additional features include:

- how to use the keyword **NOT** (or the symbol ¬) to negate a condition (page 469)
- how to **scan** a character field, to see if a certain text exists *anywhere* within the field (page 460)
- how to specify conditions based on **bit fields** (page 465)
- how to specify a condition based on a field's raw **hexadecimal** value (page 464)
- what to do if you want to specify **date literals** in DD/MM/YY or DD/MM/YYYY format (page 140), like this:

  ```
  INCLUDEIF: SALES-DATE > 15/4/1995
  ```

- how the **KEYRANGE** or **STOPWHEN parms** of the INPUT statement can be used to limit the records included in your run (page 542)

The complete syntax for the INCLUDEIF statement appears in Chapter 10, "Control Statement Syntax" (page 540).

Lesson 3. How to Create Your Own Fields

This lesson teaches you how to create your own fields to use in producing your report. The control statement discussed is:

- the COMPUTE statement

Sometimes the data you need for a report is not contained in the input file. Yet the necessary data might be easily computed from one or more fields which *are* in the input file. In such cases, simply create a new field by using the COMPUTE statement.

Creating Numeric Fields

A COMPUTE statement specifies the name of the new field to create and supplies a *computational expression* to use in assigning a value to that field. The complete rules for computational expressions are discussed in "Computational Expressions" (page 472). Generally, your expression will consist of one or more arithmetic operations performed on numeric fields or numeric literals.

For example, the sample SALES–FILE has numeric fields named AMOUNT and TAX. We can use the COMPUTE statement to create a new field containing the total amount due just by adding those two fields together, like this:

```
COMPUTE:  TOTAL-AMOUNT = AMOUNT + TAX
```

The above statement creates a new field named TOTAL–AMOUNT. It is computed by adding the AMOUNT field and the TAX field together. Now that the TOTAL–AMOUNT field has been created, we can use that field in *any way* that other fields can be used. For example, a computed field can be used: as a column in the body of the report; in the report titles; as a sort field; as a control break field; as part of a conditional expression (in the INCLUDEIF statement); even as an operand in subsequent COMPUTE statements to create other fields.

Figure 6 shows a report that uses the above COMPUTE statement.

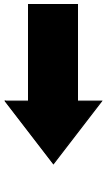
> **Note:** COMPUTE statements normally appear after the INPUT statement. A COMPUTE statement must appear before any other control statement that refers to the field being created. In **Figure 6**, the COMPUTE statement for TOTAL–AMOUNT had to come before the COLUMNS statement, since the COLUMNS statement referred to that field.

You can perform addition, subtraction, multiplication, and division in the COMPUTE statement. Use the +, –, * and / symbols, respectively. You may also use parentheses as needed to indicate the order in which the operations should be performed.

> **Note:** When performing subtraction, always put a **blank space before and after the minus sign**. Otherwise, the minus sign will appear to be part of a field name. Blanks are optional around the other arithmetic operators.

These Control Statements:

```
INPUT:   SALES-FILE
COMPUTE: TOTAL-AMOUNT     = AMOUNT + TAX
COMPUTE: SALES-COMMISSION = TOTAL-AMOUNT * .33
COLUMNS: EMPL-NAME  CUSTOMER  AMOUNT  TAX  TOTAL-AMOUNT  SALES-COMMISSION
```



Produce this Report:

```
TUE  05/16/95    8:26 AM        DATA FROM SALES-FILE                    PAGE     1

   EMPL                                                      TOTAL          SALES
   NAME     CUSTOMER          AMOUNT       TAX             AMOUNT      COMMISSION

JOHNSON    ACE ELECTRICAL        101.38        6.09          107.47         35.4651
BAKER      JACKS CAFE            137.00        8.22          145.22         47.9226
MORRISON   STAR MARKET            44.35        2.66           47.01         15.5133
MORRISON   A1 PHOTOGRAPHY         29.65        1.78           31.43         10.3719
SIMPSON    EUROPEAN DELI          14.99        0.90           15.89          5.2437
JOHNSON    VILLA HOTEL           234.45       14.07          248.52         82.0116
JOHNSON    MARYS ANTIQUES          9.98        0.60           10.58          3.4914
BAKER      JACKS CAFE            135.75        8.15          143.90         47.4870
THOMAS     YOGURT CITY             9.98        0.60           10.58          3.4914
JONES      EZ GROCERY             10.25        0.62           10.87          3.5871
JONES      TOY TOWN              121.76        7.31          129.07         42.5931
JONES      TOY TOWN               10.25        0.62           10.87          3.5871
JOHNSON    ACME BUILDING         500.00       30.00          530.00        174.9000
SIMPSON    J & S LUMBER           23.87        1.43           25.30          8.3490

*** GRAND TOTAL (    14 ITEMS)
                                1,383.66       83.05        1,466.71        484.0143
```

Remarks:

- the column heading used for computed fields is (by default) the field name itself, broken apart at each dash
- computed numeric fields receive Grand Totals just like other numeric fields

Figure 6. Using the COMPUTE statement to create numeric fields

As another example of a creating a numeric field, let's say we wanted to compute a sales commission for each sale. The commission will be 33% of the total value of the sale, including the tax. We could compute the sales commission with the following statement:

```
COMPUTE:  SALES-COMMISSION = TOTAL-AMOUNT * .33
```

This statement creates a new field called SALES–COMMISSION which is computed by multiplying TOTAL–AMOUNT by .33. Notice that we used the result of our previous COMPUTE statement to perform the computation in this statement.

Figure 6 (page 47) shows a report that uses the COMPUTE statement shown above.

In addition to the basic arithmetic operations, there are also a large number of built–in functions that you can use in the COMPUTE statement. These built–in functions allow you to perform more complex mathematical operations on numeric operands. A complete list of built–in functions is found in Appendix D, "Built-In Functions" (page 628).

Creating Character Fields

So far we have been creating numeric fields. Now let's consider how to create your own character fields. There is only one operation used in computing character fields. It is the **concatenation** operation. (Don't let that word scare you if it is new to you. "Concatenating" simply means "stringing together" two or more character fields.) The plus sign (+) is used as the symbol for concatenation. For example:

```
COMPUTE: WHOLE-NAME = LAST-NAME + FIRST-NAME
```

The above statement creates a new field named WHOLE–NAME. It is created by concatenating the contents of the LAST–NAME field and the contents of the FIRST–NAME field. The result is a single field which now contains both the first and last names of the employee. The new field will be 30 bytes long — the combined length of the two operands.

You can also concatenate more than two fields together. For example:

```
COMPUTE: MAILING-CODE = STATE + '-' + EMPL-NUM
```

This example creates a new field called MAILING–CODE which consists of the contents of the STATE field, followed by a dash, followed by the contents of the EMPL–NUM field.

In addition to the concatenation operation, there are also a number of built–in functions that can be used when creating character fields. For example, the #LEFT function can be used to extract the leftmost *n* bytes of a character field. Here is an example of how to use the #LEFT built–in function:

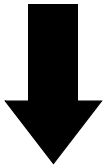
```
COMPUTE: FIRST-INITIAL = #LEFT(FIRST-NAME,1)
```

This statement creates a new character field which consists of only the first character (that is, the leftmost 1 byte) of the FIRST–NAME field.

Figure 7 shows a report that uses each of the above COMPUTE statements.

These Control Statements:

```
INPUT:    EMPL-FILE
COMPUTE:  WHOLE-NAME = LAST-NAME + FIRST-NAME
COMPUTE:  MAILING-CODE = STATE + '-' + EMPL-NUM
COMPUTE:  FIRST-INITIAL = #LEFT(FIRST-NAME,1)
COLUMNS:  EMPL-NUM  WHOLE-NAME  MAILING-CODE  FIRST-INITIAL  CITY  STATE
```



Produce this Report:

```
TUE  05/16/95    8:27 AM   DATA FROM EMPL-FILE                PAGE     1

EMPL             WHOLE               MAILING  FIRST
NUM               NAME                CODE    INITIAL      CITY        STATE

036  JONES          JERRY           CA-036     J    SAN FRANCISCO    CA
037  JOHNSON        THOMAS          AZ-037     T    SCOTTSDALE       AZ
039  JOHNSON        LINDA           CA-039     L    SANTA ROSA       CA
040  MACDONALD      RICHARD         CA-040     R    PLEASANTON       CA
041  SIMPSON        TIMOTHY         CA-041     T    ARCADIA          CA
042  MORRISON       MICHAEL         CA-042     M    GLENDALE         CA
043  CHRISTOPHERSON MELISSA         AZ-043     M    PHOENIX          AZ
044  BAKER          VIVIAN          CA-044     V    WALNUT CREEK     CA
045  THOMAS         MARTIN          CA-045     M    CONCORD          CA

*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the column heading used for computed fields is (by default) the field name itself, broken apart at each dash

Figure 7. Using the COMPUTE statement to create character fields.

Assigning Values to Fields Based on Conditions

Up until now we have been using "simple" COMPUTE statements. In a **simple COMPUTE statement**, the value of the new field is defined by a single computational expression.

But it is also possible to use conditional logic in a COMPUTE statement. In **conditional COMPUTE statements**, one of several different expressions will be used to assign a value to the new field. The expression that is used will depend on one or more conditions that you specify. Conditional COMPUTE statements can be very powerful tools in producing reports. Here is an example of a conditional COMPUTE statement:

```
COMPUTE: BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                 WHEN(HIRE-DATE >= 1/1/1980)  ASSIGN(TOTAL-SALES * .05)
```

The above statement creates a field named BONUS. However, in this example the BONUS field can be computed in one of two ways: for employees hired before January 1, 1980, the bonus is 8 percent of total sales (TOTAL–SALES * .08). But, for employees hired on or after January 1, 1980, the bonus is only 5 percent of total sales (TOTAL–SALES * .05).

When assigning a value to the BONUS field, Spectrum Writer evaluates the conditional expression in each WHEN parm. As soon as a WHEN expression is found that is true, the computational expression from the corresponding ASSIGN parm is used to assign a value to BONUS. (Any remaining WHEN parms are not evaluated.)

You may have as many pairs of WHEN and ASSIGN parms as you like in a COMPUTE statement. If none of the WHEN expressions are true, a value of zero will be assigned to the field. To assign some other value when none of the WHEN parms are true, you may use the ELSE parm. For example:

```
COMPUTE: BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                 ELSE                         ASSIGN(TOTAL-SALES * .05)
```

The above statement has the same effect as the previous example, but is a little simpler. It has only one WHEN expression. For employees whose hire date is before January 1, 1980, the bonus will be computed based on 8 percent. For all other cases, the bonus will be computed based on 5 percent.

You may also use conditional COMPUTE statements to create character fields. For example:

```
COMPUTE: TITLE = WHEN(SEX = 'M')  ASSIGN('MR')
                 ELSE             ASSIGN('MS')
```

The statement above creates a new field called TITLE. The contents of TITLE will be "MR" if the SEX field contains an "M", and "MS" otherwise.

Figure 8 shows a report that uses some of the conditional COMPUTE statements just discussed.

When defining character fields with a conditional COMPUTE statement, a value of spaces is assigned if none of the WHEN expressions are true and no ELSE parm is specified.

All of our examples so far have used just a single condition within the WHEN parm. You can, however, use any valid conditional expression within the WHEN parm. The conditional expression can contain as many different conditions as you like, separated with the words AND and OR, and optionally grouped with parentheses. (A conditional expression is the sort of expression that is allowed in the INCLUDEIF statement, as was described in the section

These Control Statements:

```
INPUT:    EMPL-FILE
COMPUTE:  BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                  WHEN(HIRE-DATE >= 1/1/1980)  ASSIGN(TOTAL-SALES * .05)
COMPUTE:  TITLE = WHEN(SEX = 'M')  ASSIGN('MR')
                  ELSE             ASSIGN('MS')
COLUMNS:  TITLE  LAST-NAME  FIRST-NAME  SEX  HIRE-DATE  TOTAL-SALES  BONUS
```

Produce this Report:

```
TUE  05/16/95    8:29 AM        DATA FROM EMPL-FILE                      PAGE     1

           LAST            FIRST            HIRE        TOTAL
TITLE      NAME            NAME        SEX  DATE        SALES          BONUS

 MR   JONES           JERRY            M  01/31/80       42,509.89        2,125.4945
 MR   JOHNSON         THOMAS           M  06/21/75       86,999.24        6,959.9392
 MS   JOHNSON         LINDA            F  11/25/79       75,023.55        6,001.8840
 MR   MACDONALD       RICHARD          M  07/04/82        2,560.98          128.0490
 MR   SIMPSON         TIMOTHY          M  12/01/82        8,723.88          436.1940
 MR   MORRISON        MICHAEL          M  11/30/79       98,054.99        7,844.3992
 MS   CHRISTOPHERSON  MELISSA          F  08/15/81       47,665.31        2,383.2655
 MS   BAKER           VIVIAN           F  06/04/82       92,125.89        4,606.2945
 MR   THOMAS          MARTIN           M  06/04/82       60,193.49        3,009.6745

*** GRAND TOTAL (9 ITEMS)                                513,857.22       33,495.1944
```

Remarks:

- the BONUS field is calculated differently, depending on the contents of the HIRE–DATE field
- the value assigned to the TITLE field is based on the contents of the SEX field

Figure 8. Assigning values to computed fields based on conditions

"How to Write Conditional Expressions" on page 40. The complete rules for writing conditional expressions are given in "Conditional Expressions" on page 459.)

Additional examples of COMPUTE statements are shown beginning on page 506.

Summary

Here is a summary of what we learned in this lesson:

- the COMPUTE statement is used to create new fields
- a **simple COMPUTE statement** assigns the result of a single computational expression to a new field
- a **conditional COMPUTE statement** uses one of several different computational expressions, depending on the conditions that you specify

The next lesson will show you how to specify your own report titles.

To Learn More

There are some additional features associated with the COMPUTE statement which we have not covered in this lesson. Some of these additional features are discussed under the COMPUTE statement in Chapter 10, "Control Statement Syntax" (page 506). Other additional features are discussed in Chapter 4, "Beyond the Basics." Examples of additional topics include:

- how to create **date** fields (page 514)
- how to create **time** fields (page 272)
- how to create **bit (boolean)** fields (page 514)
- how to specify how many **decimal places** a numeric or time field should contain (page 511)
- how to specify **column headings** for the fields you create (page 511)
- how to specify how your field should be **formatted** when it is printed in a report (page 510)
- how to specify whether a numeric or time field should be totalled in the **Grand Totals** line at the end of the report (page 148)
- how to **retain the previous value** of a COMPUTE field in certain cases (page 234)

The complete syntax for the COMPUTE statement appears in Chapter 10, "Control Statement Syntax" (page 506).

Lesson 4. How to Make Your Own Report Titles

This lesson teaches you how to specify your own report titles. The control statement discussed is:

- the TITLE statement

How to Use the TITLE Statement

As we've seen in the previous lessons, a TITLE statement is not required to produce a report. If you do not supply a TITLE statement when requesting your report, Spectrum Writer provides a default title.

To specify your own report titles, simply use one or more TITLE statements. For each TITLE statement you supply, Spectrum Writer will print one title line at the top of each page of the report. TITLE statements may appear anywhere after the INPUT statement.

After the word TITLE and the colon, enclosed your desired title text in either single or double quotation marks. For example:

```
TITLE:  'ABC COMPANY -- RECENT SALES'
```

Note: If your title is too big to fit on a single line, you may continue it onto additional lines. See "How to Continue a Control Statement Onto Multiple Lines" (page 444) for information on continuing control statements.

You will probably want to include the date and page number in your titles. Do this by using the special built–in fields named #TODAY and #PAGENUM. (Don't let the pound sign scare you. All of Spectrum Writer's built–in field names begin with this character. This is to help distinguish them from fields in your own files that may have similar names.)

When using #TODAY and #PAGENUM in your TITLE statement, do not enclose them in quotation marks. Anything enclosed in quotation marks is printed *as is* in the title. Anything not within quotation marks must be the name of a field, whose *contents* you want in the title. The words #TODAY and #PAGENUM are the names of built-in fields, whose contents are the system date and the current page number. Here is an example of specifying titles that contain the date and page number:

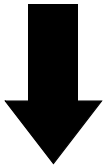
```
TITLE:  'ABC COMPANY -- RECENT SALES'
TITLE:  #TODAY
TITLE:  'PAGE' #PAGENUM
```

The three TITLE statements above result in three title lines in the report. The first title line is the literal text "ABC COMPANY – RECENT SALES". The second title line just contains the current date. The third title line contains the word "PAGE" followed by the page number itself. This third title line illustrates a new point: a TITLE statement can contain more than one item. In this case, it contains one literal text ('PAGE') and one field name (#PAGENUM).

Figure 9 shows a report produced using the above TITLE statements. Notice that the titles are automatically centered over the report.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'ABC COMPANY -- RECENT SALES'
TITLE:   #TODAY
TITLE:   'PAGE' #PAGENUM
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                        ABC COMPANY -- RECENT SALES
                                  12/01/95
                                 PAGE    1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT      TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL        101.38      6.09
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE            137.00      8.22
EAST   MORRISON   03/29/95 15:30:22 STAR MARKET            44.35      2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY         29.65      1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI          14.99      0.90
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL           234.45     14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES          9.98      0.60
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE            135.75      8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY             9.98      0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY            10.25      0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN              121.76      7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN               10.25      0.62
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING         500.00     30.00
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER           23.87      1.43

*** GRAND TOTAL (14 ITEMS)                             1,383.66     83.05
```

Remarks:

- this report has three title lines, corresponding to the three TITLE statements
- the second title line simply contains the current date (#TODAY)
- the third title line contains the literal word "PAGE" followed by the page number (#PAGENUM)
- all title lines are centered over the report

Figure 9. Using the TITLE statement to specify your own titles

More Date and Time Features

When you use #TODAY in your title, Spectrum Writer formats it in the standard default date format (MM/DD/YY). If you want to **spell out the month name** in the date, specify the LONG1 "display format" after #TODAY, like this:

```
TITLE: #TODAY(LONG1)
```

The above statement would cause, for example, "DECEMBER 1, 1995" to appear in the title, rather than "12/01/95". The report in **Figure 10** uses the LONG1 display format. The use of LONG1 and other display formats is discussed in more detail in "How to Change the Appearance of Items in the Title" (page 165). For a complete list of display formats to choose from when formatting dates in your titles, see Appendix B, "Display Formats" (page 617).

In addition to the current date, you can also use the built–in fields #TIME and #DAYNAME in your TITLE statement. These allow you to print the time of day and the day of the week in your titles.

Figure 10 also illustrates the #TIME built–in field.

How to Align the Title

What if we want just a single title line that contains the date, time and the page number along with our literal text? The following example shows how to do that:

```
TITLE: #TODAY #TIME  /  'ABC COMPANY – RECENT SALES'  /  'PAGE' #PAGENUM
```

Notice that the above TITLE statement contains two **slashes** (/). These are used to separate the title line into three parts. When slashes are not used (as in the previous examples), the whole title is simply *centered* over the report. But when slashes are used, the first part of the title (#TODAY and #TIME, in the case above) is aligned with the *left* edge of the report. The middle part (the literal text) is *centered* over the report. The last part ("PAGE" and #PAGENUM) is aligned with the *right* edge of the report. The use of slashes in the TITLE statement gives you the maximum control over how your title lines look.

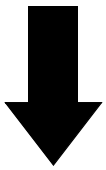
Figure 10 shows a sample report that uses slashes to align a three-part title. **Figure 14** (page 68) illustrates the alignment of a two-part title.

How to Put File Data in the Title

As mentioned earlier, TITLE statement text that is enclosed within quotation marks will appear "as is" in the title. You can also put field names (without quotation marks) in the TITLE statement. A field name can be one of the special built-in fields, such as #PAGENUM. Or it can be the name of a regular field from the input file (or even a COMPUTE field). When inserting the contents of a data field into the title, Spectrum Writer uses the data found in the first record used on the current report page. **Figure 14** (page 68) shows an example of including file data in a title.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   #TODAY(LONG1)  #TIME  /  'RECENT SALES'  /  'PAGE'  #PAGENUM
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
DECEMBER 1, 1995    8:27 AM      RECENT SALES                      PAGE    1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER        AMOUNT        TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL      101.38        6.09
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE         137.00        8.22
EAST   MORRISON   03/29/95 15:30:22 STAR MARKET         44.35        2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY      29.65        1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI       14.99        0.90
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL        234.45       14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES       9.98        0.60
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE         135.75        8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY          9.98        0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY          10.25        0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN           121.76        7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN            10.25        0.62
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING      500.00       30.00
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER        23.87        1.43

*** GRAND TOTAL (14 ITEMS)                          1,383.66       83.05
```

Remarks:

- the two slashes divide the TITLE statements into three parts
- the first part (the date and time) is left aligned over the report
- the second part (the name of the report) is centered over the report
- the third part (the page number) is right aligned over the report
- the LONG1 "display format" causes the month name to be spelled out in the date

Figure 10. Using slashes to align the different parts of a title

Summary

Here is a summary of what we learned in this lesson:

- use the TITLE statement to specify **your own titles** for a report
- if more than one TITLE statement is used, the title lines print in the **same order** in which the TITLE statements appear
- use Spectrum Writer's built–in fields to include the **date, time, day of the week, and page number** in your titles
- use slashes to separate your title into **left, center, and right** aligned parts

The next lesson will teach you how to customize the formatting of your report.

To Learn More

There are some additional features associated with the TITLE statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional features include:

- how to change the way the **dates, times and numbers are formatted** in the title (page 165)
- how to use **any combination** of left aligned, centered, and right aligned title parts (page 168)
- how to include the **jobname** in your title (page 163)
- how to print "footnotes" at the **bottom** of each page of the report (page 175)

The complete syntax for the TITLE statement is given in Chapter 10, "Control Statement Syntax" (page 602).

Lesson 5. Changing the Format of your Report

This lesson teaches you how to specify your own formatting options for a report. The formatting options discussed are:

- display formats
- column headings
- column widths

Using Display Formats

Spectrum Writer provides many "display formats" that you can choose from when displaying fields in a report. A complete list of display formats is found in Appendix B, "Display Formats" (page 617). When no display format is specified (as in most of the examples in the previous lessons), Spectrum Writer uses a default format. To specify your own display format, just place it in parentheses after the appropriate field name. (Do *not* leave a space between the field name and the open parenthesis.) Display formats are allowed in most statements. For example:

```
TITLE:   #TODAY(LONG1)
COLUMNS: SALES-DATE(SHORT3)  SALES-TIME(HH-MM)   AMOUNT(DOLLAR)
```

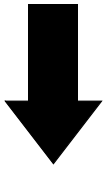
The above statements specify a display format for each field:

- the #TODAY field (in the title) will be formatted in Spectrum Writer's LONG1 format (that is, as MMMMMMMMM DD, YYYY).
- the SALES–DATE column in the report will be formatted in the SHORT3 format (that is, DD MMM YY).
- the SALES–TIME field will be formatted in the HH–MM format (that is, without the seconds). The time will be rounded to the nearest minute and formatted as HH:MM.
- the AMOUNT field will be formatted as a dollar value, with a floating dollar sign

Figure 11 shows a report that illustrates these display formats.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   #TODAY(LONG1)  /  'EXAMPLES OF SPECIAL FORMATTING'  /  #PAGENUM
COLUMNS: REGION   EMPL-NAME('SALES PERSON')  SALES-DATE(SHORT3)
         SALES-TIME(HH-MM)  CUSTOMER  AMOUNT(DOLLAR)  TAX(5)
```



Produce this Report:

```
DECEMBER 1, 1995     EXAMPLES OF SPECIAL FORMATTING                    1

                      SALES   SALES
REGION SALES PERSON   DATE    TIME   CUSTOMER        AMOUNT       TAX

SOUTH   JOHNSON     12 MAR 95 10:25 ACE ELECTRICAL          $101.38  6.09
WEST    BAKER       26 MAR 95 12:09 JACKS CAFE              $137.00  8.22
EAST    MORRISON    29 MAR 95 15:30 STAR MARKET              $44.35  2.66
EAST    MORRISON    30 MAR 95 19:06 A1 PHOTOGRAPHY           $29.65  1.78
EAST    SIMPSON     01 APR 95 08:18 EUROPEAN DELI            $14.99  0.90
NORTH   JOHNSON     01 APR 95 17:03 VILLA HOTEL             $234.45 14.07
NORTH   JOHNSON     05 APR 95 14:33 MARYS ANTIQUES            $9.98  0.60
WEST    BAKER       12 APR 95 14:31 JACKS CAFE              $135.75  8.15
WEST    THOMAS      14 APR 95 15:42 YOGURT CITY               $9.98  0.60
NORTH   JONES       15 APR 95 07:59 EZ GROCERY               $10.25  0.62
NORTH   JONES       15 APR 95 08:02 TOY TOWN                 $121.76  7.31
NORTH   JONES       15 APR 95 13:53 TOY TOWN                  $10.25  0.62
SOUTH   JOHNSON     16 APR 95 11:49 ACME BUILDING            $500.00 30.00
EAST    SIMPSON     30 APR 95 15:30 J & S LUMBER              $23.87  1.43

*** GRAND TOTAL (14 ITEMS)                                 $1,383.66 83.05
```

Remarks:

- The display formats (LONG1, SHORT3, HH–MM and DOLLAR) specify how the data is formatted in the report
- The override column heading changes the column heading for the EMPL–NAME field
- The override width parm makes the TAX column only 5 bytes wide
- Changes made to the detail line formatting are also reflected in the Grand Total line

Figure 11. Using override display formats, column headings and column widths

Specifying Column Headings

Another way to customize your report is with override column headings. You remember that Spectrum Writer uses the field name itself as the default column heading. To specify your own column heading, just place the desired text in parentheses after the appropriate field name in the COLUMNS statement. For example:

```
COLUMNS: EMPL-NAME('SALES PERSON')
```

In the above statement, we specified our own column heading for the EMPL–NAME field. As you can see in the report in **Figure 11** (page 59), the EMPL–NAME column now has "SALES PERSON" as its column heading.

> **Note:** To break your column heading text into multiple lines, use the vertical bar (|) as a line separator. For example:
>
> ```
> COLUMNS: EMPL-NAME('SALES|PERSON')
> ```

The above statement would result in a two-line column heading for the EMPL-NAME column. The word SALES would be stacked over the word PERSON.

> **Note:** The vertical bar is the "Shift 1" key on most mainframe terminals. When working at a PC running terminal emulation software, you will probably not see a key with this symbol on it. (The "pipeline" character is *not* the same as the vertical bar.) Some emulator programs use the right–hand square bracket key (]) to send a vertical bar to the mainframe.

Specifying a Column's Width

One other way to customize your report is to specify a column width for a particular column. When no column width is specified, Spectrum Writer chooses a default column width. You may want a larger column width (to hold larger numeric values, for example). Or, you may want a smaller column width (to save space so you can squeeze more columns into your report). Just specify the desired column width in parentheses after the field name. For example:

```
COLUMNS: TAX(5)
```

The above statement tells Spectrum Writer to make the TAX column just 5 bytes wide in the report. This is also illustrated in the report in **Figure 11** (page 59).

Multiple Overrides

You can specify more than one override for a single field. Their order within the parentheses is not important. Just separate the overrides with spaces and/or commas. For example, the following statement specifies an override column heading, display format and column width:

```
COLUMNS:  AMOUNT('AMOUNT OF SALES', DOLLAR, 8)
```

Summary

Here is a summary of what we learned in this lesson:

- use an override **display format** to change the way a field is formatted in a report
- use override **column headings** to change the column headings in a report
- specify a **column width** to change the width of a column in a report
- each of these overrides should be **put in parentheses** after the appropriate field name

The next lesson will teach you how to sort your report into whatever order you want.

To Learn More

There are many additional ways to change the format of your report. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional formatting features include:

- how to **left-justify, center or right-justify** data within its column (page 146)
- how to **blank out repeating values** in a column(page 144)
- how to **blank out zero values** (page 129)
- how to change the **spacing between columns** in a report (page 128)
- how to use a **character other than the vertical bar** (|) to separate column headings into multiple lines (page 130)
- how to change the default display format for **all fields** in a report (page 562)
- how to format reports using **international conventions** (page 140)

The complete syntax for the COLUMNS statement is given in Chapter 10, "Control Statement Syntax" (page 498).

Lesson 6. How to Specify the Report Order

This lesson teaches you how to sort your report into any order you want. The control statement discussed is:

- the SORT statement

How to Use the SORT Statement

When no SORT statement is specified, Spectrum Writer defaults to printing the report records in their original input file order. For example, the records in the sample SALES–FILE are stored in sales date order. Therefore, the sales reports in the previous lessons (for example, on page 59) all appeared in sales date order. The EMPL–FILE sample file is a VSAM file stored in EMPL–NUM order. Therefore, all previous reports from that file have been in employee number order (for example, the report on page 38).

To print a report in a different order, just add a SORT statement. The SORT statement can appear anywhere after the INPUT statement. Only one SORT statement is allowed per report, but it may contain as many "sort fields" as you like. Spectrum Writer will sort your report on all of the sort fields.

For example, let's request a report from the SALES–FILE and sort it on three fields:

```
SORT:  REGION  EMPL-NAME  SALES-DATE
```

To begin with, the report will be sorted according to the first sort field — REGION. If there are multiple records for the same REGION, then those records will be further sorted using the second sort field, EMPL–NAME. Records having the same value for both the REGION and the EMPL–NAME fields will be further sorted on the third sort field — SALES–DATE.

Figure 12 shows a report produced with the above statement.

By the way, the SORT statement can refer to any of the fields in the input file (as well as any COMPUTE field). You are not limited to just the fields that are listed in the COLUMNS statement.

By default, Spectrum Writer sorts reports into **ascending order** on each sort field. If you want to sort the report into **descending order** for a field, put the DESCENDING parm (or just DESC) in parentheses immediately after the field name. For example, to sort a sales report into reverse employee number order, you could use this SORT statement:

```
SORT:  EMPL-NUM(DESC)
```

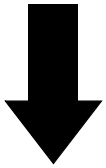
Automatic Sorting

If you prefer, you can let Spectrum Writer *automatically* sort your report for you. To have your report automatically sorted on the first 5 columns of data, simply specify the AUTOSORT option, like this:

```
OPTIONS: AUTOSORT
```

These Control Statements:

```
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
TITLE:   'RECENT SALES'
TITLE:   'SORTED BY REGION, EMPLOYEE NAME, AND SALES DATE'
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                                RECENT SALES
                SORTED BY REGION, EMPLOYEE NAME, AND SALES DATE

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/95 15:30:22 STAR MARKET           44.35      2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY       29.65      1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI        14.99      0.90
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER         23.87      1.43
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45     14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES        9.98      0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY           10.25      0.62
NORTH  JONES      04/15/95 13:52:41 TOY TOWN             10.25      0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN            121.76      7.31
SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL      101.38      6.09
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING       500.00     30.00
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE          137.00      8.22
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE          135.75      8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY           9.98      0.60

*** GRAND TOTAL (14 ITEMS)                          1,383.66     83.05
```

Remarks:

- the SORT statement causes the report to be sorted on REGION, EMPL–NAME and SALES–DATE

Figure 12. Using a SORT statement to specify the sort order of a report

Summary

Here is a summary of what we learned in this lesson:

- use the SORT statement to **sort your report**
- you can sort on **multiple sort fields**
- you can sort in either **ascending or descending** order

The next lesson will show you how to create control breaks and print subtotals and other statistics in your reports.

To Learn More

There are some additional features associated with the SORT statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional features include:

- creating a **control break** with the SORT statement (page 177)
- specifying **control break spacing** with the SORT statement (page 178)
- requesting **totals and statistics** in the SORT statement (page 186)

The complete syntax for the SORT statement is given in Chapter 10, "Control Statement Syntax" (page 595).

Lesson 7. How to Create Control Breaks

This lesson teaches you what control breaks are, and shows how to request them in your report. This lesson also shows how to print totals and other statistics in reports. The control statement discussed is:

- the BREAK statement

How to Use the BREAK Statement

If you are not a programmer, the term "control break" may be new to you. But it is a very simple concept. And as you will see, control breaks can make your reports much more useful.

Consider the result of sorting a report on some field. By sorting the report on a field, we *group together* all the report lines that contain a particular value for that field. For example, in the report in **Figure 12** (page 63) we sorted first of all on the REGION field. As you can see, this caused the report lines to be grouped together by region. All of the report lines for the East region appear together at the beginning of the report. Next come all of the report lines for the North region, and so on. By sorting on the REGION field, we grouped together all of the records for each region.

Often it is desirable to perform special processing whenever one such group of records ends and another group is about to begin. For example, you might want to print a line of totals for the group that just ended. Or, you might want to print a few blank lines before the next group starts printing, or even skip to a new page. This processing is called **control break processing**. A **control break** is said to occur whenever one group of records ends and another group is about to begin. The field that is being grouped (for example, REGION) is called the **control break field** (or often just the **break field**). A control break field *must* also be a sort field, since it is by being sorted that records are grouped together in the first place.

You may designate any sort field as a control break field. Just name the field in a BREAK statement:

```
SORT:  REGION  EMPL-NAME  SALES-DATE
BREAK: REGION
```

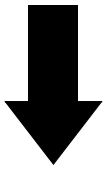
The above statement makes REGION a control break field. Now we will get REGION totals in the report whenever one region finishes printing and another region is about to begin.

After these totals, two blank lines will print. Then the report lines for the next region start to print, and so on.

Figure 13 shows a report that uses the above BREAK statement to produce a control break.

These Control Statements:

```
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION
TITLE:   'RECENT SALES'
TITLE:   'TOTALLED BY REGION'
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT TAX
```



Produce this Report:

```
                              RECENT SALES
                           TOTALLED BY REGION

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/95 15:30:22 STAR MARKET          44.35      2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY       29.65      1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI        14.99      0.90
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER         23.87      1.43
*** TOTAL FOR EAST  (4 ITEMS)                          112.86      6.77


NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45     14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES        9.98      0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY           10.25      0.62
NORTH  JONES      04/15/95 13:52:41 TOY TOWN             10.25      0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN            121.76      7.31
*** TOTAL FOR NORTH (5 ITEMS)                          386.69     23.22


SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL      101.38      6.09
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING       500.00     30.00
*** TOTAL FOR SOUTH (2 ITEMS)                          601.38     36.09


WEST   BAKER      03/26/95 12:09:09 JACKS CAFE          137.00      8.22
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE          135.75      8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY           9.98      0.60
*** TOTAL FOR WEST  (3 ITEMS)                          282.73     16.97



****** GRAND TOTAL (14 ITEMS)                        1,383.66     83.05
```

Remarks:

- REGION is a sort field in this report
- the BREAK statement makes REGION a control break field
- whenever the value of the REGION column changes, a control break occurs
- at each control break a total line prints, followed by two blank lines

Figure 13. Using the BREAK statement to create a control break

How to Specify Control Break Spacing

You can use additional parms in the BREAK statement to customize your control break. For example, you can specify a **break spacing parm**. This parm tells Spectrum Writer what kind of spacing to perform at the control break. By default, Spectrum Writer prints two blank lines at each control break (after the totals line). You can use a spacing parm to request either a different number of blank lines, or to request a page break.

For example, the following statement makes REGION a break field and specifies that 3 blank lines should print at the control break:

```
BREAK:  REGION  SPACE(3)
```

If you want to skip to a new page whenever the contents of the REGION field changes, use the PAGE spacing parm, like this:

```
BREAK:  REGION  SPACE(PAGE)
```

The SPACE(PAGE) parm specifies that, rather than printing 2 blank lines whenever the REGION field changes, the report should skip to a new page.

The report in **Figure 14** illustrates the use of the PAGE spacing parm to request a page break.

How to Print Statistics at a Control Break

You may want to print statistics other than totals at a control break. The total line, as we have seen, prints automatically at control breaks. By supplying the appropriate parm in the BREAK statement, you can also print up to five additional statistical lines at a control break. These additional lines are:

- an **average** line
- a **non–zero average** line (the average of all non–zero values)
- a **maximum** line
- a **minimum** line
- a **non–zero minimum** line (the minimum non–zero value)

The parms that correspond to these statistical lines are:

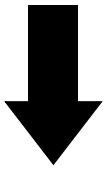
- AVERAGE (or AVG)
- NZAVERAGE (or NZAVG)
- MAXIMUM (or MAX)
- MINIMUM (or MIN)
- NZMINIMUM (or NZMIN)

You can specify as many of these parms as you like in the BREAK statement. The parms may be specified in any order. (The statistic lines in the report, however, will always print in a standard fixed order.) For example:

```
BREAK:  REGION  AVERAGE  MAXIMUM
```

These Control Statements:

```
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION  SPACE(PAGE)
TITLE:   'SALES FOR REGION:'  REGION   /   'PAGE'  #PAGENUM
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
SALES FOR REGION: EAST                                              PAGE    1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT        TAX

EAST   MORRISON   03/29/95 15:30:22 STAR MARKET          44.35        2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY       29.65        1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI        14.99        0.90
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER         23.87        1.43
*** TOTAL FOR EAST  (4 ITEMS)                           112.86        6.77
```

```
SALES FOR REGION: NORTH                                             PAGE    2

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT        TAX

NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45       14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES        9.98        0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY           10.25        0.62
NORTH  JONES      04/15/95 13:52:41 TOY TOWN             10.25        0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN            121.76        7.31
*** TOTAL FOR NORTH (5 ITEMS)                           386.69       23.22
```

```
SALES FOR REGION: SOUTH                                             PAGE    3

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT        TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL      101.38        6.09
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING       500.00       30.00

                        (other report lines not shown)
```

Remarks:

- the SPACE(PAGE) parm causes the report to skip to a new page whenever the REGION field changes value
- since each page contains data for only a single region, we chose to include the REGION field in the title
- a single slash (/) in the TITLE statement divides the title into two parts (left- and right-aligned)

Figure 14. A BREAK statement that produces a page break

The BREAK statement above requests that an average line and a maximum line print (in addition to the totals line) whenever the contents of the REGION field changes.

Figure 15 (page 70) shows a sample report that uses the preceding BREAK statement.

How to Produce Multiple Control Breaks

You may designate more than one sort field as a control break field. Spectrum Writer even allows *all* of your sort fields to be control break fields. However, most reports look best when no more than the first two or three sort fields are used as control breaks. The following example makes the first two sort fields control break fields:

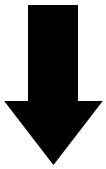
```
SORT:  REGION  EMPL-NAME  SALES-DATE
BREAK: REGION    SPACE(3)
BREAK: EMPL-NAME SPACE(1)
```

In the statements above, we made both REGION and EMPL–NAME control break fields. A control break will occur whenever the REGION field changes values (as in the previous examples). A total line will print for the region, and then 3 blank lines will print. But in this example, the second sort field, EMPL–NAME, is also designated as a control break field. So, a control break will also occur whenever the EMPL–NAME field changes value. A total line will print for the employee, followed by 1 blank line. **Figure 16** shows a sample report that uses the above statements.

Note: When multiple BREAK statements are used, they may appear in any order. However, all BREAK statements must appear after the SORT statement.

These Control Statements:

```
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION  AVERAGE  MAXIMUM
TITLE:   'RECENT SALES'
TITLE:   'TOTALLED BY REGION'
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                              RECENT SALES
                           TOTALLED BY REGION

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/95 15:30:22 STAR MARKET          44.35      2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY       29.65      1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI        14.99      0.90
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER         23.87      1.43
*** TOTAL FOR EAST  (4 ITEMS)                          112.86      6.77
*** AVERAGE VALUE                                       28.22      1.69
*** MAXIMUM VALUE                                       44.35      2.66


NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45     14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES        9.98      0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY           10.25      0.62
NORTH  JONES      04/15/95 13:52:41 TOY TOWN             10.25      0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN            121.76      7.31
*** TOTAL FOR NORTH (5 ITEMS)                          386.69     23.22
*** AVERAGE VALUE                                       77.34      4.64
*** MAXIMUM VALUE                                      234.45     14.07

                    (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                        1,383.66     83.05
****** AVERAGE VALUE                                    98.83      5.93
****** MAXIMUM VALUE                                   500.00     30.00
```

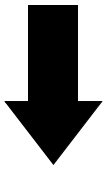
Remarks:

- the AVERAGE and MAXIMUM parms (in the BREAK statement) cause two statistical lines to print (in addition to the totals line) whenever the REGION field changes value
- at the Grand Total, the same statistical lines also print

Figure 15. A report that prints statistical information at control breaks and the Grand Totals

These Control Statements:

```
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION  SPACE(3)
BREAK:   EMPL-NAME  SPACE(1)
TITLE:   'SALES TOTALLED BY EMPLOYEE AND REGION'
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                    SALES TOTALLED BY EMPLOYEE AND REGION

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT       TAX

EAST   MORRISON   03/29/95 15:30:22 STAR MARKET          44.35        2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY       29.65        1.78
*** TOTAL FOR MORRISON   (2 ITEMS)                       74.00        4.44

EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI        14.99        0.90
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER         23.87        1.43
*** TOTAL FOR SIMPSON    (2 ITEMS)                       38.86        2.33

****** TOTAL FOR EAST  (4 ITEMS)                        112.86        6.77



NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL         234.45       14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES        9.98        0.60
*** TOTAL FOR JOHNSON    (2 ITEMS)                     244.43       14.67

NORTH  JONES      04/15/95 07:58:32 EZ GROCERY          10.25        0.62
NORTH  JONES      04/15/95 13:52:41 TOY TOWN            10.25        0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN           121.76        7.31
*** TOTAL FOR JONES      (3 ITEMS)                     142.26        8.55

****** TOTAL FOR NORTH (5 ITEMS)                        386.69       23.22

                         (other report lines not shown)


********* GRAND TOTAL (14 ITEMS)                     1,383.66       83.05
```

Remarks:

- the two BREAK statements make both REGION and EMP–NAME control break fields
- when the EMPL–NAME field changes, employee totals print, followed by 1 blank line
- when the REGION field changes, region totals print, followed by 3 blank lines
- the employee total line begins with 3 asterisks, while the region total line begins with 6 asterisks, and the Grand Total line has 9 asterisks (indicating the level of the break)

Figure 16. A report with two levels of control breaks

Summary

Here is a summary of what we learned in this lesson:

- use the BREAK statement to specify a **control break field**
- control break fields must also be **sort fields**
- use the SPACE parm to specify your own **spacing** at the control break
- use one or more statistical parms to request that **statistical lines** print at a control break
- you can specify **multiple control breaks** in the same report

The next lesson will show you how to turn reports with control breaks into "summary reports."

To Learn More

There are some additional features associated with the BREAK statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional topics include:

- additional control break spacing parms, including one that skips to a **new sheet of paper** (page 178)
- how to print one or more **customized headers at the beginning** of a control break (page 200)
- how to print one or more **customized lines at the end** of a control break (page 188)
- how to **customize the total line** and the other statistical lines (page 182 and page 186)
- how to **suppress the total line** at a control break (page 185)
- how to print only the total lines to produce a **summary report** (page 73 and page 209)
- how to compute **percentages and ratios** that apply to an entire control group (page 202)
- how to customize the **Grand Totals** at the end of the report (page 207)

The complete syntax for the BREAK statement is given Chapter 10, "Control Statement Syntax" (page 481).

Lesson 8. How to Create Summary Reports

This lesson teaches you how to produce summary reports. The control statement discussed is:

- the OPTIONS statement

How to Create a Summary Report

A summary report is one which does not show the detail information for every record included in the report. Instead the detail information is summarized and only the totals are printed in the report.

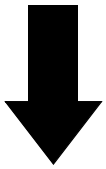
Control breaks are used to create the desired total lines. Consider the report shown earlier on page 66. It is a detail report that lists each sale made in every region. The control break on REGION causes a total line to print after the detail lines for each region have printed. By adding the following statement, we can suppress the detail lines and print just the region totals:

```
OPTIONS: SUMMARY
```

Figure 17 shows a summary report that uses the above statement.

These Control Statements:

```
OPTIONS: SUMMARY
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION
TITLE:   'REGIONAL SALES SUMMARY'
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                          REGIONAL SALES SUMMARY

           EMPL     SALES    SALES
REGION     NAME      DATE     TIME     CUSTOMER        AMOUNT        TAX

*** TOTAL FOR EAST  (4 ITEMS)                            112.86       6.77
*** TOTAL FOR NORTH (5 ITEMS)                            386.69      23.22
*** TOTAL FOR SOUTH (2 ITEMS)                            601.38      36.09
*** TOTAL FOR WEST  (3 ITEMS)                            282.73      16.97

****** GRAND TOTAL  (14 ITEMS)                         1,383.66      83.05
```

Remarks:

- this is the same report as on page 66, except for the additional OPTIONS statement
- the SUMMARY parm (in the OPTIONS statement) suppresses the detail report lines, leaving just a summary report
- in summary reports, only the numeric columns are filled in (with total values)

Figure 17. Producing a summary report

Summary

Here is a summary of what we learned in this lesson:

- use the SUMMARY option (in the OPTIONS statement) to **create a summary report**
- a summary report **must have at least one control break field**

The next lesson will show you how to use data from more than one input file in a report.

To Learn More

There are some additional features associated with summary reports which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Examples of additional features include:

- **customizing** the summary lines in your report (page 182)
- **printing statistics** (such as averages, maximums and minimums) in your summary report (page 186)
- creating **multiple levels** of summarization (page 204)
- printing a **limited number of detail records** in each control group, creating reports such as "The Top 3 Sales in Each Region" (page 212)

Lesson 9. How to Use Data from More Than One File

This lesson teaches you how to read records from additional input files for use in your report. The control statement discussed is:

- the READ statement

All of the sample reports produced so far have used data from only one input file. The data has come from the file specified in the INPUT statement, called the **primary input file**. There are times when all of the data needed for a particular report will not be found in just a single file. One of Spectrum Writer's most powerful features is its ability to link to *any number* of additional files to produce a report.

How Auxiliary Input Files Are Processed

Each report is allowed to have only one primary input file, specified in the INPUT statement. When data from additional input files is required to produce a report, a READ statement is used. The READ statement causes a record to be read from another input file, called an **auxiliary input file**. You may have as many READ statements as you like in a single report.

Here is how Spectrum Writer processes the primary and auxiliary input files. Spectrum Writer first reads a single record from the primary input file. (This file is always read *sequentially*, beginning with the first record in the file.) Next, if any auxiliary input files were specified, Spectrum Writer also reads one record from each of those files. (These files are always read *randomly*, using a key.) At this point, Spectrum Writer will have read one record from each of the input files. The fields from *all* of these records are now available for use in producing the report. These fields can be used:

- as columns in the body of the report
- in titles
- as sort fields
- as control break fields
- in conditional expressions
- in calculations
- and in any other way that fields from the primary input file are used

After processing this set of records, Spectrum Writer then repeats the process. Another record is read sequentially from the primary input file. Then random reads are performed to each of the auxiliary input files. This next group of records is then used in making the report, and so on. This process is repeated until there are no more records left in the primary input file.

By simply adding a READ statement to your report request, you automatically make all of the data fields from another file available for use in producing your report.

There is one important thing about auxiliary input files to keep in mind. Since these files are ready randomly, they *must* be keyed files (or DB2 tables). (VSAM files are often keyed files.)

In a keyed file, each record has a unique "key" value associated with it. When a random read is made to such a file, a **read key** must be specified to identify which record to read. What read key should you tell Spectrum Writer use when reading a record from an auxiliary input file? In order to be useful, the auxiliary input record should be somehow related to the primary input record. Usually, the record from the primary input file will contain the key of a corresponding record in the auxiliary input file. That key from the primary input file will be used as the read key.

> **Note:** If you are not familiar with such terms as "keyed files" and "read keys," ask your programmer to help you determine whether a particular file is keyed or not, and also to help you decide what read key to use.

How to Use the READ Statement

Now let's look at a concrete example of how to use the READ statement. Begin by considering **Figure 18**, which shows a simple report that uses only a primary input file (the SALES–FILE). This report shows information about each sale made by an employee.

This report includes columns for two fields that we haven't used in previous examples, so we'll explain them. They are the EMPL–NUM field and the PRODUCT–CODE field. The EMPL–NUM is the employee number of the employee who made the sale. The PRODUCT–CODE is a code that identifies which product was sold to the customer.

Now, let's assume that we need this same report to also show each employee's social security number. The social security number is not available in the SALES–FILE. But it *is* a field in the EMPL–FILE. (See the report on page 38.) In order to produce such a report, we need data from a second input file— the EMPL–FILE.

The EMPL–FILE is a keyed VSAM file. Its key is the 3–byte employee number. The records in the SALES–FILE also contain an employee number, so we can use that field as the "read key" when we read the EMPL–FILE. So, we can make the EMPL–FILE an auxiliary input file by simply adding this statement:

```
READ:  EMPL-FILE   READKEY(EMPL-NUM)
```

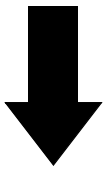
This READ statement tells Spectrum Writer to use the EMPL–NUM field from the records in the SALES–FILE as a key to read an auxiliary record from the EMPL–FILE. All control statements after this READ statement may now refer to the fields in the EMPL–FILE, as well as to those in the SALES–FILE. So, we can now add the SOCIAL–SEC–NUM field from the EMPL–FILE to our COLUMNS statement:

```
READ:     EMPL-FILE  READKEY(EMPL-NUM)
COLUMNS:  EMPL-NAME  SALES-FILE.EMPL-NUM  SOCIAL-SEC-NUM
          SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```

Notice that in the above COLUMNS statement we must now prefix the EMPL–NUM field with a record name (SALES–FILE.EMPL–NUM). This is because after the READ statement, EMPL–NUM is no longer a unique field name. A field by that name exists in both the SALES–FILE and the EMPL–FILE. (See Appendix F, "Files Used in Examples" (page 648).) Since the EMPL–NUM will have the same value in both of the records, it doesn't really matter which one we specify in the COLUMNS statement, but we do have to specify a unique name. In this case we specified the EMPL–NUM field from the SALES–FILE. (For more information on using "record names" to qualify field names, see "How to Name the Input File Records" on page 228.)

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'RECENT SALES'
COLUMNS: EMPL-NAME  EMPL-NUM  SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```



Produce this Report:

```
                        RECENT SALES

   EMPL    EMPL  SALES                                    PRODUCT
   NAME    NUM   DATE     CUSTOMER        AMOUNT          CODE

JOHNSON    037  03/12/95 ACE ELECTRICAL          101.38   952
BAKER      044  03/26/95 JACKS CAFE              137.00   978
MORRISON   042  03/29/95 STAR MARKET              44.35   907
MORRISON   042  03/30/95 A1 PHOTOGRAPHY           29.65   919
SIMPSON    041  04/01/95 EUROPEAN DELI            14.99   916
JOHNSON    039  04/01/95 VILLA HOTEL             234.45   926
JOHNSON    039  04/05/95 MARYS ANTIQUES            9.98   997
BAKER      044  04/12/95 JACKS CAFE              135.75   916
THOMAS     045  04/14/95 YOGURT CITY               9.98   997
JONES      036  04/15/95 EZ GROCERY               10.25   977
JONES      036  04/15/95 TOY TOWN                121.76   907
JONES      036  04/15/95 TOY TOWN                 10.25   977
JOHNSON    037  04/16/95 ACME BUILDING           500.00   976
SIMPSON    041  04/30/95 J & S LUMBER             23.87   916


*** GRAND TOTAL (14 ITEMS)                    1,383.66
```

Remarks:

- all fields used in this report come from the SALES–FILE

Figure 18. A report that uses only the primary input file

Figure 19 shows a report which uses the above statements. The report now has the desired new column showing each employee's social security number. Notice that we also sorted the report on SOCIAL–SEC–NUM. Remember that you can use fields from auxiliary input files in any way that you can use fields from the primary input file.

"One-to-Many" Random Reads

Normally, Spectrum Writer reads just a single record from your auxiliary input file — the record whose key field matches the READKEY value. If you want to use *all of the records* which match the READKEY (or partial READKEY), add the MULTI parm to your READ statement. When the READ statement has the MULTI parm, Spectrum Writer creates and processes "logical input records" by matching the primary input file row with *each* qualifying record from the auxiliary input file. For more information on how the MULTI parm works, see "How to Perform "One–to–Many" Reads" on page 232.

How to Use Multiple READ Statements

You may use as many READ statements in a run as you like. The report in **Figure 20** uses two READ statements. The primary input file is once again the SALES–FILE, which contains one record for each sale made by an employee.

To obtain additional data about the employee who made each sale, we use a READ statement for the EMPL–FILE (just like in the preceding example). The EMPL–NUM field in the SALES–FILE contains the key necessary to read the correct EMPL–FILE record.

To obtain additional information about each *product* sold, a second READ statement names the PRODUCT–FILE as another auxiliary input file. (The PRODUCT–FILE is also described in Appendix F, "Files Used in Examples" on page 648.)

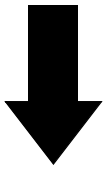
However, there is one minor complication in reading records from this file. The key in the PRODUCT–FILE records is 4 bytes long. It consists of the letter "P" followed by a 3–byte product code. The SALES–FILE does not contain a field which can be used *directly* as the read key to the PRODUCT–FILE. But it does contain the 3–byte PRODUCT–CODE field, which we can use to build the 4–byte read key. A COMPUTE statement is therefore used to create a new field (called PRODKEY) which consists of the letter "P" followed by the product code. This computed field is then used as the read key in the READ statement for the PRODUCT–FILE:

```
COMPUTE: PRODKEY = 'P' + PRODUCT-CODE
READ:    PRODUCT-FILE   READKEY(PRODKEY)
```

By having two READ statements in addition to the INPUT statement, the report now uses data from three input files. Data from all of these files can be used in any of the subsequent control statements. In the report in **Figure 20**, the COLUMNS statement uses two fields from the auxiliary input files. It uses the SOCIAL–SEC–NUM field from the EMPL–FILE and the PRODUCT–DESC field from the PRODUCT–FILE.

These Control Statements:

```
INPUT:   SALES-FILE
READ:    EMPL-FILE  READKEY(EMPL-NUM)
SORT:    SOCIAL-SEC-NUM
TITLE:   'SALES SORTED BY SOCIAL SECURITY NUMBER'
COLUMNS: EMPL-NAME  SALES-FILE.EMPL-NUM  SOCIAL-SEC-NUM
         SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```



Produce this Report:

SALES SORTED BY SOCIAL SECURITY NUMBER

EMPL NAME	SALES FILE EMPL NUM	SOCIAL SEC NUM	SALES DATE	CUSTOMER	AMOUNT	PRODUCT CODE
JOHNSON	039	004-77-9981	04/05/95	MARYS ANTIQUES	9.98	997
JOHNSON	039	004-77-9981	04/01/95	VILLA HOTEL	234.45	926
JONES	036	012-09-8765	04/15/95	EZ GROCERY	10.25	977
JONES	036	012-09-8765	04/15/95	TOY TOWN	10.25	977
JONES	036	012-09-8765	04/15/95	TOY TOWN	121.76	907
SIMPSON	041	112-05-0456	04/30/95	J & S LUMBER	23.87	916
SIMPSON	041	112-05-0456	04/01/95	EUROPEAN DELI	14.99	916
THOMAS	045	776-83-8221	04/14/95	YOGURT CITY	9.98	997
BAKER	044	878-19-0156	04/12/95	JACKS CAFE	135.75	916
BAKER	044	878-19-0156	03/26/95	JACKS CAFE	137.00	978
MORRISON	042	900-12-0556	03/30/95	A1 PHOTOGRAPHY	29.65	919
MORRISON	042	900-12-0556	03/29/95	STAR MARKET	44.35	907
JOHNSON	037	912-04-0334	03/12/95	ACE ELECTRICAL	101.38	952
JOHNSON	037	912-04-0334	04/16/95	ACME BUILDING	500.00	976

*** GRAND TOTAL (14 ITEMS) 1,383.66

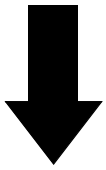
Remarks:

- the READ statement makes the fields from the EMPL–FILE available for use
- the COLUMNS statement includes the SOCIAL–SEC–NUM field from the EMPL–FILE
- we also sorted the report on the SOCIAL–SEC–NUM field from the EMPL–FILE
- the EMPL–NUM field must be prefixed with a record name in the COLUMNS statement, since a field by that name exists in both input files

Figure 19. A report that uses a READ statement to specify an auxiliary input file

These Control Statements:

```
INPUT:    SALES-FILE
READ:     EMPL-FILE      READKEY(EMPL-NUM)
COMPUTE:  PRODKEY = 'P' + PRODUCT-CODE
READ:     PRODUCT-FILE   READKEY(PRODKEY)
TITLE:    'SALES SORTED BY SOCIAL SECURITY NUMBER'
COLUMNS:  EMPL-NAME
          SALES-FILE.EMPL-NUM
          SOCIAL-SEC-NUM
          SALES-DATE
          CUSTOMER
          PRODUCT-CODE
          PRODUCT-DESC
```



Produce this Report:

SALES SORTED BY SOCIAL SECURITY NUMBER

EMPL NAME	SALES FILE EMPL NUM	SOCIAL SEC NUM	SALES DATE	CUSTOMER	AMOUNT	PRODUCT CODE	PRODUCT DESC
JOHNSON	039	004-77-9981	04/05/95	MARYS ANTIQUES	9.98	997	MAILING LABELS
JOHNSON	039	004-77-9981	04/01/95	VILLA HOTEL	234.45	926	DESK CALENDARS
JONES	036	012-09-8765	04/15/95	EZ GROCERY	10.25	977	PAPER CLIPS
JONES	036	012-09-8765	04/15/95	TOY TOWN	10.25	977	PAPER CLIPS
JONES	036	012-09-8765	04/15/95	TOY TOWN	121.76	907	INKPADS
SIMPSON	041	112-05-0456	04/30/95	J & S LUMBER	23.87	916	RED PENS
SIMPSON	041	112-05-0456	04/01/95	EUROPEAN DELI	14.99	916	RED PENS
THOMAS	045	776-83-8221	04/14/95	YOGURT CITY	9.98	997	MAILING LABELS
BAKER	044	878-19-0156	04/12/95	JACKS CAFE	135.75	916	RED PENS
BAKER	044	878-19-0156	03/26/95	JACKS CAFE	137.00	978	HOLE PUNCHERS
MORRISON	042	900-12-0556	03/30/95	A1 PHOTOGRAPHY	29.65	919	GREEN PENS
MORRISON	042	900-12-0556	03/29/95	STAR MARKET	44.35	907	INKPADS
JOHNSON	037	912-04-0334	03/12/95	ACE ELECTRICAL	101.38	952	PENCILS (NO. 1)
JOHNSON	037	912-04-0334	04/16/95	ACME BUILDING	500.00	976	CHAIRS

*** GRAND TOTAL (14 ITEMS) 1,383.66

Remarks:

- all fields from the SALES–FILE, the EMPL–FILE and the PRODUCT–FILE are available for use in the report
- the key to the PRODUCT–FILE is a computed field
- the EMPL–NUM field must be prefixed with a record name in the COLUMNS statement, since a field by that name exists in two input files (SALES–FILE and EMPL–FILE)
- the SOCIAL–SEC–NUM field comes from the EMPL–FILE auxiliary input file
- the PRODUCT–DESC field comes from the PRODUCT–FILE auxiliary input file

Figure 20. A report that uses two READ statements to specify two auxiliary input files

Summary

Here is a summary of what we learned in this lesson:

- the READ statement is used to read records from **auxiliary input files**
- the file named in a READ statement must be a **keyed file** (or a DB2 table)
- you may have as **many READ statements as you like** in a single report

To Learn More

There are some additional features associated with the READ statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional features include:

- how to assign a **record name** to the records read from auxiliary input files (page 228)
- how to read multiple records (containing **different keys**) from the same auxiliary input file (page 224)
- how to use **data from one auxiliary input file as the read key** to another auxiliary input file (page 226)
- how to specify **generic keys** and **"KGE" keys** in the READ statement (page 230)
- how to read multiple records (with the **same key or partial key**) from the auxiliary input file (page 232)
- what happens when **no record is found** for a particular read key (page 229)
- how to determine whether the read for a particular key was **successful** or not (page 230)
- how to use the READ statement to obtain data from a **DB2 table or view** (page 393)
- how to use the ONIOERROR option to **increase the severity** of I/O errors on an input file (page 586)

The complete **syntax** for the READ statement, as well as a more detailed **narrative** of how Spectrum Writer assembles input records during the report process, is given in Chapter 10, "Control Statement Syntax" (page 578).

Chapter 3. How to Request a PC File

Chapter Table of Contents

Chapter 3. How to Request a PC File 83

Lesson 1. How to Produce a PC File in 5 Minutes 88
Converting a Whole Mainframe File 88
Another 5–Minute Example 90
Using Your Company's Files 90

Lesson 2. How to Include Only Certain Records In Your PC File 93
How to Use the INCLUDEIF Statement 93
How to Write Conditional Expressions 95

Lesson 3. How to Create Your Own Fields 98
Creating Numeric Fields 98
Creating Character Fields 100
Assigning Values to Fields Based on Conditions 102

Lesson 4. How to Specify the PC File Order 105
How to Use the SORT Statement 105
Automatic Sorting 105

Lesson 5. How to Create Control Breaks 108
How to Use the BREAK Statement 108
Customizing the Control Break 108

Lesson 6. How to Create Summary Files 113
How to Create a Summary File 113

Lesson 7. How to Use Data from More Than One File 116
How Auxiliary Input Files Are Processed 116
How to Use the READ Statement 117
"One-to-Many" Random Reads 120
How to Use Multiple READ Statements 120

Chapter 3. How to Request a PC File

This chapter teaches you how to turn mainframe data into PC files to use in your favorite PC program. Spectrum Writer makes using mainframe data in PC programs as easy as 1-2-3.

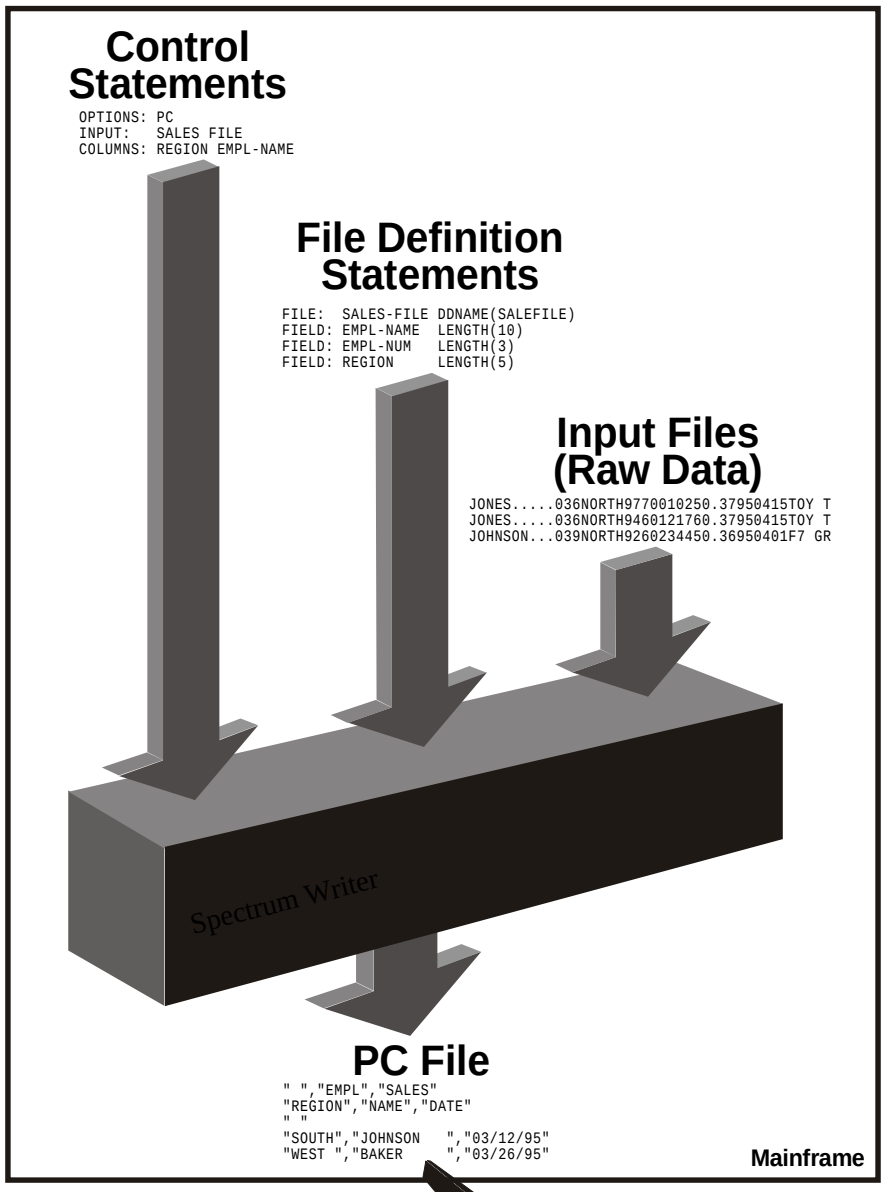
1. **Use Spectrum Writer to create a custom PC file on your mainframe.**
   Spectrum Writer's language is non–procedural, which means you just describe the *result* you want, not the programming steps needed to do it. Describe your PC file with a few simple "control statements". (These control statements are the same ones you already learned about in the previous chapter.) You can create a PC file with just three control statements. The lessons in this chapter teach you how to use the control statements.

   Once you've written the necessary control statements, submit a batch job to execute Spectrum Writer. Spectrum Writer examines the control statements describing the PC file you want. It automatically locates the appropriate "file definition" statements stored in a copy library. (These statements define your mainframe files.) Spectrum Writer then accesses the mainframe data and creates the desired PC file on your mainframe.

2. **Download the PC file to your PC.**
   Just use your shop's existing download facility to transfer the PC file to your PC.

3. **Use the PC file in your PC program.**
   Start the PC program and "open" or "import" the PC file with a few simple keystrokes. When you use the PC file in a spreadsheet program, for example, the program automatically moves the data into the correct rows and columns. Each downloaded record results in one *row* in a spreadsheet. And each field becomes a *column* in a spreadsheet.

Using mainframe data in your favorite PC program is as easy as that with Spectrum Writer!



Step 1. Create PC file on mainframe

Step 2. Download PC file to PC

Step 3. Use PC file in PC program

Figure 21 lists all of the Spectrum Writer control statements available when requesting PC files and describes the function of each one.

The remainder of this chapter is divided into seven easy lessons that explain how to use the various control statements to request PC files.

After reading just the first lesson, you will be able to produce useful PC files with Spectrum Writer. The other lessons introduce additional control statements, and explain their roles in producing increasingly sophisticated PC files. It is not necessary to read all of the other lessons initially. Nor is it necessary to read the lessons in sequential order. Read the summaries below and decide which lessons you need for the kind of PC files you want to produce.

Lesson 1. How to Produce a PC File in 5 Minutes.
This lesson shows how to produce PC files using just three simple control statements — the INPUT, COLUMNS and OPTIONS statements. You will use these three statements for every PC file you make.

Lesson 2. How to Include Only Certain Records In Your PC File.
This lesson shows how to use the INCLUDEIF statement to specify which mainframe records to include in your PC file.

Lesson 3. How to Create Your Own Fields.
This lesson shows you how to create your own fields by performing computations on existing fields. This is done with the COMPUTE statement.

Lesson 4. How to Specify the PC File Order.
This lesson shows how to sort your PC file into whatever order you want. The use of the SORT statement is explained.

Lesson 5. How to Create Control Breaks.
This lesson shows how to break a PC file up into sections, with subtotals appearing at the end of each section. The use of the BREAK statement to request such "control breaks" is explained.

Lesson 6. How to Create Summary Files.
This lesson shows you how to turn a PC file with subtotals into a small summary file that is more easily downloaded to a PC.

Lesson 7. How to Use Data from More Than One File.
This lesson shows how easy it is to read records from additional files when producing PC files. By adding a single READ statement, you automatically have access to all of the fields from an additional file.

These lessons show the most common use of each control statement. Most control statements also have additional features that are not discussed in these lessons. Additional ways to use these control statements are discussed in Chapter 4, "Beyond the Basics." The complete syntax for each control statement is shown in Chapter 10, "Control Statement Syntax."

SPECTRUM WRITER CONTROL STATEMENTS (GROUPED BY FUNCTION)

Statements that Define How Input Data Looks

FILE	Defines a file
FIELD	Defines a field within a file
ASM	Defines a file using an Assembler record layout
COBOL	Defines a file using a Cobol record layout
COMPUTE	Computes a new user-defined field

Statements that Specify the Input Files to Use to Make the PC File

INPUT	Specifies the primary input file
READ	Specifies an auxiliary input file

Statements that Describe the Body of a PC File

INCLUDEIF	Specifies which input records to include in the PC file
COLUMNS	Specifies the PC file's columns and column headings

Statements that Define the PC File Order and Control Breaks

SORT	Specifies the PC file order and, optionally, specifies control break fields
BREAK	Specifies control break processing

Miscellaneous Statements

OPTIONS	Specifies the kind of PC file needed, as well as various other special options
NEWOUT	Indicates that subsequent statements will define a new PC file
COPY	Copies additional control statements for processing

Figure 21. Spectrum Writer Control Statements for making PC Files

Lesson 1. How to Produce a PC File in 5 Minutes

This lesson teaches you how to turn your mainframe data into PC files using just three simple control statements. These statements are:

- the OPTIONS statement
- the INPUT statement
- the COLUMNS statement

Converting a Whole Mainframe File

With Spectrum Writer, it only takes three statements to convert an entire mainframe file into a comma-delimited PC file, ready to import into your favorite PC program:

```
OPTION:  PC
INPUT:   SALES-FILE
COLUMNS: SALES-FILE
```

Figure 22 shows a portion of the PC file created with the above statements. It also shows the spreadsheet obtained by importing the PC file into Excel.

Let's examine what each statement does. The **OPTION statement** above tells Spectrum Writer that you want to produce a comma-delimited PC file (instead of a report) in this run. Such PC files can be imported into virtually any PC spreadsheet, data base or word processing program.

The **INPUT statement** identifies the mainframe file that contains the data that you want to put into your PC file. In this case, we specified SALES–FILE. This is a sample "sales file" that is used in many of the examples in this manual. This file contains information about each sale made by the employees of an imaginary company.

The **COLUMNS statement** specifies what columns of data we want in our PC spreadsheet. Here you can name the individual fields from the input file that you want to use to populate the columns of the spreadsheet. However, in this example we named the input file itself. That means that we want a column in the spreadsheet for *every* field in the input file.

With just these three statements, we've given Spectrum Writer everything it needs to turn mainframe data into a PC file! That's all there is to creating custom PC files with Spectrum Writer. Three simple statements let you accomplish what would otherwise have taken an entire COBOL program to do!

> **Note:** The JCL used to create this PC file is shown on page 416 for OS/390 (page 430 for VSE).

These Control Statements:

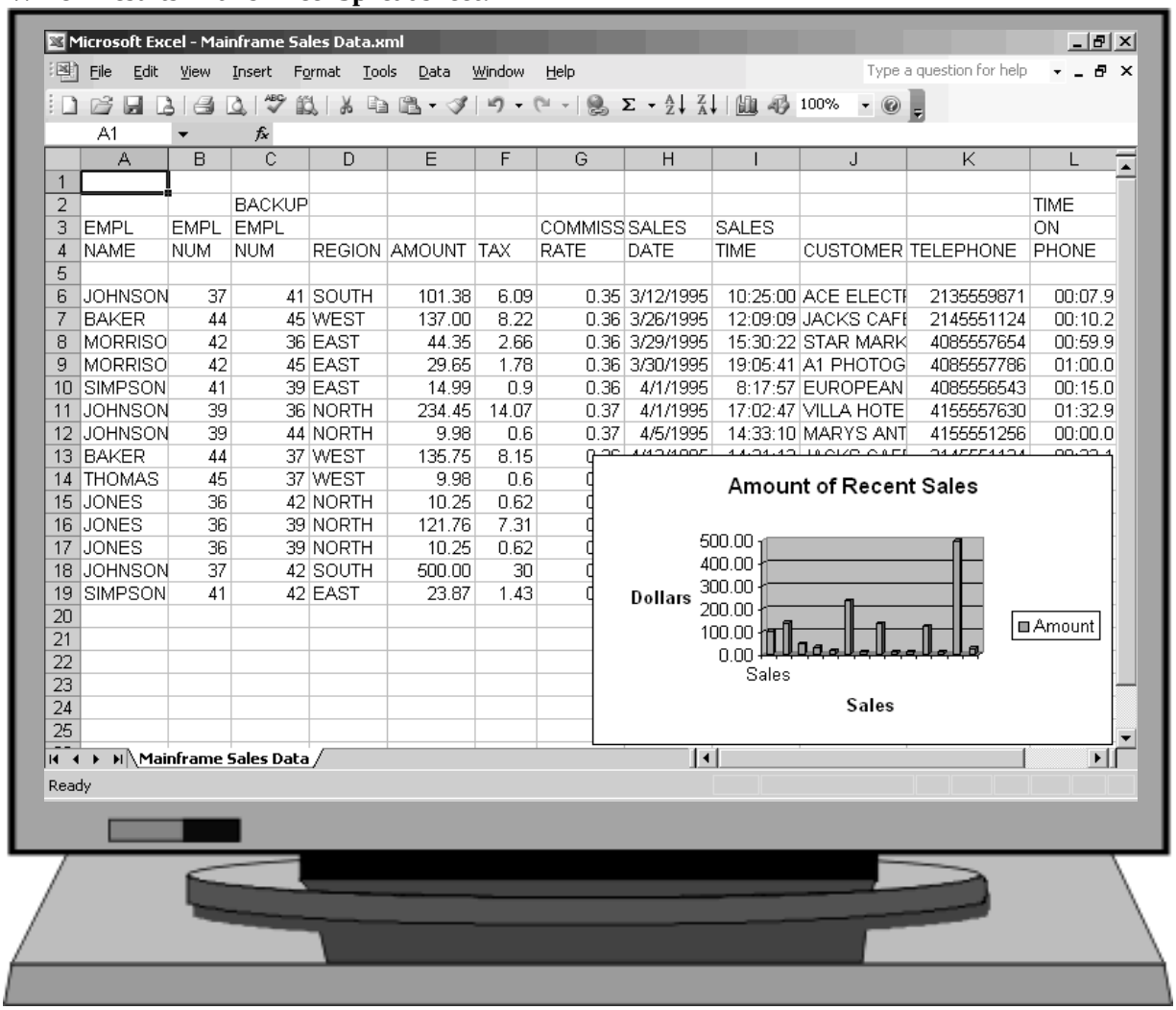
```
OPTION:  PC
INPUT:   SALES-FILE
COLUMNS: SALES-FILE
```

Produced this PC File:

```
" "," ","BACKUP"," "," "," "," "," "," "," "," "," ","TIME"
"EMPL","EMPL","EMPL"," "," "," ","COMMISSION","SALES","SALES"," "," ","ON"
"NAME","NUM","NUM","REGION","AMOUNT","TAX","RATE","DATE","TIME","CUSTOMER","TELEPHONE","PHONE"
" "," "," "," "," "," "," "," "," "," "," "," "," "," "," "," "
"JOHNSON   ","037","041","SOUTH",      101.38,      6.09,     0.350,"03/12/95","10:25:00","ACE ELECTRICAL ", 2135559871,"00:00:07.9"
"BAKER     ","044","045","WEST ",      137.00,      8.22,     0.360,"03/26/95","12:09:09","JACKS CAFE     ", 2145551124,"00:00:10.2"
"MORRISON  ","042","036","EAST ",       44.35,      2.66,     0.360,"03/29/95","15:30:22","STAR MARKET    ", 4085557654,"00:00:59.9"
"MORRISON  ","042","045","EAST ",       29.65,      1.78,     0.360,"03/30/95","19:05:41","A1 PHOTOGRAPHY ", 4085557786,"00:01:00.0"
```

(additional lines not shown)

Which Results in this Excel Spreadsheet:



	A	B	C	D	E	F	G	H	I	J	K	L
1												
2			BACKUP									TIME
3	EMPL	EMPL	EMPL				COMMISS	SALES	SALES			ON
4	NAME	NUM	NUM	REGION	AMOUNT	TAX	RATE	DATE	TIME	CUSTOMER	TELEPHONE	PHONE
5												
6	JOHNSON	37	41	SOUTH	101.38	6.09	0.35	3/12/1995	10:25:00	ACE ELECTI	2135559871	00:07.9
7	BAKER	44	45	WEST	137.00	8.22	0.36	3/26/1995	12:09:09	JACKS CAFI	2145551124	00:10.2
8	MORRISO	42	36	EAST	44.35	2.66	0.36	3/29/1995	15:30:22	STAR MARK	4085557654	00:59.9
9	MORRISO	42	45	EAST	29.65	1.78	0.36	3/30/1995	19:05:41	A1 PHOTOG	4085557786	01:00.0
10	SIMPSON	41	39	EAST	14.99	0.9	0.36	4/1/1995	8:17:57	EUROPEAN	4085556543	00:15.0
11	JOHNSON	39	36	NORTH	234.45	14.07	0.37	4/1/1995	17:02:47	VILLA HOTE	4155557630	01:32.9
12	JOHNSON	39	44	NORTH	9.98	0.6	0.37	4/5/1995	14:33:10	MARYS ANT	4155551256	00:00.0
13	BAKER	44	37	WEST	135.75	8.15						
14	THOMAS	45	37	WEST	9.98	0.6						
15	JONES	36	42	NORTH	10.25	0.62						
16	JONES	36	39	NORTH	121.76	7.31						
17	JONES	36	39	NORTH	10.25	0.62						
18	JOHNSON	37	42	SOUTH	500.00	30						
19	SIMPSON	41	42	EAST	23.87	1.43						
20												
21												
22												
23												
24												
25												

Figure 22. An Excel spreadsheet containing the entire mainframe SALES-FILE took only 3 statements

Another 5–Minute Example

Now let's make another PC file, this time using a different input file. This time we will create a Quattro Pro spreadsheet using data from the EMPL–FILE. EMPL–FILE is a sample employee file. We will create a simple employee directory from that file. In this example, we don’t need *every* field from the EMPL-FILE. We just want the spreadsheet to have columns showing: employee number, last name, first name, sex, social security number, date hired, and their city and state. We only need the following three statements:

```
OPTIONS: PC
INPUT:   EMPL-FILE
COLUMNS: EMPL-NUM  LAST-NAME  FIRST-NAME  SEX  SOCIAL-SEC-NUM
         HIRE-DATE  CITY  STATE
```

The OPTIONS statement above again specifies that we want to create a PC file rather than a report. The INPUT statement above specifies that the data we need for the PC file will come from the employee file (EMPL–FILE). The COLUMNS statement specifies the columns of data we want. Notice that we needed two lines for the COLUMNS statement in this example. You can continue a control statement onto as many lines as you need to. Just leave at least one blank space at the beginning of each continuation line.

The Quattro Pro spreadsheet resulting from the above statements is shown in **Figure 23**.

You have now seen two examples showing just how easy it is to create PC files with Spectrum Writer. That's all there is to it! You now know enough to request basic PC files from the files at your company.

Using Your Company's Files

You may be wondering how Spectrum Writer knows the names of your company's files and fields. The answer is that your company's files are defined to Spectrum Writer by other control statements that are kept in a Spectrum Writer "copy library." For example, the statements used to define the sample files used in the preceding examples are shown in Appendix F, "Files Used in Examples" (page 648).

For a list of the file names and field names available for you to use, ask your programmer. They can print that information from the Spectrum Writer Copy Library, in a format similar to that shown in Appendix F.

If you already know the name of the *file* to use, you can use a "dummy" run to easily get a list of all of its fields. Just use an INPUT statement with the SHOWFLDS(YES) parm, like this:

```
INPUT: SALES-FILE  SHOWFLDS(YES)
```

The above statement tells Spectrum Writer to print (in the control statement listing) a list of all of the fields defined for SALES–FILE.

If a file that you need to use has not yet been defined, see Chapter 6, "How to Define Your Input Files" to learn how to do that.

These Control Statements:

```
OPTIONS: PC
INPUT:   EMPL-FILE
COLUMNS: EMPL-NUM  LAST-NAME  FIRST-NAME  SEX  SOCIAL-SEC-NUM
         HIRE-DATE  CITY  STATE
```

Result in this Quattro Pro Spreadsheet:

	A	B	C	D	E	F	G	H
1					SOCIAL			
2	EMPL	LAST	FIRST		SEC	HIRE		
3	NUM	NAME	NAME	SEX	NUM	DATE	CITY	STATE
4								
5	036	JONES	JERRY	M	12098765	01/31/80	SAN FRANCISCO	CA
6	037	JOHNSON	THOMAS	M	912040334	06/21/75	SCOTTSDALE	AZ
7	039	JOHNSON	LINDA	F	4779981	11/25/79	SANTA ROSA	CA
8	040	MACDONA	RICHARD	M	889790013	07/04/82	PLEASANTON	CA
9	041	SIMPSON	TIMOTHY	M	112050456	12/01/82	ARCADIA	CA
10	042	MORRISO	MICHAEL	M	900120556	11/30/79	GLENDALE	CA
11	043	CHRISTOP	MELISSA	F	415090761	08/15/81	PHOENIX	AZ
12	044	BAKER	VIVIAN	F	878190156	06/04/82	WALNUT CREEK	CA
13	045	THOMAS	MARTIN	M	776838221	06/04/82	CONCORD	CA
14								
15								
16								
17								
18								
19								
20								
21								
22								

Figure 23. A Quattro Pro employee directory produced with just three control statements

Summary

Here is a summary of what we learned in this lesson:

- the OPTIONS statement lets you request that a PC file be produced, instead of a report
- the INPUT statement tells Spectrum Writer which mainframe input file contains the data needed in your PC file
- the COLUMNS statement tells Spectrum Writer what "columns" of data to put in your PC file
- by using just these three statements you can produce a PC file

The next lesson will teach you how to limit the records that are included in your PC file.

To Learn More

To learn more about writing control statements in general, see Chapter 9, "General Syntax Rules." In that chapter you will learn such things as:

- **how long** each line can be (page 443)
- how to **continue** control statements onto multiple lines (page 444)

There are some additional features associated with the OPTIONS, INPUT and COLUMNS statements which we have not covered in this lesson. Some of these additional features are discussed in Chapter 4, "Beyond the Basics." Examples of additional features are:

- how to specify your own **column headings** for a PC file (page 130)
- how to **suppress column headings** in your PC file (page 130)
- how to reserve **more room for numeric columns** in your PC file (page 135)
- how to create a column that contains a **literal text** (page 126)
- how to produce **multiple rows in the PC file** for each input record (page 151)
- how to turn data from **DB2 tables and views** into PC files (page 397)
- how to turn data from **existing mainframe reports** into PC files (page 258)
- how to customize your PC file, by specifying such things as: the **delimiter character** to use, the **"text qualifier"** character to use (for example, a quotation mark), the date format to use, whether to enclose dates in quotation marks, etc. (page 275)

The complete syntax for the OPTIONS, INPUT and COLUMNS statements appears in Chapter 10, "Control Statement Syntax" (pages 555, 542 and 498 respectively).

Lesson 2. How to Include Only Certain Records In Your PC File

This lesson teaches you how to select only certain records from the input file for inclusion in your PC file. The control statement discussed is:

- the INCLUDEIF statement

How to Use the INCLUDEIF Statement

In the previous lesson we saw how to select certain *fields* to be downloaded. (We used the COLUMNS statement to identify the fields that we wanted.) Now let's look at how to download only selected *records* from the mainframe file. We will use the INCLUDEIF ("include if") statement.

When no INCLUDEIF statement is specified, Spectrum Writer includes every record from the mainframe file. Use the INCLUDEIF statement to tell Spectrum Writer to "include" a record in the PC file only "if" one or more conditions are met.

This feature is very useful when you are working with large mainframe files. Downloading the entire file might take a long time (and use up lots of hard disk). Using the INCLUDEIF statement lets you download only the records that you actually need.

For example, assume that we want to download data from the SALES–FILE to a spreadsheet similar to the one in **Figure 22** (page 89). But this time let's just download the data for the employee named Jones. We simply add the following INCLUDEIF statement to our other control statements:

```
INCLUDEIF:  EMPL-NAME = 'JONES'
```

The above INCLUDEIF statement tells Spectrum Writer to "include" records from the SALES–FILE "if" the EMPL–NAME field is equal to 'JONES'. Spectrum Writer still reads through the entire SALES–FILE, just like before. But now it *examines* each record before including it in the PC file. If the record's EMPL–NAME field contains the value 'JONES', then the record is included in the PC file. If the EMPL–NAME field contains any other value, then that record is not included in the PC file.

Figure 24 shows an Access table produced using the above statement. Only the sales made by Jones appear in that table.

> **Note:** Notice that we specified an additional option in the OPTIONS statement in this example. The **NOCOLHDG option** tells Spectrum Writer that we do not want column headings in the PC file. Column headings are often desirable in file imported into *spreadsheets*. But this PC file is being imported into an Access database. The records should all contain real data. During the import process, Access let us manually specify a "field name" to use for each column.

The INCLUDEIF statement may appear anywhere after the INPUT statement. Only one INCLUDEIF statement is allowed per run, but it may contain as many conditions as you like.

These Control Statements:

```
OPTIONS:   PC NOCOLHDG
INPUT:     SALES-FILE
INCLUDEIF: EMPL-NAME = 'JONES'
COLUMNS:   REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Result in this Access Table:

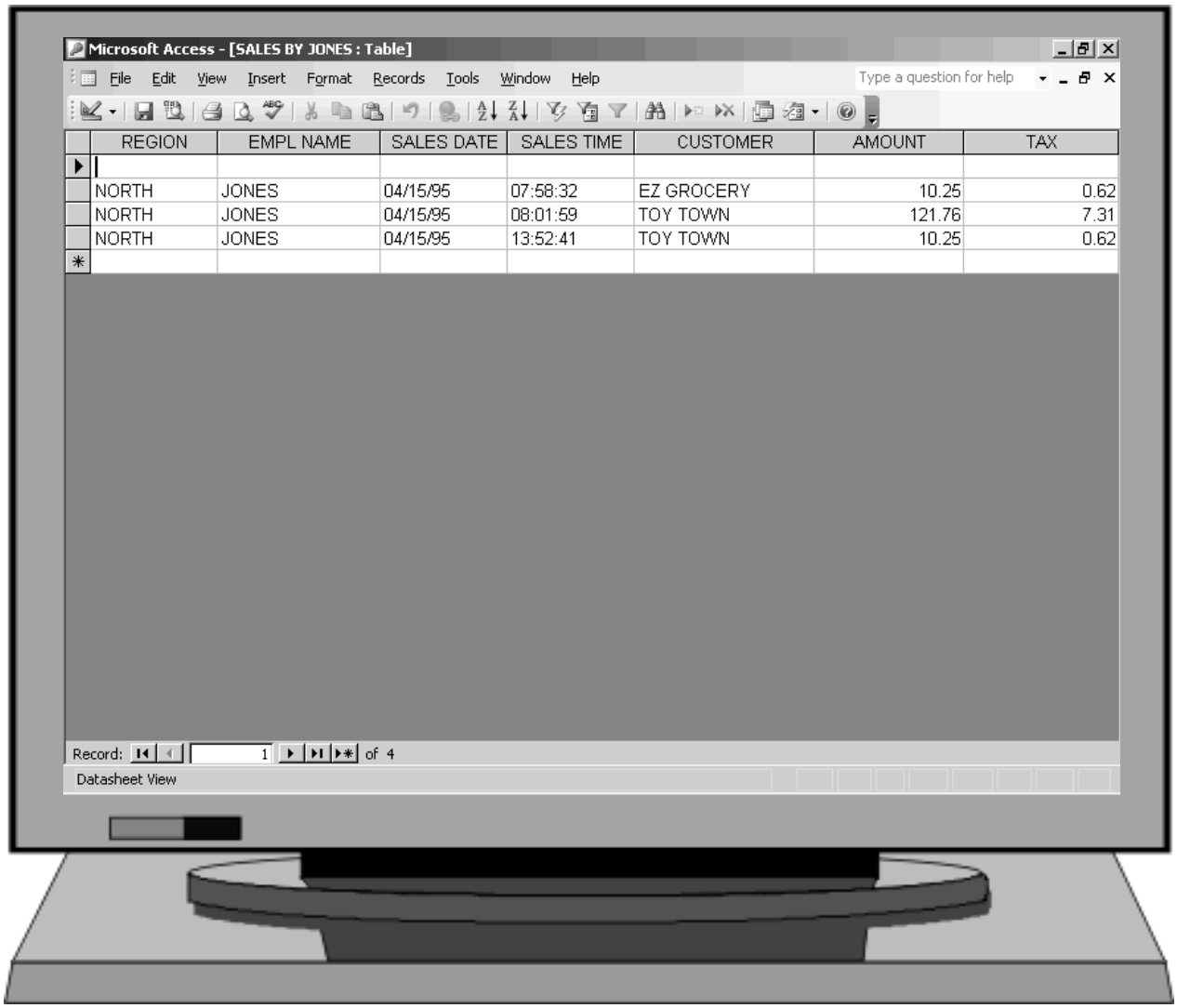


Figure 24. Using an INCLUDEIF statement to specify which records to include in a PC file

By the way, the INCLUDEIF statement can refer to any of the fields in the input file (as well as any COMPUTE field). You are not limited to just the fields that are listed in the COLUMNS statement.

How to Write Conditional Expressions

The INCLUDEIF statement simply contains a **conditional expression**. The complete rules for writing conditional expressions are explained in "Conditional Expressions" (page 459). Briefly, a conditional expression contains one or more "conditions," separated with the words AND or OR. A **condition** usually involves comparing the contents of one field with the contents of another field, or with a literal value. Let's look at some more examples of INCLUDEIF statements and their conditional expressions.

> **Note:** If you are a programmer, you will notice that the syntax for conditional expressions is very similar to the syntax used in "IF statements" in COBOL, PL/1, and BASIC. If you are familiar with any of these languages, you should find it especially easy to write INCLUDEIF statements.

You may want your PC file to include all records which **do not contain** a certain value. Do this by specifying "not equal" in your condition. For example:

```
INCLUDEIF:  EMPL-NAME ¬= 'JONES'
```

The above statement specifies that the PC file should include all records from the input file whose EMPL–NAME field is not equal to 'JONES'.

> **Note:** In addition to ¬=, you can also use <> to indicate "not equal", like this:
>
> ```
> INCLUDEIF: EMPL-NAME <> 'JONES'
> ```

You may want to include a record in your PC file if **either of two conditions** is true. To do this, use an INCLUDEIF statement with two conditions, separated by the word OR. Consider the following statement:

```
INCLUDEIF:  EMPL-NAME = 'JONES'  OR  AMOUNT > 100
```

The above statement states that a record should be included in the PC file "if the EMPL–NAME field is equal to 'JONES' or if the AMOUNT field is greater than 100." The word OR indicates that records from the input file will be included if either one (or both) of the conditions are true.

Notice in the above statement that we enclosed 'JONES' in single quotation marks, while we did not use quotation marks around the 100. That is because EMPL–NAME is a character field, while AMOUNT is a numeric field. Character literals (such as 'JONES') must be enclosed in quotation marks. You can use either single (') or double (") quotation marks. But numeric literal (such as 100), as well as date and time literals, are *not* enclosed in quotation marks. Numeric literal also must not contain commas. (The rules for writing literals are thoroughly explained in "How to Write Literals" on page 448.)

As another example, you may want to include records in your PC file when **both of two conditions** are true. For example, let's say we want a listing only of sales that were made by Jones and that were also for an amount over $100. For this PC file, two conditions must

both be true: the EMPL–NAME field must be equal to 'JONES' and the AMOUNT field must be over 100. Use the word AND to specify that both conditions must be true, like this:

```
INCLUDEIF:  EMPL-NAME = 'JONES'  AND  AMOUNT > 100
```

Now as Spectrum Writer reads each record from the input file, it will include a record in the PC file only "if the EMPL–NAME field is equal to 'JONES' and the AMOUNT field is greater than 100."

Here is an example of including records in a PC file based on the contents of a **date field**:

```
INCLUDEIF:  SALES-DATE  >  4/15/1995
```

The above statement specifies that records should be included in the PC file only if their SALES–DATE field contains a date greater than (after) April 15, 1995.

> **Note:** You may be wondering if you need to use a different format for your date literals when you know that a particular date field is stored in a record as a Julian date (YYDDD format.) The answer is no. All date literals in your control statements should be written as MM/DD/YYYY (or MM/DD/YY). Spectrum Writer automatically takes care of any date conversions that may be required. Thus, you test Julian date fields just like all other date fields:
>
> ```
> INCLUDEIF: JULIAN-START-DATE >= 1/1/2000
> ```

Here is an example of including records in a PC file based on the contents of a **time field**:

```
INCLUDEIF:  SALES-TIME  <  17:00:00
```

The above statement specifies that records should be included in the PC file only if their SALES–TIME field contains a time less than (before) 17:00:00 (which is 5 PM).

> **Note:** You are allowed to omit the seconds in your time literals, if you prefer. When the seconds are not specified, zero seconds is assumed. Thus, you could also write the above statement this way:
>
> ```
> INCLUDEIF:  SALES-TIME  <  17:00
> ```

If your INCLUDEIF statement contains both the words OR and AND, you should use parentheses to indicate the order in which to perform the comparisons. Consider the following statement:

```
INCLUDEIF: EMPL-NAME = 'JONES'  OR
           (SALES-DATE > 4/15/1995  AND  SALES-DATE < 4/30/1995)
```

In the above statement, records will be included if the EMPL–NAME field is equal to 'JONES', *or* if both of the SALES–DATE comparisons are true. The parentheses cause the two SALES–DATE comparisons to be treated as one condition. That condition is true if the SALES–DATE is greater than April 15, 1995 and is less than April 30, 1995.

> **Note:** In addition to the actual words AND and OR, you can also use the symbols "&" and "|", respectively, in your conditional expressions.

Summary

Here is a summary of what we learned in this lesson:

- use the INCLUDEIF statement when you want to include only certain records from the input file in your PC file
- the INCLUDEIF statement contains one or more conditions, separated by the words AND or OR
- groups of conditions can be enclosed in parentheses, to indicate the order in which the comparisons should be performed

The next lesson will show you how to compute your own new fields to download to your PC.

To Learn More

There are some additional features associated with the INCLUDEIF statement which we have not covered in this lesson. These additional features are discussed in "Conditional Expressions" (page 459). The additional features include:

- how to use the keyword **NOT** (or the symbol ¬) to negate a condition (page 469)
- how to **scan** a character field, to see if a certain text exists anywhere within the field (page 460)
- how to specify conditions based on **bit fields** (page 465)
- how to specify a condition based on a field's raw **hexadecimal** value (page 464)
- what to do if you prefer to specify **date literals** in DD/MM/YY or DD/MM/YYYY format (page 140), like this:

```
INCLUDEIF: SALES-DATE > 15/4/1995
```

- how the **KEYRANGE** and **STOPWHEN** parms of the INPUT statement can be used to limit the records included in your run (page 542)

The complete syntax for the INCLUDEIF statement appears in Chapter 10, "Control Statement Syntax" (page 540).

Lesson 3. How to Create Your Own Fields

This lesson teaches you how to create your own fields to include in your PC file. The control statement discussed is:

- the COMPUTE statement

Sometimes the data you need to download to your PC program is not contained in the input file. Yet the necessary data might be easily computed from one or more fields which *are* in the input file. In such cases, simply create a new field by using the COMPUTE statement.

Creating Numeric Fields

A COMPUTE statement specifies the name of the new field to create and supplies a *computational expression* to use in assigning a value to that field. The complete rules for computational expressions are discussed in "Computational Expressions" (page 472). Generally, your expression will consist of one or more arithmetic operations performed on numeric fields and/or numeric literals.

For example, the sample SALES–FILE has numeric fields named AMOUNT and TAX. We can use the COMPUTE statement to create a new field containing the total amount due just by adding those two fields together, like this:

```
COMPUTE:  TOTAL-AMOUNT = AMOUNT + TAX
```

The above statement creates a new field named TOTAL–AMOUNT. It is computed by adding the AMOUNT field and the TAX field together. Now that the TOTAL–AMOUNT field has been created, we can use that field in *any way* that other fields can be used. For example, a computed field can be used: as a column of data in your PC file; as a sort field; as a control break field; as part of a conditional expression (in the INCLUDEIF statement); even as an operand in subsequent COMPUTE statements to create other fields.

The Access table in **Figure 25** was obtained by using the above COMPUTE statement.

COMPUTE statements normally appear somewhere after the INPUT statement. The COMPUTE statement must appear before any other control statement that refers to the field being created. In **Figure 25**, the COMPUTE statement for TOTAL–AMOUNT had to come before the COLUMNS statement, since the COLUMNS statement referred to that field.

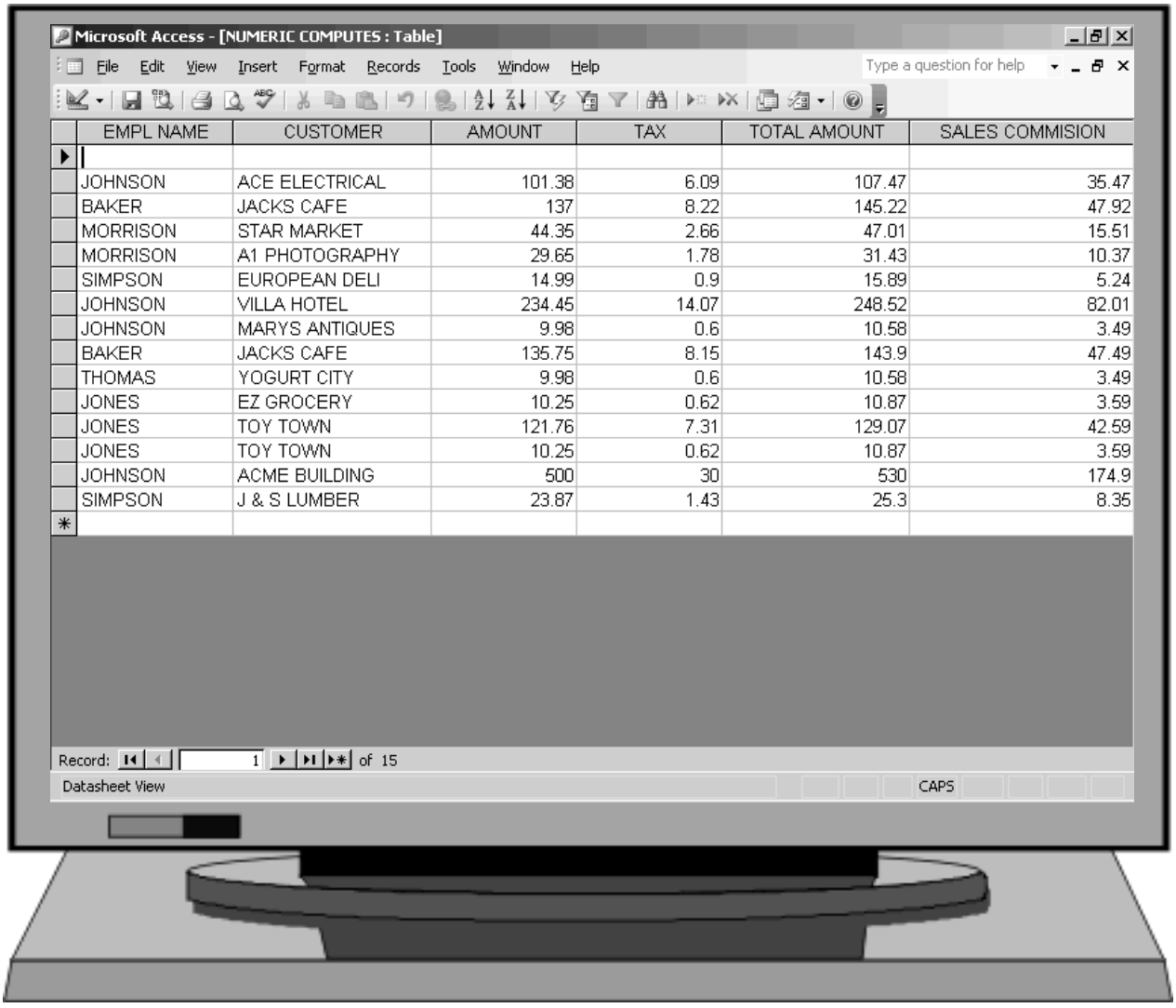
You can perform addition, subtraction, multiplication, and division in the COMPUTE statement. Use the +, –, * and / symbols, respectively. You may also use parentheses as needed to indicate the order in which the operations should be performed.

> **Note:** When performing subtraction, always put a **blank space before and after the minus sign**. Otherwise, the minus sign will appear to be part of a field name. Blanks are optional around the other arithmetic operators.

These Control Statements:

```
OPTIONS: PC NOCOLHDG
INPUT:   SALES-FILE
COMPUTE: TOTAL-AMOUNT        = AMOUNT + TAX
COMPUTE: SALES-COMMISSION(2) = TOTAL-AMOUNT * .33
COLUMNS: EMPL-NAME  CUSTOMER  AMOUNT  TAX  TOTAL-AMOUNT  SALES-COMMISSION
```

Result in this Microsoft Access table:



EMPL NAME	CUSTOMER	AMOUNT	TAX	TOTAL AMOUNT	SALES COMMISION
JOHNSON	ACE ELECTRICAL	101.38	6.09	107.47	35.47
BAKER	JACKS CAFE	137	8.22	145.22	47.92
MORRISON	STAR MARKET	44.35	2.66	47.01	15.51
MORRISON	A1 PHOTOGRAPHY	29.65	1.78	31.43	10.37
SIMPSON	EUROPEAN DELI	14.99	0.9	15.89	5.24
JOHNSON	VILLA HOTEL	234.45	14.07	248.52	82.01
JOHNSON	MARYS ANTIQUES	9.98	0.6	10.58	3.49
BAKER	JACKS CAFE	135.75	8.15	143.9	47.49
THOMAS	YOGURT CITY	9.98	0.6	10.58	3.49
JONES	EZ GROCERY	10.25	0.62	10.87	3.59
JONES	TOY TOWN	121.76	7.31	129.07	42.59
JONES	TOY TOWN	10.25	0.62	10.87	3.59
JOHNSON	ACME BUILDING	500	30	530	174.9
SIMPSON	J & S LUMBER	23.87	1.43	25.3	8.35

Figure 25. Using the COMPUTE statement to create numeric fields for a PC file

As another example of a creating a numeric field, let's say we wanted to compute a sales commission for each sale. The commission will be 33% of the total value of the sale, including the tax. We could compute the sales commission with the following statement:

```
COMPUTE:  SALES-COMMISSION(2) = TOTAL-AMOUNT * .33
```

This statement creates a new field called SALES–COMMISSION which is computed by multiplying TOTAL–AMOUNT by .33. Notice that we used the result of our previous COMPUTE statement to perform the computation in this statement.

The "(2)" after SALES-COMMISSION tells Spectrum Writer that we want that field to have 2 decimal places. Without that parm, Spectrum Writer would have defaulted to keeping 4 decimal places (in this example).

The Access table in **Figure 25** (page 99) uses the above statement.

In addition to the basic arithmetic operations, there are also a large number of built–in functions that you can use in the COMPUTE statement. These built–in functions allow you to perform more complex mathematical operations on numeric operands. A complete list of built–in functions is found in Appendix D, "Built-In Functions" (page 628).

Creating Character Fields

So far we have been creating numeric fields. Now let's consider how to create your own *character* fields. There is only one operation used in computing character fields. It is the **concatenation** operation. (Don't let that word scare you if it is new to you. "Concatenating" simply means "stringing together" two or more character fields.) The plus sign (+) is used as the symbol for concatenation. For example:

```
COMPUTE: WHOLE-NAME = LAST-NAME + FIRST-NAME
```

The above statement creates a new field named WHOLE–NAME. It is created by concatenating the contents of the LAST–NAME field and the contents of the FIRST–NAME field. The result is a single field which now contains both the last and first names of the employee. The new field will be 30 bytes long — the combined length of the two operands.

You can also concatenate more than two fields together. For example:

```
COMPUTE: MAILING-CODE = STATE + '–' + EMPL-NUM
```

This example creates a new field called MAILING–CODE which consists of the contents of the STATE field, followed by a dash, followed by the contents of the EMPL–NUM field.

In addition to the concatenation operation, there are also a number of built–in functions that can be used when creating character fields. For example, the #LEFT function can be used to extract the leftmost *n* bytes of a character field. Here is an example of how to use the #LEFT built–in function:

```
COMPUTE: FIRST-INITIAL = #LEFT(FIRST-NAME,1)
```

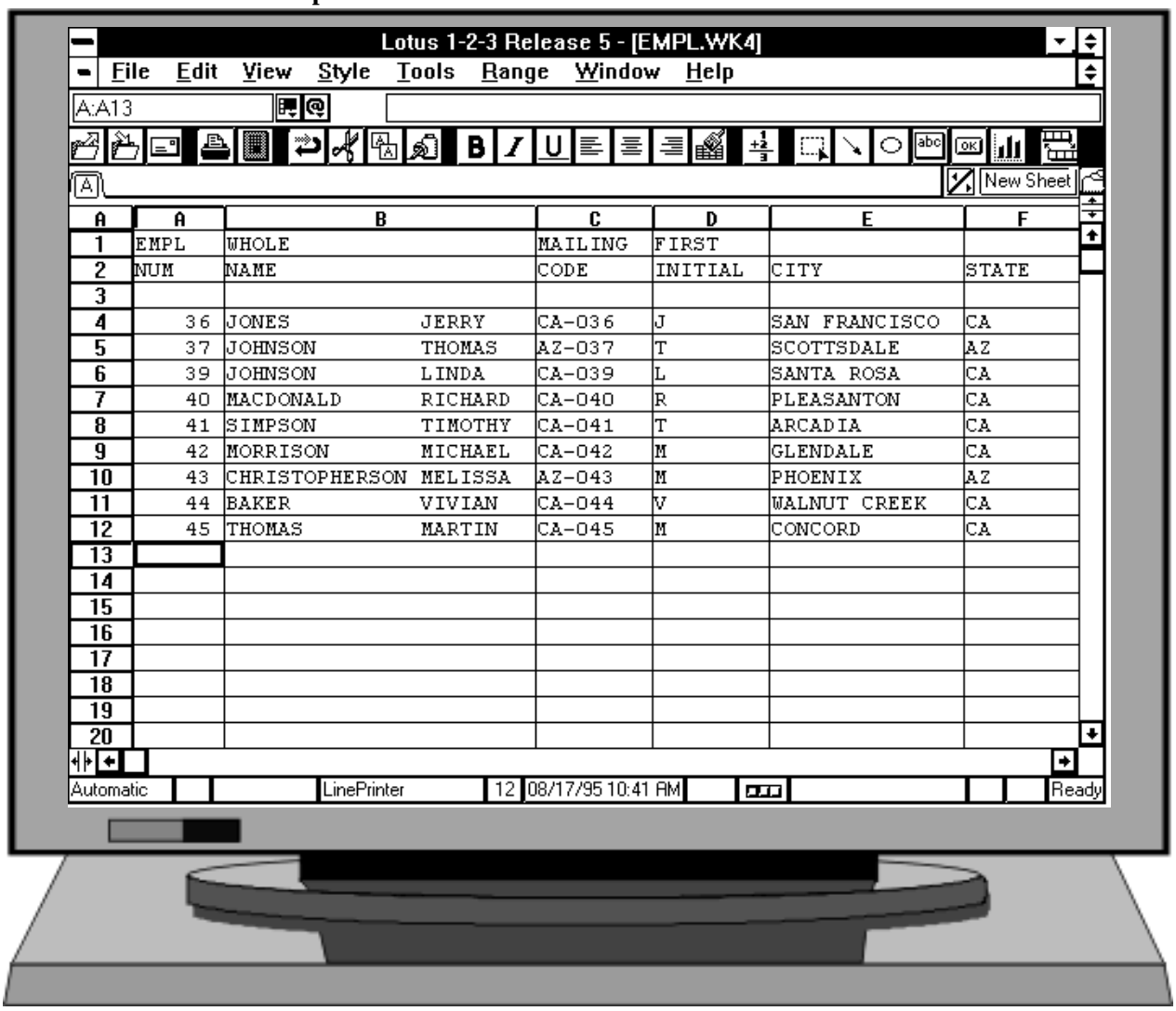
This statement creates a new character field which consists of only the first character (that is, the leftmost 1 byte) of the FIRST–NAME field.

Figure 26 shows a spreadsheet that uses each of the above COMPUTE statements.

These Control Statements:

```
OPTIONS: PC
INPUT:   EMPL-FILE
COMPUTE: WHOLE-NAME = LAST-NAME + FIRST-NAME
COMPUTE: MAILING-CODE = STATE + '-' + EMPL-NUM
COMPUTE: FIRST-INITIAL = #LEFT(FIRST-NAME,1)
COLUMNS: EMPL-NUM WHOLE-NAME MAILING-CODE FIRST-INITIAL CITY STATE
```

Result in this Lotus 1-2-3 spreadsheet:



	A	B	C	D	E	F
1	EMPL	WHOLE	MAILING	FIRST		
2	NUM	NAME	CODE	INITIAL	CITY	STATE
3						
4	36	JONES JERRY	CA-036	J	SAN FRANCISCO	CA
5	37	JOHNSON THOMAS	AZ-037	T	SCOTTSDALE	AZ
6	39	JOHNSON LINDA	CA-039	L	SANTA ROSA	CA
7	40	MACDONALD RICHARD	CA-040	R	PLEASANTON	CA
8	41	SIMPSON TIMOTHY	CA-041	T	ARCADIA	CA
9	42	MORRISON MICHAEL	CA-042	M	GLENDALE	CA
10	43	CHRISTOPHERSON MELISSA	AZ-043	M	PHOENIX	AZ
11	44	BAKER VIVIAN	CA-044	V	WALNUT CREEK	CA
12	45	THOMAS MARTIN	CA-045	M	CONCORD	CA
13						
14						
15						
16						
17						
18						
19						
20						

Figure 26. Using the COMPUTE statement to create character fields for a PC file

Assigning Values to Fields Based on Conditions

Up until now we have been using "simple" COMPUTE statements. In a **simple COMPUTE statement**, the value of the new field is defined by a single computational expression.

But it is also possible to use conditional logic in a COMPUTE statement. In **conditional COMPUTE statements**, one of several different expressions will be used to assign a value to the new field. The expression that is used will depend on one or more conditions that you specify. Conditional COMPUTE statements can be very powerful tools in producing PC files. Here is an example of a conditional COMPUTE statement:

```
COMPUTE: BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                 WHEN(HIRE-DATE >= 1/1/1980)  ASSIGN(TOTAL-SALES * .05)
```

The above statement creates a field named BONUS. However, in this example the BONUS field can be computed in one of two ways: for employees hired before January 1, 1980, the bonus is 8 percent of total sales (TOTAL–SALES * .08). But, for employees hired on or after January 1, 1980, the bonus is only 5 percent of total sales (TOTAL–SALES * .05).

When assigning a value to the BONUS field, Spectrum Writer evaluates the conditional expression in each WHEN parm. As soon as a WHEN expression is found that is true, the computational expression from the corresponding ASSIGN parm is used to assign a value to BONUS. (Any remaining WHEN parms are not evaluated.)

You may have as many pairs of WHEN and ASSIGN parms as you like in a COMPUTE statement. If none of the WHEN expressions are true, a value of zero will be assigned to the field. To assign some other value when none of the WHEN parms are true, you may use the ELSE parm. For example:

```
COMPUTE: BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                 ELSE                         ASSIGN(TOTAL-SALES * .05)
```

The above statement has the same effect as the previous example, but is a little simpler. It has only one WHEN expression. For employees whose hire date is before January 1, 1980, the bonus will be computed based on 8 percent. For all other cases, the bonus will be computed based on 5 percent.

You may also use conditional COMPUTE statements to create *character* fields. For example:

```
COMPUTE: TITLE = WHEN(SEX = 'M')  ASSIGN('MR')
                 ELSE             ASSIGN('MS')
```

The above statement creates a new field called TITLE. The contents of TITLE will be "MR" if the SEX field contains an "M", and "MS" otherwise.

Figure 27 shows a Lotus spreadsheet obtained by using some of the conditional COMPUTE statements just discussed.

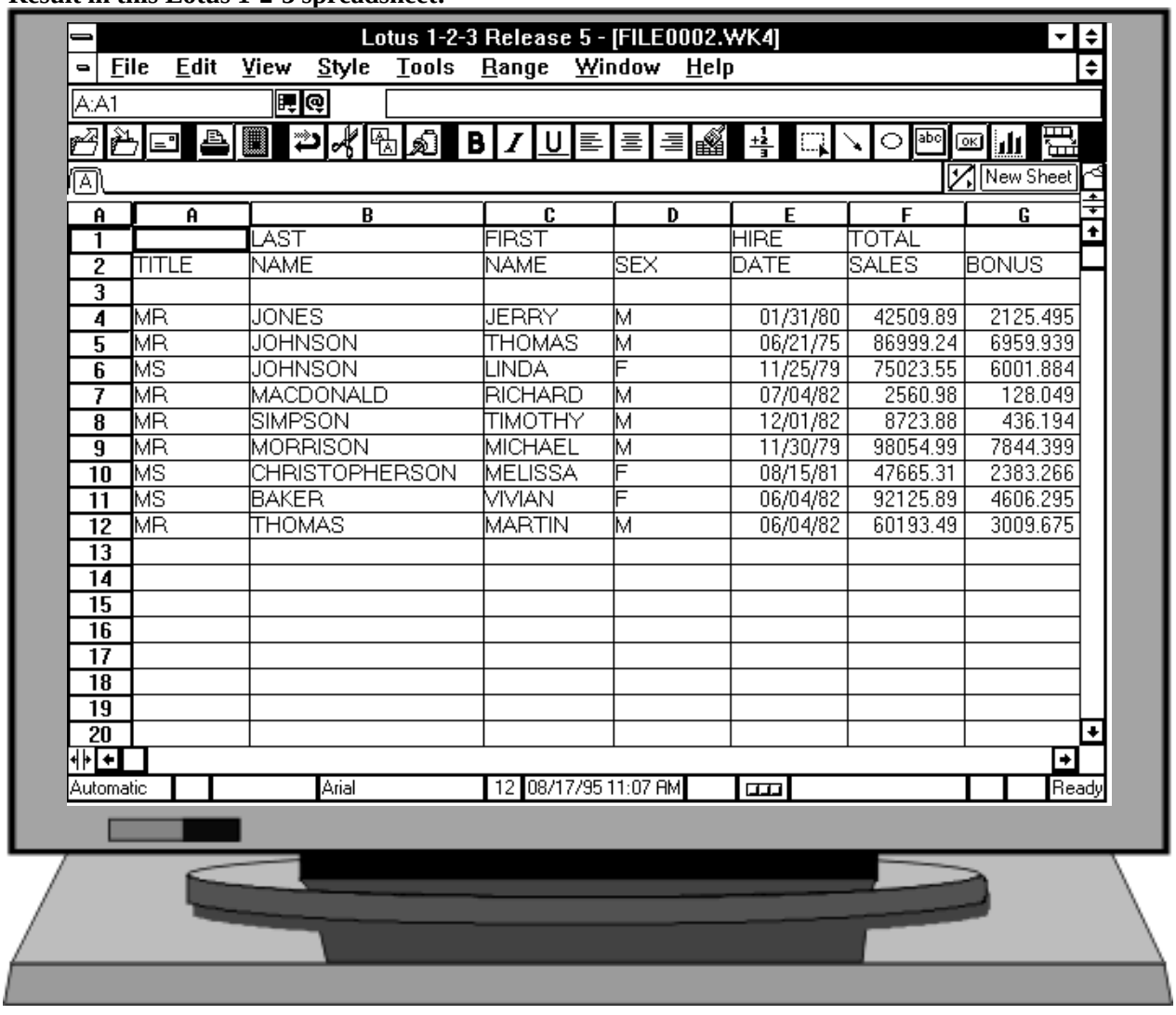
When defining character fields with a conditional COMPUTE statement, a value of **spaces** is assigned if none of the WHEN expressions are true and no ELSE parm is specified.

All of our examples so far have used just a single condition within the WHEN parm. You can, however, use any valid conditional expression within the WHEN parm. The conditional expression can contain as many different conditions as you like, separated with the words AND and OR, and optionally grouped with parentheses. (A conditional expression is the sort of expression that is allowed in the INCLUDEIF statement, as was described in "How to Write

These Control Statements:

```
OPTIONS:  PC
INPUT:    EMPL-FILE
COMPUTE:  BONUS = WHEN(HIRE-DATE <  1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                  WHEN(HIRE-DATE >= 1/1/1980)  ASSIGN(TOTAL-SALES * .05)
COMPUTE:  TITLE = WHEN(SEX = 'M')  ASSIGN('MR')
                  ELSE             ASSIGN('MS')
COLUMNS:  TITLE  LAST-NAME  FIRST-NAME  SEX  HIRE-DATE  TOTAL-SALES  BONUS
```

Result in this Lotus 1-2-3 spreadsheet:



	A	B	C	D	E	F	G
1		LAST	FIRST		HIRE	TOTAL	
2	TITLE	NAME	NAME	SEX	DATE	SALES	BONUS
3							
4	MR	JONES	JERRY	M	01/31/80	42509.89	2125.495
5	MR	JOHNSON	THOMAS	M	06/21/75	86999.24	6959.939
6	MS	JOHNSON	LINDA	F	11/25/79	75023.55	6001.884
7	MR	MACDONALD	RICHARD	M	07/04/82	2560.98	128.049
8	MR	SIMPSON	TIMOTHY	M	12/01/82	8723.88	436.194
9	MR	MORRISON	MICHAEL	M	11/30/79	98054.99	7844.399
10	MS	CHRISTOPHERSON	MELISSA	F	08/15/81	47665.31	2383.266
11	MS	BAKER	VIVIAN	F	06/04/82	92125.89	4606.295
12	MR	THOMAS	MARTIN	M	06/04/82	60193.49	3009.675
13							
14							
15							
16							
17							
18							
19							
20							

Figure 27. Assigning values to computed fields based on conditions

Conditional Expressions" on page 95.) The complete rules for writing conditional expressions are given in "Conditional Expressions" (page 459). Additional examples of COMPUTE statements are shown beginning on page 513.

Summary

Here is a summary of what we learned in this lesson:

- the COMPUTE statement is used to create new fields
- a **simple COMPUTE statement** assigns the result of a single computational expression to a new field
- a **conditional COMPUTE statement** uses one of several different computational expressions, depending on the conditions that you specify

The next lesson will teach you how to sort your PC file into whatever order you want.

To Learn More

There are some additional features associated with the COMPUTE statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Examples of additional topics include:

- how to create **date** fields (page 514)
- how to create **time** fields (page 272)
- how to create **bit** (boolean) fields (page 514)
- how to specify how many **decimal places** a numeric or time field should contain (page 511)
- how to specify **column headings** for the fields you create (page 511)
- how to specify how your field should be **formatted** when it is printed in a report (page 510)
- how to specify whether a numeric or time field should be totalled in the **Grand Totals** line at the end of the report (page 148)
- how to **retain the previous value** of a COMPUTE field in certain cases (page 234)

The complete syntax for the COMPUTE statement appears in Chapter 10, "Control Statement Syntax" (page 506).

Lesson 4. How to Specify the PC File Order

This lesson teaches you how to sort your PC file into any order you want. The control statement discussed is:

- the SORT statement

How to Use the SORT Statement

When no SORT statement is specified, Spectrum Writer defaults to writing out the PC file records in their original input file order. For example, the records in the sample SALES–FILE are stored in sales date order. Therefore, the sales spreadsheets in the previous lessons (for example, page 89) all appeared in sales date order. The EMPL–FILE sample file is a VSAM file stored in EMPL–NUM order. Therefore, the earlier spreadsheets from that file were in employee number order (see page 101 for an example).

To create a PC file in a different order, just add a SORT statement. The SORT statement can appear anywhere after the INPUT statement. Only one SORT statement is allowed per run, but it may contain as many "sort fields" as you like. Spectrum Writer will sort your PC file on all of the sort fields.

For example, let's request a PC file from the SALES–FILE and sort it on three fields:

```
SORT:  REGION  EMPL-NAME  SALES-DATE
```

To begin with, the PC file will be sorted according to the first sort field — REGION. If there are multiple records for the same REGION, then those records will be further sorted using the second sort field, EMPL–NAME. Records having the same value for both the REGION and the EMPL–NAME fields will be further sorted on the third sort field — SALES–DATE.

Figure 28 shows a Microsoft Works spreadsheet obtained by using the above statement.

By the way, the SORT statement can refer to any of the fields in the input file (as well as any COMPUTE field). You are not limited to just the fields that are listed in the COLUMNS statement.

By default, Spectrum Writer sorts PC files into **ascending order** on each sort field. If you want to sort the PC file into **descending order** for a field, put the DESCENDING parm (or just DESC) in parentheses immediately after the field name. For example, to sort a PC file into reverse employee number order, you could use this SORT statement:

```
SORT:  EMPL-NUM(DESC)
```

Automatic Sorting

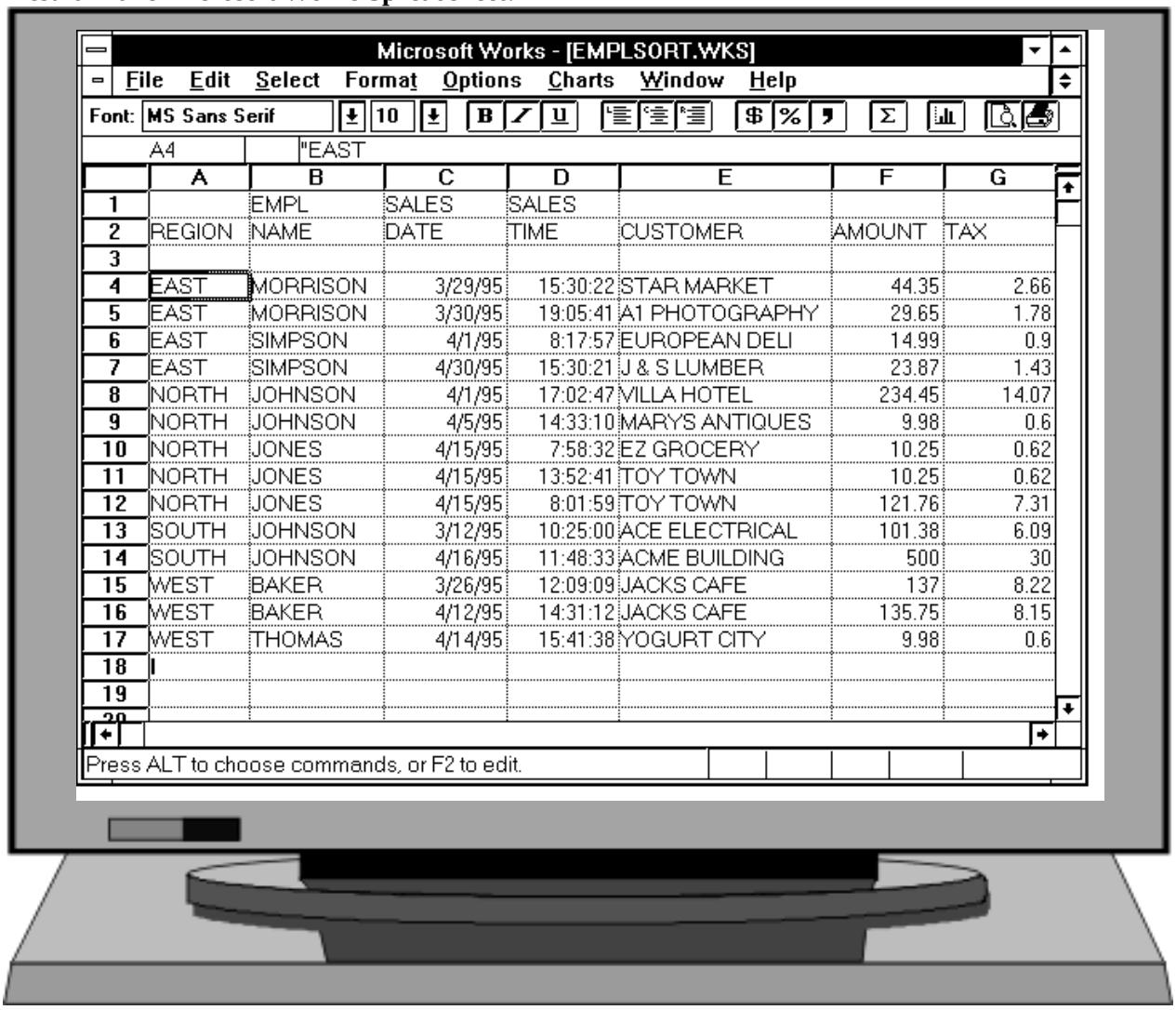
If you prefer, you can let Spectrum Writer *automatically* sort your PC file for you. To have your PC file automatically sorted on its first 5 columns of data, simply specify the AUTOSORT option, like this:

```
OPTIONS: AUTOSORT
```

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Result in this Microsoft Works Spreadsheet:



	A	B	C	D	E	F	G
1		EMPL	SALES	SALES			
2	REGION	NAME	DATE	TIME	CUSTOMER	AMOUNT	TAX
3							
4	EAST	MORRISON	3/29/95	15:30:22	STAR MARKET	44.35	2.66
5	EAST	MORRISON	3/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
6	EAST	SIMPSON	4/1/95	8:17:57	EUROPEAN DELI	14.99	0.9
7	EAST	SIMPSON	4/30/95	15:30:21	J & S LUMBER	23.87	1.43
8	NORTH	JOHNSON	4/1/95	17:02:47	VILLA HOTEL	234.45	14.07
9	NORTH	JOHNSON	4/5/95	14:33:10	MARYS ANTIQUES	9.98	0.6
10	NORTH	JONES	4/15/95	7:58:32	EZ GROCERY	10.25	0.62
11	NORTH	JONES	4/15/95	13:52:41	TOY TOWN	10.25	0.62
12	NORTH	JONES	4/15/95	8:01:59	TOY TOWN	121.76	7.31
13	SOUTH	JOHNSON	3/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
14	SOUTH	JOHNSON	4/16/95	11:48:33	ACME BUILDING	500	30
15	WEST	BAKER	3/26/95	12:09:09	JACKS CAFE	137	8.22
16	WEST	BAKER	4/12/95	14:31:12	JACKS CAFE	135.75	8.15
17	WEST	THOMAS	4/14/95	15:41:38	YOGURT CITY	9.98	0.6
18							
19							

Figure 28. Using a SORT statement to specify the sort order of a PC file

Summary

Here is a summary of what we learned in this lesson:

- use the SORT statement to **sort your PC files**
- you can sort on **multiple sort fields**
- you can sort in either **ascending or descending** order

The next lesson will teach you how to create control breaks and include subtotals in your PC file.

To Learn More

There are some additional features associated with the SORT statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional features include:

- creating a **control break** with the SORT statement (page 177)
- requesting **totals and statistics** with the SORT statement (page 186)

The complete syntax for the SORT statement is given in Chapter 10, "Control Statement Syntax" (page 595).

Lesson 5. How to Create Control Breaks

This lesson teaches you what control breaks are, and shows how to request them for your PC file. This lesson also shows how to include subtotals and other statistics in your PC file. The control statement discussed is:

- the BREAK statement

How to Use the BREAK Statement

If you are not a programmer the term "control break" may be new to you. But it is a very simple concept. And as you will see, control breaks can make your PC files much more useful.

Consider the result of sorting a PC file on some field. By sorting on a field, we *group together* all the rows that contain a particular value for that field. For example, in the spreadsheet on page 106 we sorted first of all on the REGION field. As you can see, this caused the spreadsheet rows to be grouped together by region. All of the rows for the East region appear together at the beginning of the spreadsheet. Next come all of the rows for the North region, and so on. By sorting on the REGION field, we grouped together all of the rows for each region.

Often it is desirable to perform special processing whenever one such group of rows ends and another group is about to begin. For example, you might want to have a row of totals for the group that just ended. You might also want a few blank rows after the totals to separate the different groups. Such processing is called **control break processing**. A **control break** is said to occur whenever one group of rows ends and another group is about to begin. The field that is being grouped (for example, REGION) is called the **control break field** (or often just the **break field**). A control break field *must* also be a sort field, since it is by being sorted that rows are grouped together in the first place.

You may designate any sort field as a control break field. Just name the field in a BREAK statement:

```
SORT:  REGION  EMPL-NAME  SALES-DATE
BREAK: REGION
```

The above statement makes REGION a control break field. Now we will get REGION totals in the resulting spreadsheet whenever one region ends and another region is about to begin.

Figure 29 shows an Excel spreadsheet obtained by using the BREAK statement above to produce a control break.

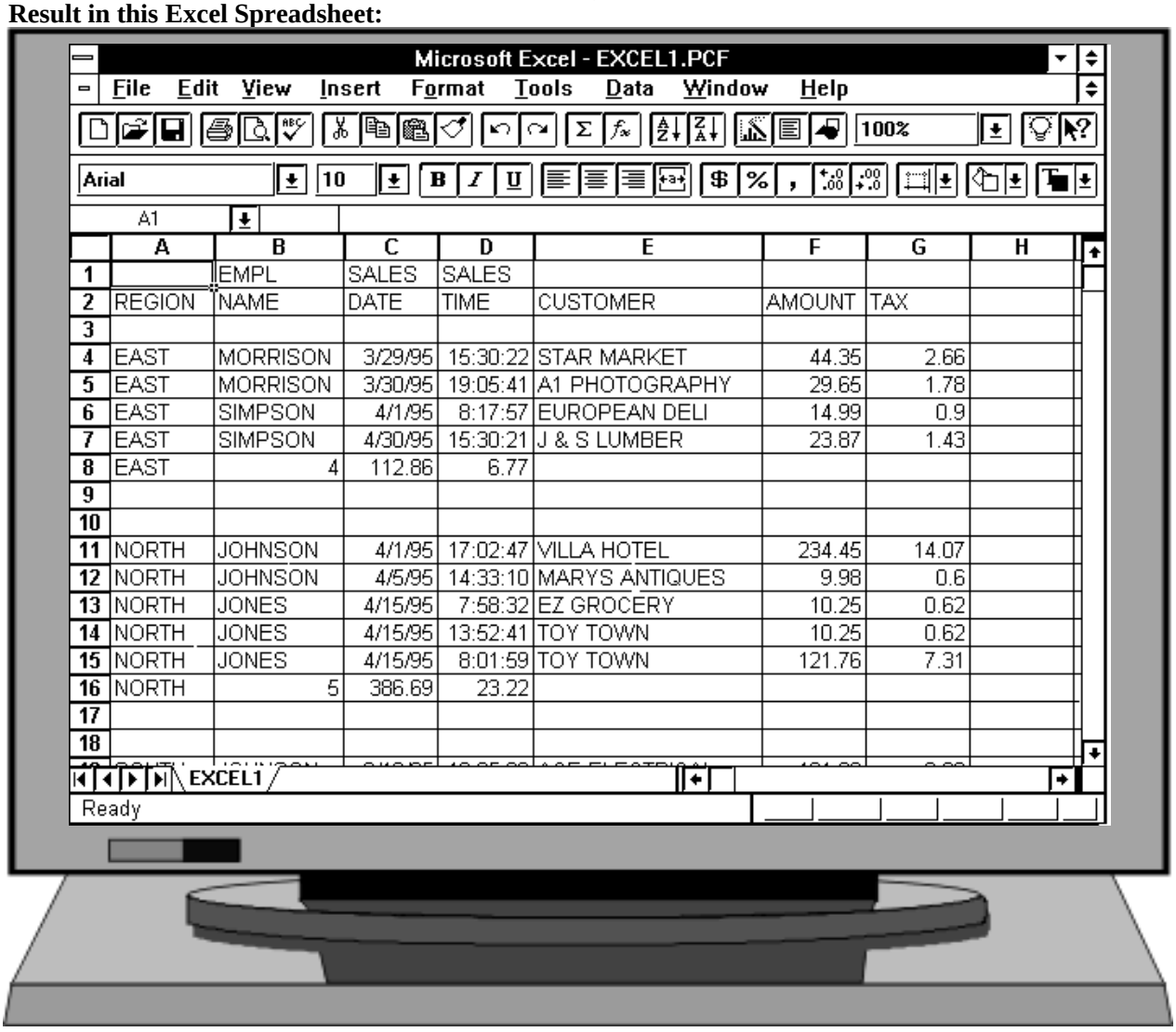
Customizing the Control Break

The Excel spreadsheet in **Figure 29** shows what Spectrum Writer's default total row looks like. It begins with the value of the break field just ended (the REGION field, in this example). The next column contains the number of items in the control group just ended. (For example, there are 4 items in the control group for the East region.) Following this are

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Result in this Excel Spreadsheet:



	A	B	C	D	E	F	G	H
1		EMPL	SALES	SALES				
2	REGION	NAME	DATE	TIME	CUSTOMER	AMOUNT	TAX	
3								
4	EAST	MORRISON	3/29/95	15:30:22	STAR MARKET	44.35	2.66	
5	EAST	MORRISON	3/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78	
6	EAST	SIMPSON	4/1/95	8:17:57	EUROPEAN DELI	14.99	0.9	
7	EAST	SIMPSON	4/30/95	15:30:21	J & S LUMBER	23.87	1.43	
8	EAST	4	112.86	6.77				
9								
10								
11	NORTH	JOHNSON	4/1/95	17:02:47	VILLA HOTEL	234.45	14.07	
12	NORTH	JOHNSON	4/5/95	14:33:10	MARYS ANTIQUES	9.98	0.6	
13	NORTH	JONES	4/15/95	7:58:32	EZ GROCERY	10.25	0.62	
14	NORTH	JONES	4/15/95	13:52:41	TOY TOWN	10.25	0.62	
15	NORTH	JONES	4/15/95	8:01:59	TOY TOWN	121.76	7.31	
16	NORTH	5	386.69	23.22				
17								
18								

Figure 29. Using the BREAK statement to create a control break with subtotals in a PC file

columns containing the total values for each numeric column in the spreadsheet (the AMOUNT and TAX fields, in this example).

The most common use of total rows is in "summary files" where the detail rows are suppressed, leaving just the total rows (see next lesson). Therefore, this default total row is designed to contain just the significant information for a control group. It does not contain any empty columns. If you are producing a spreadsheet that contains both detail rows and total rows, however, you may want to insert some blank columns in the total row. That lets you put the numeric totals in the same spreadsheet column as the corresponding detail values.

You can customize the total line at a control break by using the FOOTING parm in the BREAK statement. Consider this BREAK statement:

```
BREAK: REGION  NOTOTALS
       FOOTING(REGION ' '  ' '  ' '  ' '  AMOUNT(TOTAL)  TAX(TOTAL))
```

The above statement does two new things:

- the NOTOTALS parm suppresses Spectrum Writer's default total row at the control break
- the FOOTING parm describes a custom row to replace the default total row at each control break

The FOOTING parm works very much like the COLUMNS statement. You remember that the COLUMNS statement tells Spectrum Writer which columns are wanted in the *detail rows*. The FOOTING parm tell Spectrum Writer what columns are wanted in the *control break row*. The FOOTING parm above specifies that the contents of the REGION field should go in the first column. Then there will be four blank columns. (Each ' ' is a blank literal which results in a column that just contains a blank.) After the blank columns, the FOOTING parm specifies a column containing the total value of the AMOUNT field. And the last column contains the total value of the TAX field. By inserting four blank columns, the total AMOUNT and total TAX values line up with the detail rows. You can have as many FOOTING parms as you want in a BREAK statement. Each FOOTING parm describes one row to insert into the PC file at the control break.

You can also control the number of blank rows that appear at control breaks. By default, Spectrum Writer puts two blank rows after the total row at a control break (see page 109). Use the SPACE parm in your BREAK statement to request a different number of blank lines. For example:

```
BREAK: REGION SPACE(1)
```

The above statement requests just one blank row at the REGION control break. You may also specify SPACE(0) if you want *no* blank rows in your spreadsheet.

Figure 30 uses the FOOTING, NOTOTALS and SPACE parms to customize the control break.

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION  NOTOTALS
         SPACE(1)
         FOOTING(REGION ' '  ' '  ' '  ' '  AMOUNT(TOTAL)  TAX(TOTAL))
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Result in this Excel Spreadsheet:

Microsoft Excel - CUSTBRK.TXT

File Edit View Insert Format Tools Data Window Help

	A	B	C	D	E	F	G
1		EMPL	SALES	SALES			
2	REGION	NAME	DATE	TIME	CUSTOMER	AMOUNT	TAX
3							
4	EAST	MORRISON	3/29/95	15:30:22	STAR MARKET	44.35	2.66
5	EAST	MORRISON	3/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
6	EAST	SIMPSON	4/1/95	8:17:57	EUROPEAN DELI	14.99	0.9
7	EAST	SIMPSON	4/30/95	15:30:21	J & S LUMBER	23.87	1.43
8	EAST					112.86	6.77
9							
10	NORTH	JOHNSON	4/1/95	17:02:47	VILLA HOTEL	234.45	14.07
11	NORTH	JOHNSON	4/5/95	14:33:10	MARYS ANTIQUES	9.98	0.6
12	NORTH	JONES	4/15/95	7:58:32	EZ GROCERY	10.25	0.62
13	NORTH	JONES	4/15/95	13:52:41	TOY TOWN	10.25	0.62
14	NORTH	JONES	4/15/95	8:01:59	TOY TOWN	121.76	7.31
15	NORTH					386.69	23.22
16							
17	SOUTH	JOHNSON	3/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
18	SOUTH	JOHNSON	4/16/95	11:48:33	ACME BUILDING	500	30

CUSTBRK

Ready

Figure 30. Using FOOTING parms to customize the total row and create blank rows

Summary

Here is a summary of what we learned in this lesson:

- use the BREAK statement to specify a **control break field**
- control break fields must also be **sort fields**
- use the FOOTING parm to **customize the total row** at a control break
- use the SPACE parm to specify the number of **blank rows** at a control break

The next lesson will show you how to turn PC files with control breaks into "summary files."

To Learn More

There are some additional features associated with the BREAK statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional topics include:

- how to write one or more customized rows at the **beginning of a control break** (page 200)
- how to **customize the total row**, and the other statistical rows (page 182 and page 186)
- how to **suppress the total row** at a control break (page 185)
- how to show various **statistics** at control breaks (page 193)
- how to compute **percentages and ratios** that apply to an entire control group (page 202)
- how to have **multiple levels** of control breaks (page 204)

The complete syntax for the BREAK statement is given Chapter 10, "Control Statement Syntax" (page 481).

Lesson 6. How to Create Summary Files

This lesson teaches you how to produce summary files. The control statement discussed is:

- the OPTIONS statement

How to Create a Summary File

Sometimes you only need summarized data in your PC— not the detail data for each individual record. Why download the entire mainframe file and summarize the data on your PC. Instead, let Spectrum Writer perform the summarization for you on the mainframe. Then just download the small summary file to your PC.

Summarizing a mainframe file with Spectrum Writer is very easy.

For example, consider the Excel spreadsheet we created back on page 109. It is a detail spreadsheet that lists every sale made in every region. The control break on REGION causes a total row to appear after the detail rows for each region.

Now let's say we only want to download the total sales amount and tax amount for each region rather than the amounts for each individual sale. To do that, we need to summarize the file by region.

By adding the following statement, we can suppress the detail rows and retain just the region totals:

```
OPTIONS: SUMMARY
```

Figure 31 shows a Paradox table obtained by using the above statement. As you can see, the table has just four rows of actual data — one for each region in the mainframe file. The first column in each row contains a region name. The second column shows the number of records that were summarized in order to create that region's total. The last two columns are the total sales amount and tax amount for the sales in that region.

Using Spectrum Writer's summarization feature can be a tremendous benefit when working with very large mainframe files (perhaps containing millions of records). The summarization is done at mainframe speed, and you end up with a much smaller PC file to download to your PC.

These Control Statements:

```
OPTIONS: PC SUMMARY NOCOLHDG
INPUT:   SALES-FILE
SORT:    REGION  EMPL-NAME  SALES-DATE
BREAK:   REGION
COLUMNS: REGION  EMPL-NAME  SALES-DATE  SALES-TIME  CUSTOMER  AMOUNT  TAX
```

Result in this Paradox Table:

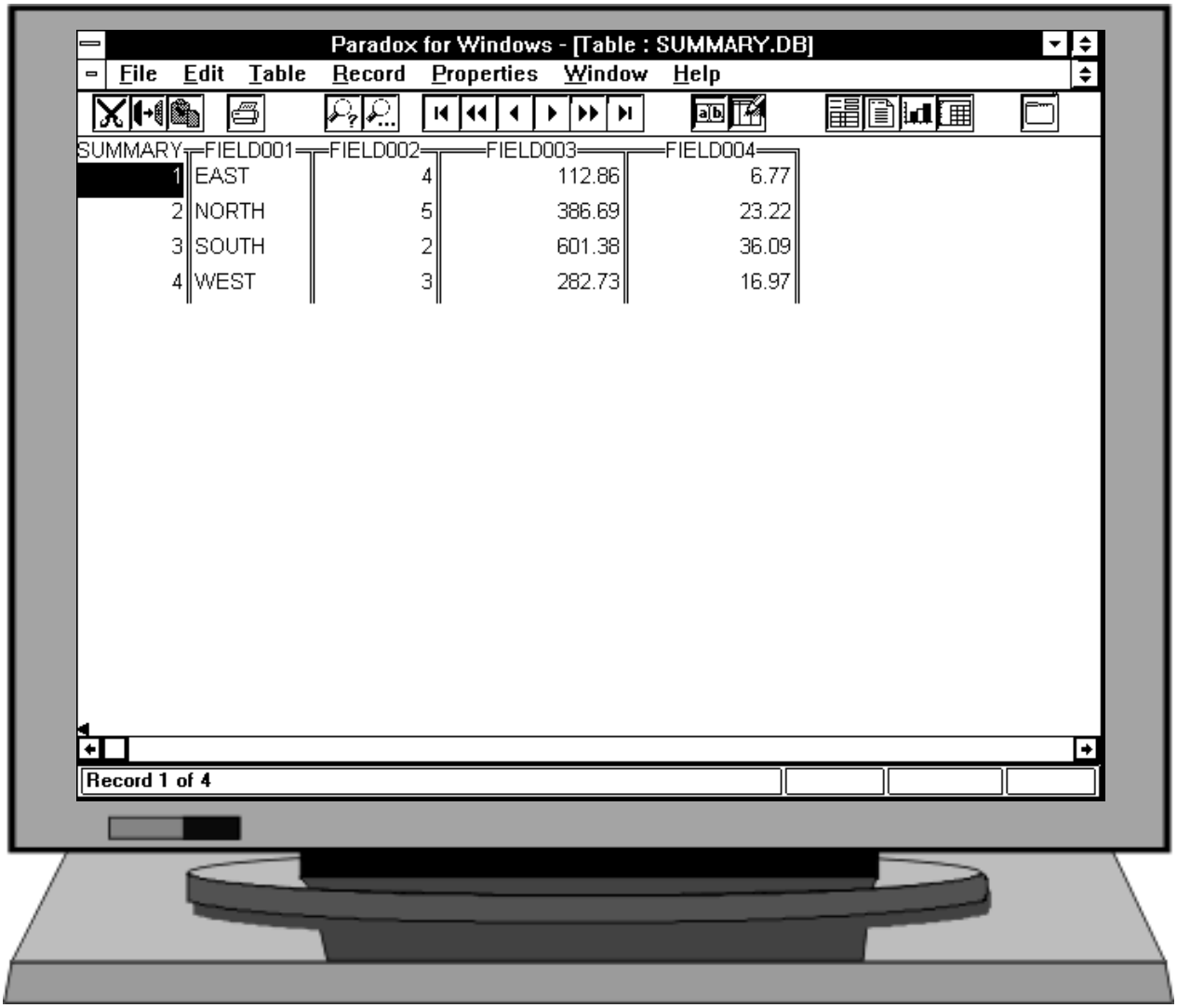


Figure 31. A Paradox table containing only summary data

Summary

Here is a summary of what we learned in this lesson:

- use the SUMMARY option (in the OPTIONS statement) to **create a summary file**
- a summary run **must have at least one control break field**

The next lesson will show you how to use data from more than one input file in a PC file.

To Learn More

There are some additional features associated with summary files which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Examples of additional features include:

- **customizing** the summary rows in your PC file (page 182)
- **obtaining statistics** (such as averages, maximums and minimums) in your summary file (page 186)
- creating **multiple levels** of summarization (page 204)
- including a **limited number of detail records** in each control group, creating spreadsheets such as "The Top 3 Sales in Each Region" (page 212)

Lesson 7. How to Use Data from More Than One File

This lesson teaches you how to read records from additional input files for use in your PC file. The control statement discussed is:

- the READ statement

All of the sample PC files produced so far have used data from only one input file. The data has come from the file specified in the INPUT statement, called the **primary input file**. There are times when all of the data needed for a particular PC file will not be found in just a single file. One of Spectrum Writer's most powerful features is its ability to link to *any number* of additional files to produce a PC file.

How Auxiliary Input Files Are Processed

Each PC file is allowed to have only one primary input file, specified in the INPUT statement. When data from additional input files is required, a READ statement is used. The READ statement causes a record to be read from another input file, called an **auxiliary input file**. You may have as many READ statements as you like in a single run.

By simply adding a READ statement to your request, you automatically make all of the fields from another whole file available for use in producing your PC file.

Here is how Spectrum Writer processes the primary and auxiliary input files. Spectrum Writer first reads a single record from the primary input file. (This file is always read *sequentially*, beginning with the first record in the file.) Next, if any auxiliary input files were specified, Spectrum Writer also reads one record from each of those files. (These files are always read *randomly*, using a key.) At this point, Spectrum Writer will have read one record from each of the input files. The fields from all of these records are now available for use in producing the PC file. These fields can be used:

- as columns of data
- as sort fields
- as control break fields
- in conditional expressions
- in calculations
- and in any other way that fields from the primary input file are used

After processing this set of records, Spectrum Writer then repeats the process. Another record is read sequentially from the primary input file. Then random reads are performed to each of the auxiliary input files. This next group of records is then used in making the PC file, and so on. This process is repeated until there are no more records left in the primary input file.

By simply adding a READ statement to your request, you automatically make all of the data fields from another file available for use in producing your PC file.

There is one important thing about auxiliary input files to keep in mind. Since these files are ready randomly, they *must* be keyed files or DB2 tables. (VSAM files are often keyed files.)

In a keyed file, each record has a unique "key" value associated with it. When a random read is made to such a file, a **read key** must be specified to identify which record to read. What read key should you tell Spectrum Writer to use when reading a record from an auxiliary input file? In order to be useful, the auxiliary input record should be somehow related to the primary input record. Usually, the record from the primary input file will contain the key of a corresponding record in the auxiliary input file. That key from the primary input file will be used as the read key.

> **Note:** If you are not familiar with such terms as "keyed files" and "read keys," ask your programmer to help you determine whether a particular file is keyed or not, and also to help you decide what read key to use.

How to Use the READ Statement

Now let's look at a concrete example of how to use the READ statement. Begin by considering **Figure 32**, which shows a spreadsheet that uses only a primary input file (the SALES–FILE). This spreadsheet shows information about each sale made by an employee. This spreadsheet includes columns for two fields that we haven't used in previous examples, so we'll explain them. They are the EMPL–NUM field and the PRODUCT–CODE field. The EMPL–NUM is the employee number of the employee who made the sale. The PRODUCT–CODE is a code that identifies the product that was sold to the customer.

Now, let's assume that we want this spreadsheet to also show each employee's social security number. The social security number is not available in the SALES–FILE. But it is a field in the EMPL–FILE. (See page 91.) In order to produce such a spreadsheet, we need data from a second input file — the EMPL–FILE.

The EMPL–FILE is a keyed VSAM file. Its key is the 3–byte employee number. The records in the SALES–FILE also contain an employee number, so we can use that field as the "read key" to use when we read the EMPL–FILE. That means we can make the EMPL–FILE an auxiliary input file by simply adding this statement:

```
READ:  EMPL-FILE   READKEY(EMPL-NUM)
```

This READ statement tells Spectrum Writer to use the EMPL–NUM field from the records in the SALES–FILE as a key to read an auxiliary record from the EMPL–FILE. All control statements after this READ statement may now refer to the fields in the EMPL–FILE, as well as to those in the SALES–FILE. So, we can now add the SOCIAL–SEC–NUM field from the EMPL–FILE to our COLUMNS statement:

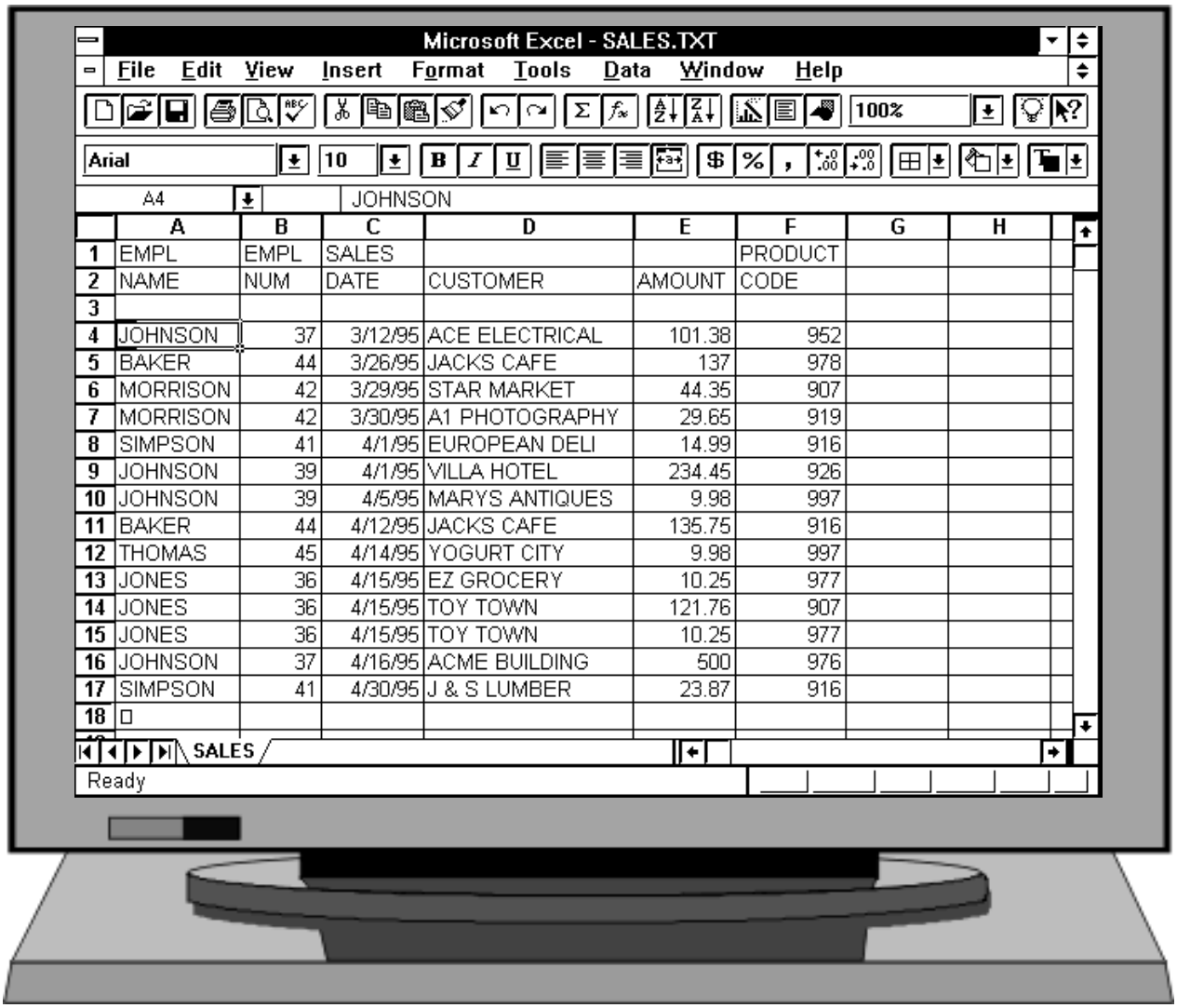
```
COLUMNS: EMPL-NAME  SALES-FILE.EMPL-NUM  SOCIAL-SEC-NUM
         SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```

Notice that in the above COLUMNS statement we must now prefix the EMPL–NUM field with a record name (like this: SALES–FILE.EMPL–NUM). This is because after the READ statement, EMPL–NUM is no longer a unique field name. A field by that name exists in both the SALES–FILE and the EMPL–FILE. (See Appendix F, "Files Used in Examples" on page 648.) Since the EMPL–NUM will have the same value in both of the records, it doesn't really matter which one we specify in the COLUMNS statement, but we do have to specify a unique name.

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
COLUMNS: EMPL-NAME  EMPL-NUM  SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```

Result in this Excel Spreadsheet:



	A	B	C	D	E	F	G	H
1	EMPL	EMPL	SALES			PRODUCT		
2	NAME	NUM	DATE	CUSTOMER	AMOUNT	CODE		
3								
4	JOHNSON	37	3/12/95	ACE ELECTRICAL	101.38	952		
5	BAKER	44	3/26/95	JACKS CAFE	137	978		
6	MORRISON	42	3/29/95	STAR MARKET	44.35	907		
7	MORRISON	42	3/30/95	A1 PHOTOGRAPHY	29.65	919		
8	SIMPSON	41	4/1/95	EUROPEAN DELI	14.99	916		
9	JOHNSON	39	4/1/95	VILLA HOTEL	234.45	926		
10	JOHNSON	39	4/5/95	MARYS ANTIQUES	9.98	997		
11	BAKER	44	4/12/95	JACKS CAFE	135.75	916		
12	THOMAS	45	4/14/95	YOGURT CITY	9.98	997		
13	JONES	36	4/15/95	EZ GROCERY	10.25	977		
14	JONES	36	4/15/95	TOY TOWN	121.76	907		
15	JONES	36	4/15/95	TOY TOWN	10.25	977		
16	JOHNSON	37	4/16/95	ACME BUILDING	500	976		
17	SIMPSON	41	4/30/95	J & S LUMBER	23.87	916		
18								

Figure 32. A spreadsheet that uses only the primary input file

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
READ:    EMPL-FILE  READKEY(EMPL-NUM)
SORT:    SOCIAL-SEC-NUM
COLUMNS: EMPL-NAME  SALES-FILE.EMPL-NUM  SOCIAL-SEC-NUM
         SALES-DATE  CUSTOMER  AMOUNT  PRODUCT-CODE
```

Result in this Excel spreadsheet:

Microsoft Excel - SALEEMPL.XLS

File Edit View Insert Format Tools Data Window Help

Arial 10 100%

A6 JOHNSON

	A	B	C	D	E	F	G	H
1		SALES						
2		FILE	SOCIAL					
3	EMPL	EMPL	SEC	SALES			PRODUCT	
4	NAME	NUM	NUM	DATE	CUSTOMER	AMOUNT	CODE	
5								
6	JOHNSON	39	4779981	4/5/95	MARYS ANTIQUES	9.98	997	
7	JOHNSON	39	4779981	4/1/95	VILLA HOTEL	234.45	926	
8	JONES	36	12098765	4/15/95	EZ GROCERY	10.25	977	
9	JONES	36	12098765	4/15/95	TOY TOWN	10.25	977	
10	JONES	36	12098765	4/15/95	TOY TOWN	121.76	907	
11	SIMPSON	41	112050456	4/30/95	J & S LUMBER	23.87	916	
12	SIMPSON	41	112050456	4/1/95	EUROPEAN DELI	14.99	916	
13	THOMAS	45	776838221	4/14/95	YOGURT CITY	9.98	997	
14	BAKER	44	878190156	4/12/95	JACKS CAFE	135.75	916	
15	BAKER	44	878190156	3/26/95	JACKS CAFE	137	978	
16	MORRISON	42	900120556	3/30/95	A1 PHOTOGRAPHY	29.65	919	
17	MORRISON	42	900120556	3/29/95	STAR MARKET	44.35	907	
18	JOHNSON	37	912040334	3/12/95	ACE ELECTRICAL	101.38	952	

DOCA313

Ready

Figure 33. A spreadsheet that uses a READ statement to specify an auxiliary input file

In this case we specified the EMPL–NUM field from the SALES–FILE. (For more information on using "record names" to qualify field names, see "How to Name the Input File Records" on page 228.)

Figure 33 (page 119) shows an Excel spreadsheet obtained by using the above statements. The spreadsheet now has the desired new column showing each employee's social security number. Notice that we also *sorted* the PC file on SOCIAL–SEC–NUM. Remember that you can use fields from auxiliary input files in any way that you can use fields from the primary input file.

"One-to-Many" Random Reads

Normally, Spectrum Writer reads just a single record from your auxiliary input file — the record whose key field matches the READKEY value. If you want to use *all of the records* which match the READKEY (or partial READKEY), add the MULTI parm to your READ statement. When the READ statement has the MULTI parm, Spectrum Writer creates and processes "logical input records" by matching the primary input file row with *each* qualifying record from the auxiliary input file. For more information on how the MULTI parm works, see "How to Perform "One–to–Many" Reads" on page 232.

How to Use Multiple READ Statements

You may use as many READ statements as you like in a run. For example, the Excel spreadsheet in **Figure 34** uses two READ statements.

The primary input file is once again the SALES–FILE, which contains one record for each sale made by an employee. It is specified in the INPUT statement:

```
INPUT: SALES-FILE
```

To obtain additional data about the employee who made each sale, we use a READ statement for the EMPL–FILE (just like in the preceding example). The EMPL–NUM field in the SALES–FILE contains the key necessary to read the correct EMPL–FILE record.

```
READ:  EMPL-FILE  READKEY(EMPL-NUM)
```

To obtain additional information about each *product* sold, a second READ statement names the PRODUCT–FILE as another auxiliary input file. (The PRODUCT–FILE is also described in Appendix F, "Files Used in Examples" on page 648.)

However, there is one minor complication in reading records from this file. The key in the PRODUCT–FILE records is 4 bytes long. It consists of the letter "P" followed by a 3–byte product code. The SALES–FILE does not contain a field which can be used *directly* as the read key to the PRODUCT–FILE. But it does contain the 3–byte PRODUCT–CODE field, which we can use to build the 4–byte read key. A COMPUTE statement is therefore used to create a new field (called PKEY) which consists of the letter "P" followed by the product code. This computed field is then used as the read key in the READ statement for the PRODUCT–FILE:

```
COMPUTE: PKEY = 'P' + PRODUCT-CODE
READ:    PRODUCT-FILE READKEY(PKEY)
```

By having two READ statements in addition to the INPUT statement, the PC file now uses data from three input files. Data from all of these files can be used in any of the subsequent

These Control Statements:

```
OPTIONS: PC
INPUT:   SALES-FILE
READ:    EMPL-FILE     READKEY(EMPL-NUM)
COMPUTE: PKEY = 'P' +  PRODUCT-CODE
READ:    PRODUCT-FILE  READKEY(PKEY)
SORT:    SOCIAL-SEC-NUM
COLUMNS: EMPL-NAME  SALES-FILE.EMPL-NUM  SOCIAL-SEC-NUM
         SALES-DATE  CUSTOMER  AMOUNT
         PRODUCT-CODE  PRODUCT-DESC
```

Result in this Excel Spreadsheet:

Microsoft Excel - PRODUCT.XLS

File Edit View Insert Format Tools Data Window Help

Arial 10 100%

H6 MAILING LABELS

	A	B	C	D	E	F	G	H
1		SALES						
2		FILE	SOCIAL					
3	EMPL	EMPL	SEC	SALES			PRODUCT	PRODUCT
4	NAME	NUM	NUM	DATE	CUSTOMER	AMOUNT	CODE	DESC
5								
6	JOHNSON	39	4779981	4/5/95	MARYS ANTIQU	9.98	997	MAILING LABELS
7	JOHNSON	39	4779981	4/1/95	VILLA HOTEL	234.45	926	DESK CALENDARS
8	JONES	36	12098765	4/15/95	EZ GROCERY	10.25	977	PAPER CLIPS
9	JONES	36	12098765	4/15/95	TOY TOWN	10.25	977	PAPER CLIPS
10	JONES	36	12098765	4/15/95	TOY TOWN	121.76	907	INKPADS
11	SIMPSON	41	112050456	4/30/95	J & S LUMBER	23.87	916	RED PENS
12	SIMPSON	41	112050456	4/1/95	EUROPEAN DEL	14.99	916	RED PENS
13	THOMAS	45	776838221	4/14/95	YOGURT CITY	9.98	997	MAILING LABELS
14	BAKER	44	878190156	4/12/95	JACKS CAFE	135.75	916	RED PENS
15	BAKER	44	878190156	3/26/95	JACKS CAFE	137	978	HOLE PUNCHERS
16	MORRISO	42	900120556	3/30/95	A1 PHOTOGRAF	29.65	919	GREEN PENS
17	MORRISO	42	900120556	3/29/95	STAR MARKET	44.35	907	INKPADS
18	JOHNSON	37	912040334	3/12/95	ACE ELECTRICA	101.38	952	PENCILS (NO. 1)

DOCA314

Ready

Figure 34. A spreadsheet that uses two READ statements to specify two auxiliary input files

control statements. In the Excel spreadsheet in **Figure 34**, the COLUMNS statement uses one field from each of the auxiliary input files. It uses the SOCIAL–SEC–NUM field from the EMPL–FILE and the PRODUCT–DESC field from the PRODUCT–FILE:

```
COLUMNS: EMPL-NAME
         SALES-FILE.EMPL-NUM
         SOCIAL-SEC-NUM
         SALES-DATE
         CUSTOMER
         PRODUCT-CODE
         PRODUCT-DESC
```

Summary

Here is a summary of what we learned in this lesson:

- the READ statement is used to read records from **auxiliary input files**
- the file named in a READ statement must be a **keyed file** (or a DB2 table)
- you may have as **many READ statements as you like** in a single run

To Learn More

There are some additional features associated with the READ statement which we have not covered in this lesson. Some of these additional features are discussed as topics in Chapter 4, "Beyond the Basics." Some additional features include:

- how to assign a **record name** to the records read from auxiliary input files (page 228)
- how to read multiple records (containing **different keys**) from the same auxiliary input file (page 224)
- how to use **data from one auxiliary input file as the read key** to another auxiliary input file (page 226)
- how to specify **generic keys** and **"KGE" keys** in the READ statement (page 230)
- how to read multiple records (with the **same key or partial key**) from the auxiliary input file (page 232)
- what happens when **no record is found** for a particular read key (page 229)
- how to determine whether the read for a particular key was **successful** or not (page 230)
- how to use the READ statement to obtain data from a **DB2 table or view** (page 397)
- how to use the ONIOERROR option to **increase the severity** of I/O errors on an input file (page 586)

The complete **syntax** for the READ statement, as well as a more detailed **narrative** of how Spectrum Writer assembles input records during the report process, is given in Chapter 10, "Control Statement Syntax" (page 578).

Chapter 4. Beyond the Basics

Chapter Table of Contents

Chapter 4. Beyond the Basics 123

Additional Features in the COLUMNS Statement 125
- Writing Print Expressions 126
- How to Change the Column Headings 130
- Special Options Related to Column Headings 133
- How to Change the Width of a Column 135
- How to Change the Way Dates, Times and Numbers Are Formatted 137
- Formatting Tips for International Users 140
- How to Format Data as ASCII 143
- How to Blank Out Repeating Values 144
- How to Change the Justification of Data within a Column 146
- How to Specify Which Columns to Total 148
- How to Produce Multi–Line Reports 151
- How to Change the Report Margins 154
- How to Print Bar Graphs 154
- How to Print Vertical Lines between Report Columns 156
- Including All Fields in the COLUMNS Statement 158

What If You Run Out of Room? 160
Why Do I See ****X**** in My Report? 160
Customizing the Report Titles 161
- How to Include Data from a File in the Title 161
- How to Include the Page Number, Date and Time in a Title 163
- How to Change the Appearance of Items in the Title 165
- How to Split the Title into Left Aligned, Centered, and Right Aligned Parts 168
- Special Options Related to Titles 175
- How to Print "Titles" at the Bottom of Each Page 175

Customizing the Control Breaks 177
- How to Change the Control Break Spacing 178
- How a Default Total Line Looks 180
- How to Customize the Total Line at a Control Break 182
- How to Suppress the Total Line at a Control Break 185
- How to Customize the Statistical Lines at a Control Break 186
- How to Print Customized Footing Lines at a Control Break 188
- How to Print the Number of Items in a Control Group 198
- How to Print Header Lines at the Beginning of a Control Group 200

Computing True Percentages and Ratios at Control Breaks 202
Reports with Multiple Control Breaks 204
How to Customize the Grand Totals 207
How to Produce Summary Reports 209
Printing a "Line Number" in Your Report 211
How to Create "Top 10" Type Reports 212
How to Count "Occurrences" in a File 214
How to Break Totals Down into Categories 217
How to Make “Crosstab” Reports 219
- A Simple Crosstab Report 219

Another Crosstab Report 221
Working With Multiple Input Files 224
Using Multiple READ Statements for the Same File 224
How to Chain READ Statements 226
How to Name the Input File Records 228
How Missing Records Are Handled 229
Testing for Missing Records 230
How I/O Errors Are Handled 230
Using Generic and KGE Keys 230
How to Perform "One–to–Many" Reads 232
Working with "Batched" Input Files 234
Working With Arrays 237
Using Normalization to Process Arrays 237
The NORMALIZE Parm 240
File Definition Tips for Records with Arrays 243
Normalizing Nested Arrays 244
Normalizing Multiple, Non-Nested Arrays 245
Normalizing only Certain Records 247
Normalizing an Auxiliary Input File 248
Normalization Errors 248
How to Print a Variable Number of Lines Per Input Record 249
Variable Number of Lines — Strategy 1 249
Variable Number of Lines — Strategy 2 254
Putting a Variable Number of Items on a Single Line 257
Creating PC Files from Non-Spectrum Writer Reports 258
Working with SMF Records 263
Working with Date Fields 269
Working with Time Fields 272
Producing Files for Non-Standard PC Programs 275
Producing Files for Mainframe Programs 280
How to "Subset" Mainframe Files 283
How to Sort Mainframe Files 283
Computing Percent of Totals 284
Creating Multiple Reports in a Single Run 289

Chapter 4. Beyond the Basics

This chapter is a user's guide to some of Spectrum Writer's additional features. Many of the control statements introduced earlier in Chapter 2, "How to Request a Report" and Chapter 3, "How to Request a PC File" are discussed in more detail in this chapter. Many reports and PC files won't require these more advanced features. But as your requests become more and more sophisticated, you may want to use some of the techniques and features illustrated in this chapter.

Additional Features in the COLUMNS Statement

We saw in previous chapters that the basic purpose of the COLUMNS statement is to name the columns desired in a report or PC file. The COLUMNS statement also has many other features that can be used to customize how a report or PC file looks. The following sections explain:

- how to include a column of **literal text** in a report or PC file (page 126)
- how to change the **spacing** between report columns (page 128)
- how to change the **column headings** (page 130 and page 133)
- how to change the **width** of a column (page 133)
- how to change the way **dates, times and numbers** are formatted (page 137)
- how to format dates, times and numbers for **international users** (page 140)
- how to format data as **ASCII** rather than EBCDIC (page 143)
- how to **blank out repeating values** (page 144)
- how to change the **justification** of data within a column (page 146)
- how to change which columns are **totalled** (page 148)
- how to produce **multi–line reports or multi–row PC files** (page 151)
- how to print **bar graphs** in a report (page 154)
- how to put a **text (such as a vertical line) between report columns** (page 156)
- how to change the report **margins** (page 154)
- how to **show columns for every field** in the input file (page 158)

Writing Print Expressions

This section explains:

- how to write **print expressions** for the COLUMNS statement
- which **fields** may appear in the COLUMNS statement
- how to include **literal texts** in the report lines
- how to specify the **number of spaces** that should appear between columns
- how **parms** can be used to customize the way a column is processed

Some of the features discussed in this section are illustrated in the sample report shown in **Figure 35**.

The contents of the COLUMNS statement is simply a **print expression**. Print expressions are used in a number of different control statements. They tell Spectrum Writer how to build one print line that will be used in a report. In the COLUMNS statement, the print expression tells how to build a detail report line for the main body of the report. (When creating PC files, the print expression tells how to build the detail output records.)

As with other print expressions in Spectrum Writer, just list one or more **items** to print.

```
COLUMNS: item1 item2 item3 ...
```

Each **item** can be either a **literal text** or a **field name**. In addition (only in COLUMNS statement print-expressions) an item can be a **record name**.

To put a **literal** text in a column of the report, simply enclose the text in either apostrophes or quotation marks. For example, the following statement causes the words NEW TEL:---------- to appear in each line of a report:

```
COLUMNS: 'NEW TEL:----------'
```

To put **data from a field in the input file** into a column of the report, simply list the desired field name. (Do *not* put the field name in apostrophes or quotation marks.) For example, the following statement causes the contents of the TELEPHONE field to appear in a report column:

```
COLUMNS: TELEPHONE
```

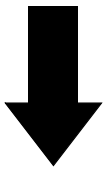
Each field listed must be "available" to Spectrum Writer at the time the COLUMNS statement is processed. That is, each field name must be one of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)
- a **built–in** field (see Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields)

To automatically create a column in the report (or output file) for **every field in the input file**, simply specify the desired record name. (Do *not* put the record name in apostrophes

These Control Statements:

```
INPUT:    EMPL-FILE
TITLE:    'TELEPHONE SIGNUP LIST'
COLUMNS:  EMPL-NUM    5
          LAST-NAME
          FIRST-NAME
          'OLD TEL:'  0
          TELEPHONE   2
          'NEW TEL: ----------'
```



Produce this Report:

```
                              TELEPHONE SIGNUP LIST

EMPL          LAST            FIRST
NUM           NAME            NAME            TELEPHONE            ____________________

036      JONES           JERRY           OLD TEL:(415) 555-7653  NEW TEL: ----------
037      JOHNSON         THOMAS          OLD TEL:(602) 555-6654  NEW TEL: ----------
039      JOHNSON         LINDA           OLD TEL:(415) 555-6785  NEW TEL: ----------
040      MACDONALD       RICHARD         OLD TEL:(415) 555-9887  NEW TEL: ----------
041      SIMPSON         TIMOTHY         OLD TEL:(818) 555-1887  NEW TEL: ----------
042      MORRISON        MICHAEL         OLD TEL:(818) 555-4748  NEW TEL: ----------
043      CHRISTOPHERSON  MELISSA         OLD TEL:(602) 555-4556  NEW TEL: ----------
044      BAKER           VIVIAN          OLD TEL:(415) 555-1209  NEW TEL: ----------
045      THOMAS          MARTIN          OLD TEL:(415) 555-1152  NEW TEL: ----------

*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the LAST–NAME column is 5 spaces over from the EMPL–NUM column
- the literal texts "OLD TEL:" and "NEW TEL: ----------" appear in each line of the report
- the spacing factor of zero puts zero spaces between the "OLD TEL:" column and the TELEPHONE column
- the literal text columns do not have default column headings

Figure 35. Using spacing factors and literal texts in the COLUMN statement

or quotation marks.) For example, the following statement causes all fields in the SALES-FILE to appear in report columns:

```
COLUMNS: SALES-FILE
```

Use of record names in the COLUMNS statement is discussed further in "Including All Fields in the COLUMNS Statement" on page 158

As in other print expressions, you may also customize the print line by using optional **spacing factors** and **parms**. So, the full syntax for the COLUMNS statement is this:

```
COLUMNS:  [n] item1(parms) [n] item2(parms) [n] item3(parms) ...
```

The optional **spacing factor** [n] is the number of blank spaces to leave between two columns in the report. If you omit the spacing factor, the default is for *one* blank space to appear between columns. (A spacing factor of zero is allowed if you want *no* spaces between two columns of your report.) As an example, the following statement causes two blanks to appear between the LAST–NAME and the FIRST–NAME columns, and causes five blanks to appear between the FIRST–NAME and the HIRE–DATE columns:

```
COLUMNS: LAST-NAME 2 FIRST-NAME 5 HIRE-DATE
```

Note: To change the *default* spacing factor (between all columns), use the COLSPACE parm of the OPTIONS statement (page 560).

The optional **parms** are used to provide details about how to display individual columns in the report. You may specify one or more parms, enclosed in parentheses, immediately following an item in the print expression. (Do *not* leave a space between the item and the first parenthesis.) You may use any combination of parms, in any order. Separate the parms with a comma and/or blanks. For example, the following statement has a width parm and a justification parm for the LAST–NAME field:

```
COLUMNS: LAST-NAME(50, CENTER)  FIRST-NAME
```

The following table shows the parms that are available in the COLUMNS statement. Subsequent sections of this chapter explain in detail how to use each of these parms.

COLUMNS STATEMENT PARMS	
PARM	**DESCRIPTION**
ACCUM/NOACCUM	Specifies whether the column should be accumulated or not. Accumulated columns receive totals at control breaks and at the end of the report. For more information on using these parms, see page 148. The following example specifies that the TOTAL–SALES column should not be accumulated (and therefore not totalled): `COLUMNS:  TOTAL-SALES(NOACCUM)`
ASCII	Specifies that the field should be formatted in ASCII, rather than in EBCDIC `COLUMNS:  REGION(ASCII) SALES-DATE(LONG1, ASCII)` See page 143 for more information on creating ASCII output files.

COLUMNS STATEMENT PARMS (CONTINUED)	
PARM	**DESCRIPTION**
BIZ	Means "blank if zero." Specifies that the column should be left blank whenever the numeric, date or time item contains zeros. The following example specifies that the TOTAL-SALES and SALES-TIME columns should be left blank whenever their value is zero. `COLUMNS: TOTAL-SALES(BIZ) SALES-TIME(BIZ)`
'column heading'	Specifies the column heading to be used for an item. For more information on using the column heading parm, see page 130. The following example specifies that the column heading for the LAST–NAME column should be "SELLERS LAST NAME": `COLUMNS: LAST-NAME('SELLERS LAST NAME')`
display–format	Specifies how to format the field in the report column. A complete list of display formats appears in Appendix B, "Display Formats" (page 617). For more information on using a display format parm, see page 137. The following example specifies that the HIRE–DATE column should be displayed in the LONG1 format, with the month name spelled out: `COLUMNS: HIRE-DATE(LONG1)`
LEFT/ CENTER/ RIGHT	Specifies how to justify the contents of a column. For more information on using a justification parm, see page 146. The following example specifies that the contents of the LAST–NAME column should be center justified: `COLUMNS: LAST-NAME(CENTER)`
NOREPEAT/ NOREPEATPAGE	Specifies that "repeating values" in a column should not be printed. (Blanks will appear instead.) NOREPEAT specifies that repeated values should not be printed anywhere except in the first line of each page and the first line of each control group. NOREPEATPAGE specifies that repeated values should not be printed anywhere except in the first line of each page. For example: `COLUMNS: LAST-NAME(NOREPEAT)`
width	This numeric parm specifies how wide the report column should be. For more information on using a width parm, see page 135. The following example specifies that the TOTAL–SALES column of the report should be only 6 characters wide: `COLUMNS: TOTALS-SALES(6)`

How to Change the Column Headings

This section explains:

- how Spectrum Writer determines **default column headings**
- how to **specify your own column headings**
- how to **suppress column headings**

Most of the features discussed in this section are illustrated in the sample report in **Figure 36**.

If you do not specify a column heading for a field in the COLUMNS statement, Spectrum Writer uses a default column heading. The default heading will be:

- the column heading (if any) specified when the field was first defined (in a FIELD or COMPUTE statement), or
- the field name itself, broken apart at each dash or underscore, with each part of the name going onto a separate heading line. (For example, the default column heading for LAST–NAME is a two–line heading, with "LAST" on one line and "NAME" on the next line, as illustrated in **Figure 35** on page 127.)

Note: By default, column headings are *not* automatically generated for multi–line reports (those using more than one COLUMNS statement). To learn how to create column headings for multi–line reports, see the section titled "How to Produce Multi–Line Reports" on page 151.

To specify your own column heading for a field, put your column heading in parentheses immediately after the field name. (Do *not* leave a space between the field name and the first parenthesis.) Enclose the column heading in either apostrophes or quotation marks. For example:

```
COLUMNS:  LAST-NAME("EMPLOYEE'S LAST NAME")
```

The above statement would cause the text "EMPLOYEE'S LAST NAME" to be used as the column heading for the LAST–NAME column. Since this is a rather long heading, you may want to split it onto two lines. Use the "vertical bar" character (|) within the column heading text to indicate where to split the text into separate lines. You may use as many lines for the column heading as you like, but most reports look best with no more than three or four lines of column headings. Here is an example of the use of the vertical bar to break the column heading into two lines:

```
COLUMNS:  LAST-NAME("EMPLOYEE'S|LAST NAME")
```

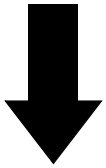
The example above will cause a two–line column heading to be used for the LAST–NAME column. The first heading line will contain the word "EMPLOYEE'S" and the second line will have the words "LAST NAME". The following example shows how to make a three–line column heading for the SEX column:

```
COLUMNS:  SEX('S|E|X')
```

In the above statement, each of the three column headings lines now has only one character. Since the SEX field is also only one character long, the column will now default to being

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE LISTING'
COLUMNS: LAST-NAME("EMPLOYEE'S|LAST NAME")
         FIRST-NAME('NAME           ')
         EMPL-NUM('  ')
         HIRE-DATE('')
         SEX('S|E|X')
         SEX
         '----------'('DELIVERY|DATE')
```



Produce this Report:

```
                    EMPLOYEE LISTING
                                              S
  EMPLOYEE'S                                  E      DELIVERY
   LAST NAME    NAME           ___           X SEX    DATE

JONES           JERRY          036 01/31/80 M  M  ----------
JOHNSON         THOMAS         037 06/21/75 M  M  ----------
JOHNSON         LINDA          039 11/25/79 F  F  ----------
MACDONALD       RICHARD        040 07/04/82 M  M  ----------
SIMPSON         TIMOTHY        041 12/01/82 M  M  ----------
MORRISON        MICHAEL        042 11/30/79 M  M  ----------
CHRISTOPHERSON  MELISSA        043 08/15/81 F  F  ----------
BAKER           VIVIAN         044 06/04/82 F  F  ----------
THOMAS          MARTIN         045 06/04/82 M  M  ----------


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the LAST-NAME column heading is split onto two lines
- the FIRST–NAME column heading ("NAME") is left–justified
- the EMPL–NUM column has no column heading, but does have the underscores
- the HIRE–DATE column has no column heading and no underscores
- the SEX column with the stacked heading takes up only one character
- the column of literal text now has a column heading

Figure 36. Specifying your own column headings

one character wide, rather than three. Stacking column headings like this can help you squeeze more columns into your report.

Note: The vertical bar is the "Shift 1" key on most mainframe terminals. When working at a PC running terminal emulation software, you will probably not see a key with this symbol on it. Some terminal emulator programs use the "pipeline" key as the vertical bar key. Some others use the right–hand square bracket key "]" for this purpose.

Note: You can use the HDGSEP parm of the OPTIONS statement to select a **different character** to use as the separator character for column heading texts (page 563). Here is an example that uses a slash, rather than a vertical bar, to separate column headings lines:

```
OPTION:  HDGSEP('/')
COLUMNS: LAST-NAME("EMPLOYEE'S/LAST NAME")  SEX('S/E/X')
```

Note: If you find that you frequently have to override column headings in the COLUMNS statement, consider changing the field's *default* column heading. Default column headings are specified in the FIELD statement (page 350) or the COMPUTE statement (page 506).

Column headings are automatically *centered* over their columns in reports (but not in PC files). Therefore, you do not need to try to add extra spaces within your column headings to force correct alignment. If for some reason you want left– or right–justified column headings, then you should include enough leading or trailing blanks *within* the heading text to take up the whole width of the column. For example, if LAST–NAME is a 15 character column, and you want the column heading "NAME" to be appear **left–justified** over it, use 11 trailing blanks within the column heading text, like this:

```
COLUMNS:  LAST-NAME('NAME           ')
```

You can also use *leading* blanks to force **right–justification** of a column heading:

```
COLUMNS:  AMOUNT('      AMOUNT')
```

If you do not want any column heading for a particular column, you can use an all blank column heading text, like this:

```
COLUMNS: LAST-NAME(' ')
```

The above example causes blanks to be used as the column heading for the LAST–NAME column in the report.

Following the last column heading line, Spectrum Writer prints an additional line of underscores to indicate the exact width of each column. (This underscore line *overprints* the final column heading text line— it is not a separate print line.) These underscores appear even for columns with blank column heading texts. To suppress even the **underscores** for a column, use a null column heading text — without even blanks within it. For example:

```
COLUMNS:  LAST-NAME('')
```

The above example causes the LAST–NAME column to appear with no column headings and with no underscores.

To **suppress all column headings**, use the NOCOLHDGS parm of the OPTIONS statement (page 567). This option means that no column headings (and no underscore line) should print. This is often used when you want to specify all of the column heading lines yourself, using TITLE statements (see page 153).

Some printers do not support the overprinting of lines (as is needed to properly print the underscore line after the column headings). If this is the case and you want to suppress the entire underscore line, use the NOUNDERSCORES parm of the OPTIONS statement (as described on page 568).

Column headings are *not* automatically generated for **columns of literal text**. You may, however, specify your own column headings for literal texts just as you would for a field. The following example illustrates how to specify a column heading for a column of literal text:

```
COLUMNS:  '----------'('DELIVERY|DATE')
```

Special Options Related to Column Headings

The following table summarizes the options that affect the column headings. Use an OPTIONS statement to specify these options (page 555).

OPTIONS RELATED TO COLUMN HEADINGS

OPTION	DESCRIPTION
COLHDGONCE	Suppresses titles and prints the column headings just once at the beginning of the report. Report will have no page breaks.
HEADINGSEP('/') **HDGSEP('/')**	Specifies the column heading separator character to be used in the COLUMNS statement. This character indicates where a column heading text should be split onto a new line. The default column heading separator character is the vertical bar.
HGCOLHDG	Specifies that "Harvard Graphics style" column headings are wanted. When specified, only a single line will be used for the column headings. This is useful when creating PC files whose first record should contain a "legend" for each of the data columns.
MULTICOLHDG	Specifies that column headings *are* wanted for a report that contains multiple COLUMNS statements. (The default is to suppress column headings when more than one COLUMNS statement is specified.) When MULTICOLHDG is specified, column headings will be generated for the items in the *first* COLUMNS statement.
NOCOLHDGS	Suppresses all column headings. Does not affect titles. Report will have normal page break processing.
NOOVERPRINT	Specifies that no "overprinting" is wanted. When specified, the underscore line for the column headings will be single spaced rather than overprinted.

OPTIONS RELATED TO COLUMN HEADINGS (CONTINUED)	
OPTION	**DESCRIPTION**
NOTITLES	Suppresses all titles and column headings. Report will have no page breaks.
NOUNDERSCORES	Suppresses the underscore line that normally prints "under" the column headings. This option may be useful when creating reports that will be viewed online.
TITLEONCE	Prints titles (and any column headings) just once at the beginning of the report. Report will have no page breaks.

How to Change the Width of a Column

This section explains:

- how Spectrum Writer determines the **default width** of a column
- how to **specify your own column width**

Most of the features discussed in this section are illustrated in the sample report in **Figure 37**.

Spectrum Writer considers several factors when deciding what size to make each column, including:

- the number of characters in a character field (or literal)
- how many digits will likely be needed to display numeric fields, including the Grand Total value at the end of the report
- the width of the column heading

Based on these considerations, Spectrum Writer chooses a default width for each column. You may need to change this width in some cases. Do this by enclosing a numeric **width parm** in parentheses immediately after the field name. (Do *not* leave a space between the field name and the first parenthesis.)

For example, there may be too much data in a report to fit on the page. In this case, you might use a width parm to **shorten** some of the larger character fields. The following example shortens the LAST–NAME field to only 10 characters:

```
COLUMNS: LAST-NAME(10)
```

Of course, any last names containing more than 10 characters will be truncated in the report column.

> **Note:** *Numeric* columns are never truncated. Doing so might result in misleading figures appearing in the report. Instead, if a column is too small to display all significant digits (or a minus sign) for a numeric field, the column will be filled with a "size" error indicator (which looks like this: *****S*****). **Figure 37** shows an example of this.

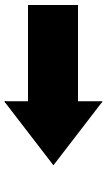
When shortening columns, it is possible to specify a column width that is shorter than the *column headings*. In this case, the column headings will also be truncated. Therefore, when specifying a shorter column width you may also need to specify new column headings. The new column headings should be broken into parts small enough to fit within the specified column width. Here is an example of a COLUMNS statement which specifies a column width of only 3, and also specifies column headings that are only 3 characters long:

```
COLUMNS: LAST-NAME(3,'LST NAM')
```

As mentioned above, you may occasionally see a "size" error indicator (****S****) in a numeric column. This means that the column wasn't wide enough to display all the digits in the number. Sometimes, a column will be wide enough to display the numeric value in the regular report lines, but will not be big enough to display the Grand Total value at the end of the report. In these cases you need to **widen** the column to provide enough room to

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE LISTING'
COLUMNS: EMPL-NUM(3)
         LAST-NAME
         LAST-NAME(10)
         LAST-NAME(3,'LST|NAM')
         HIRE-DATE
         HIRE-DATE(5)
         TOTAL-SALES(22)
         TOTAL-SALES(8)
```



Produce this Report:

```
                                EMPLOYEE LISTING

EMP      LAST          LAST     LST   HIRE    HIRE          TOTAL           TOTAL
NUM      NAME          NAME     NAM   DATE    DATE          SALES           SALES

036 JONES           JONES      JON 01/31/80 01/31              42,509.89 ****S***
037 JOHNSON         JOHNSON    JOH 06/21/75 06/21              86,999.24 ****S***
039 JOHNSON         JOHNSON    JOH 11/25/79 11/25              75,023.55 ****S***
040 MACDONALD       MACDONALD  MAC 07/04/82 07/04               2,560.98 2,560.98
041 SIMPSON         SIMPSON    SIM 12/01/82 12/01               8,723.88 8,723.88
042 MORRISON        MORRISON   MOR 11/30/79 11/30              98,054.99 ****S***
043 CHRISTOPHERSON  CHRISTOPHE CHR 08/15/81 08/15              47,665.31 ****S***
044 BAKER           BAKER      BAK 06/04/82 06/04              92,125.89 ****S***
045 THOMAS          THOMAS     THO 06/04/82 06/04              60,193.49 ****S***


*** GRAND TOTAL (9 ITEMS)                                    513,857.22 ****S***
```

Remarks:

- the EMPL–NUM column is 3 bytes wide, causing the default column headings to be truncated
- the second LAST–NAME column has been shortened to 10 bytes
- the third LAST–NAME column is shortened to 3 bytes, and also specifies shortened column headings
- the second HIRE–DATE column has been shortened to 5 characters so that only the month and day appear
- the first TOTAL–SALES column has been widened to accommodate numbers into the hundreds of trillions
- the second TOTAL–SALES column has been shortened so much that "size" errors now occur for large values, resulting in the ****S*** size error indicator

Figure 37. Specifying the width of report columns

display the Grand Total value. For example, the following COLUMNS statement allows 22 characters for the TOTAL–SALES field:

```
COLUMNS: TOTAL-SALES(22)
```

Note that this does not mean that there will be room for 22 *digits* to print in the column. The 22 character width of the column will also includes such things as commas, a decimal point, and a minus sign, if necessary.

Another way to widen a numeric column is to use a large PICTURE as an override display format. (Display formats are discussed in the following section.) The following example also widens the TOTAL–SALES column to 22 characters, and has the advantage of making it easier to visualize how many digits that will accommodate:

```
COLUMNS: TOTAL-SALES(PIC'ZZZ,ZZZ,ZZZ,ZZZ,ZZ9.99')
```

How to Change the Way Dates, Times and Numbers Are Formatted

This section explains:

- what a **display format** is
- the **default display formats** used to display data
- how to specify **your own display format**

PC File Note: Display formats should not normally be used when creating PC files. Spectrum Writer chooses the display format needed to create an import file for the PC program specified in the OPTIONS statement. After importing your PC file into a PC spreadsheet, you can use the PC program's features to change the way dates or numbers are formatted.

Most of the features discussed in this section are illustrated in the sample report in **Figure 38**.

When formatting data in a report (especially dates, times and numbers), there are several decisions to make. For example, a date might be formatted in any of the following ways (to list just a few possibilities):

```
12/31/90
DECEMBER 31, 1990
31 DEC 90
```

Similarly, a numeric value might be formatted in any of these ways (and others):

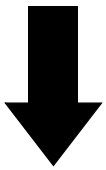
```
1,234
1,234.000
1234
0001234
$1,234
+1234
```

Time values can be formatted in the following ways, among others:

```
12:34:56
12:35
```

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE LISTING'
COLUMNS: EMPL-NUM
         LAST-NAME
         SOCIAL-SEC-NUM(PIC'999-99-9999')
         HIRE-DATE(LONG1)
         STATUS-BYTE(HEX)
         TOTAL-SALES(DOLLAR)
```



Produce this Report:

```
                          EMPLOYEE LISTING

                     SOCIAL
EMPL      LAST         SEC          HIRE        STATUS      TOTAL
NUM       NAME         NUM          DATE         BYTE       SALES

036  JONES           012-09-8765 JANUARY 31, 1980     C1        $42,509.89
037  JOHNSON         912-04-0334 JUNE 21, 1975        C1        $86,999.24
039  JOHNSON         004-77-9981 NOVEMBER 25, 1979    C1        $75,023.55
040  MACDONALD       889-79-0013 JULY 4, 1982         40         $2,560.98
041  SIMPSON         112-05-0456 DECEMBER 1, 1982     C1         $8,723.88
042  MORRISON        900-12-0556 NOVEMBER 30, 1979    C1        $98,054.99
043  CHRISTOPHERSON  415-09-0761 AUGUST 15, 1981      C1        $47,665.31
044  BAKER           878-19-0156 JUNE 4, 1982         C1        $92,125.89
045  THOMAS          776-83-8221 JUNE 4, 1982         C1        $60,193.49


*** GRAND TOTAL (9 ITEMS)                                      $513,857.22
```

Remarks:

- the SOCIAL–SEC–NUM column shows leading zeros, and has dashes in the appropriate places
- the HIRE–DATE columns shows the date in the LONG1 format, with the month name spelled out
- the STATUS–BYTE is shown in its hexadecimal representation
- the TOTAL–SALES column has a floating dollar sign
- the Grand Total line uses the same display format for TOTAL–SALES as the regular report lines

Figure 38. Customizing the way dates and numbers are formatted in a report

Spectrum Writer supports many different **display formats** that specify exactly how to format a field in a report. A complete list of these display formats is found in Appendix B, "Display Formats" on page 617.

If you do not specify a display format in the COLUMNS statement, Spectrum Writer will use the display format from:

- the FIELD or COMPUTE statement that defined the field
- an OPTIONS statement FORMAT parm
- the default display format shown in the table on page 618

To specify your own display format for a field, put a display format parm in parentheses immediately after the field name. (Do *not* leave a space between the field name and the first parenthesis.) Be sure to use a display format that is valid for the field's data type. (For example, you cannot request that a *numeric* field be displayed with a *date* display format.)

Here is an example of specifying display formats in the COLUMNS statement:

```
COLUMNS: LAST-NAME
         SOCIAL-SEC-NUM(PIC'999-99-9999')
         HIRE-DATE(LONG1)
         STATUS-BYTE(HEX)
         TOTAL-SALES(DOLLAR)
```

The previous statement specifies that:

- the SOCIAL–SEC–NUM field should be formatted with leading zeros *not* suppressed, and with dashes in the appropriate positions
- the HIRE–DATE field should be formatted with the month name completely spelled out
- the STATUS–BYTE field should be shown in it hexadecimal representation
- the TOTAL–SALES field should be formatted with a floating dollar sign.

Note: Many of the date display formats cause dates to be formatted with a delimiter. By default, the delimiter used is a slash. If you prefer a different delimiter, use the DATEDELIM option. For example:

```
OPTIONS: DATEDELIM('–')
```

The above statement causes a dash (—) to be used as the delimiter, rather than a slash (/), when formatting dates. Thus, if the above statement was used, a date formatted with the DD–MM–YY display format might look like this:

```
31-12-95
```

Note: Similarly, you can change the delimiter used to format time fields by using the TIMEDELIM option. For example:

```
OPTIONS: TIMEDELIM('.')
```

The above statement causes a dot (.) to be used as the delimiter, rather than a colon (:), when formatting times. Thus, if the above statement was used, a time formatted with the HH–MM display format might look like this:

```
12.30
```

Note: The same display format used in formatting data for the regular report lines is also used to format the data in the **total line**, and in any other **statistical lines** requested. This means, for example, that if you want to see an extra decimal digit for a column's average value (at a control break), you should specify a PICTURE that has the correct number of decimal digits in the COLUMNS statement. **Figure 42** (page 149) shows an example of this.

Note: You can also specify the BIZ ("blank if zero") parm along with a display format. That causes all non-zero data to be formatted according to the display format. However, whenever the value to be formatted is zero, the column will be left blank. You can use the BIZ parm with numeric, date and time fields. A date is considered to have a zero value if the month, day and last 2 digits of the year are all zeros (regardless of the value of the century part of the year).

Formatting Tips for International Users

This section suggests some options that international users may wish to use when creating reports

The following table lists a number of options of special interest to international users. The report in **Figure 39** (page 142) uses some of these options.

OPTIONS OF INTEREST TO INTERNATIONAL USERS		
OPTIONS STATEMENT PARM	**DESCRIPTION**	**EXAMPLE**
FORMAT(DD–MM–YY)	Makes DD–MM–YY the default date display format. All dates in the report will now appear as "DD/MM/YY" by default.	31/12/96
DATEDELIM('.') **DATEDELIM('–')**	Makes a dot (or dash) the standard delimiter used to format all dates in the report.	31.12.96 31–12–96
TIMEDELIM('.') **TIMEDELIM('–')**	Makes a dot (or dash) the standard delimiter used to format all times in the report.	12.34.56 12–34–56
FORMAT(DOTSEP)	Makes DOTSEP the default display format for all numeric fields in the report. A dot is used to separate thousands and millions, etc. A comma indicates where the decimal digits begin.	1.234.567,89

OPTIONS OF INTEREST TO INTERNATIONAL USERS (CONTINUED)		
OPTIONS STATEMENT PARM	**DESCRIPTION**	**EXAMPLE**
FORMAT(PIC'ZZZ ZZZ ZZ9.9')	Makes the default numeric display format the specified picture. Spaces are used to separate thousands, millions, etc.	`1 234 567.8`
FORMAT(PIC'ZZ ZZZ ZZ9V,9')	Makes the default numeric display format the specified picture. Spaces are used to separate thousands, millions, etc. A comma is used to separate the decimal digits.	`1 234 567,8`
PIC'ZZZ.ZZ9V,99 DM'	Use a PICTURE display format similar to this to print currency symbols (like DM) after a numeric value.	`123.456,78 DM`
DDMMYYLIT	Tells Spectrum Writer that all date literals in the control statements are in DD/MM/YY or DD/MM/YYYY format. Note: the slash is always used as the delimiter in date literals. The DATEDELIM option, if any, only changes the way dates are *formatted* in the output— not the way date literals are written in control statements.	`INCLUDEIF: SALES-DATE > 31/12/98 AND < 28/2/2001`

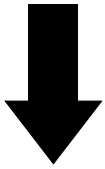
Of course, you can use any combination of the above options in a single OPTIONS statement:

```
OPTIONS:  FORMAT(DOTSEP,DD-MM-YY)  DATEDELIM('-')  TIMEDELIM('.')  DDMMYYLIT
```

If you would like to use some of these options as the default for *all reports* in your company, put the desired OPTIONS statement in a special member of your Spectrum Writer Copy Library. Then, under OS/390, use the SWOPTION DD to point to that member. Spectrum Writer will process the statements in that member before it processes the other control statements (page 424). Under VSE, use a COPY statement to copy that member at the beginning of your requests.

These Control Statements:

```
OPTIONS:   FORMAT(DOTSEP, DD-MM-YY)  DATEDELIM('.') DDMMYYLIT
INPUT:     EMPL-FILE
TITLE:     'INTERNATIONAL EMPLOYEE LISTING'
TITLE:     'HIRED AFTER 31 DECEMBER 1975'
INCLUDEIF: HIRE-DATE > 31/12/1975
COLUMNS:   EMPL-NUM
           LAST-NAME
           HIRE-DATE
           TOTAL-SALES
           TOTAL-SALES(PIC'ZZZ ZZ9V,99')
```



Produce this Report:

```
              INTERNATIONAL EMPLOYEE LISTING
               HIRED AFTER 31 DECEMBER 1975

EMPL      LAST          HIRE        TOTAL         TOTAL
NUM       NAME          DATE        SALES         SALES

036  JONES           31.01.80       42.509,89  42 509,89
039  JOHNSON         25.11.79       75.023,55  75 023,55
040  MACDONALD       04.07.82        2.560,98   2 560,98
041  SIMPSON         01.12.82        8.723,88   8 723,88
042  MORRISON        30.11.79       98.054,99  98 054,99
043  CHRISTOPHERSON  15.08.81       47.665,31  47 665,31
044  BAKER           04.06.82       92.125,89  92 125,89
045  THOMAS          04.06.82       60.193,49  60 193,49


*** GRAND TOTAL (8 ITEMS)          426.857,98 426 857,98
```

Remarks:

- the FORMAT option makes DOTSEP and DD–MM–YY the default numeric and date display formats for the report.
- the DATEDELIM('.') option causes all dates to be formatted using dots rather than slashes.
- the DDMMYYLIT options means that all date literals will be in DD/MM/YY (or DD/MM/YYYY) format. Note that slashes are still required in date literals.
- the INCLUDEIF statement uses a date literal in DD/MM/YY format to select records whose HIRE–DATE is after December 31, 1975
- the first TOTAL–SALES column uses the default display format (DOTSEP)
- the second TOTAL–SALES column uses an override PICTURE that has blanks as the separator character and a comma as the decimal character.

Figure 39. A report with international formatting options

How to Format Data as ASCII

Most PC's, network servers and mini-computers work with ASCII data, rather than the EBCDIC data used on mainframes. As a mainframe program, Spectrum Writer writes its reports out as EBCDIC data. The easiest way to convert Spectrum Writer's output from EBCDIC to ASCII is to simply let your file transfer program do it for you (as it downloads your file from the mainframe to your PC. Look for an "ASCII translation" option, or something similar. You'll probably also want to use a "CR/LF" option to append ASCII carriage return/line feeds to the end of each line.)

Unfortunately, this method doesn't work as well for *output files* as it does for reports. That's because output files often contain binary data mixed in with the character data. In such situations, you can use the ASCII parm to tell Spectrum Writer to format specific output columns in ASCII instead of EBCDIC. Then you can download your output file without the ASCII translation option, thus preserving the binary data in your records.

When the ASCII parm is specified for a column, Spectrum Writer first formats the column (in EBCDIC) in the normal way. That is, it uses the correct display format, it processes any BIZ parm, any NOREPEAT parm, and so on. Then, the final, formatted column is translated from EBCDIC to ASCII.

Here are some other points related to creating ASCII output files:

- the ASCII parm does not affect the column headings for a column. (Of course, when creating output files, column headings are normally suppressed.)
- the ASCII parm may only be specified for *fields* appearing in the COLUMNS statement (not for literals). To put an ASCII literal in your output, first use a COMPUTE statement to create a character field containing your literal:

  ```
  COMPUTE: ADDRESS-LIT = 'ADDRESS'
  COLUMNS: ADDRESS-LIT(ASCII) ADDR-LINE1(ASCII)
  ```

  Another way to specify an ASCII literal (especially very short ones like a single space) is to specify them in ASCII yourself, as hex literals:

  ```
  COMPUTE: ADDRESS-LIT = 'ADDRESS'
  COLUMNS: ADDRESS-LIT(ASCII) 0 X'20' 0 ADDR-LINE1(ASCII)
  ```

- use the COLSEP option if you want to separate the columns in your output file with an ASCII character (such as a space or a comma). For example, to put an ASCII space (hex '20') between the columns of a report, specify:

  ```
  OPTIONS: COLSEP(X'20')
  ```

- you may want to append ASCII CR/LF ("carriage return/line feed") codes (ASCII 0D0A) to the end of each output record:

  ```
  COLUMNS: NAME(ASCII) SALES-DATE(ASCII) SALES-TIME(ASCII) X'0D0A'
  ```

- the #ASCII built-in function (described in Appendix D, "Built-In Functions" on page 628) is another tool available for converting data from EBCDIC to ASCII.
- if desired, you can specify your own, custom EBCDIC-to-ASCII translation table by using the ASCIITABLE option (in an OPTIONS statement).

How to Blank Out Repeating Values

This section explains:

- how to **print blanks** in a column instead of a repeating value
- how a repeating value in the **first line of a new control group** is handled

Most of the features discussed in this section are illustrated in the sample report in **Figure 40**.

The NOREPEAT parm in a COLUMNS statement tells Spectrum Writer to blank out a column whenever it would contain the same value as in the previous line. However, the column's value is always shown (even if it is a repeated value) in two cases:

- in the first detail line of each **new page**
- in the first detail line of a **new control group** (that is, in the first detail line after a control break)

For example:

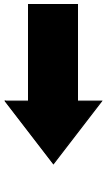
```
COLUMNS: LAST-NAME(NOREPEAT)
```

The above statement tell Spectrum Writer not to print repeating values of the LAST–NAME field.

If you prefer to also blank out repeating values in the first line of each control group, use the NOREPEATPAGE parm instead of NOREPEAT. That parm causes repeat values to be blanked out everywhere except in the first detail line of each new page.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'LIST OF SALES BY REGION'
SORT:    REGION EMPL-NAME
COLUMNS: REGION(NOREPEAT)
         EMPL-NAME(NOREPEAT)
         EMPL-NAME
         SALES-DATE
         CUSTOMER
         AMOUNT
         TAX
```



Produce this Report:

LIST OF SALES BY REGION

REGION	EMPL NAME	EMPL NAME	SALES DATE	CUSTOMER	AMOUNT	TAX
EAST	MORRISON	MORRISON	03/30/95	A1 PHOTOGRAPHY	29.65	1.78
		MORRISON	03/29/95	STAR MARKET	44.35	2.66
	SIMPSON	SIMPSON	04/30/95	J & S LUMBER	23.87	1.43
		SIMPSON	04/01/95	EUROPEAN DELI	14.99	0.90
NORTH	JOHNSON	JOHNSON	04/05/95	MARYS ANTIQUES	9.98	0.60
		JOHNSON	04/01/95	VILLA HOTEL	234.45	14.07
	JONES	JONES	04/15/95	EZ GROCERY	10.25	0.62
		JONES	04/15/95	TOY TOWN	10.25	0.62
		JONES	04/15/95	TOY TOWN	121.76	7.31
SOUTH	JOHNSON	JOHNSON	04/16/95	ACME BUILDING	500.00	30.00
		JOHNSON	03/12/95	ACE ELECTRICAL	101.38	6.09
WEST	BAKER	BAKER	03/26/95	JACKS CAFE	137.00	8.22
		BAKER	04/12/95	JACKS CAFE	135.75	8.15
	THOMAS	THOMAS	04/14/95	YOGURT CITY	9.98	0.60
*** GRAND TOTAL (14 ITEMS)					1,383.66	83.05

Remarks:

- the NOREPEAT parm for REGION and EMPL–NAME causes repeated values in those columns to be blanked out
- the second EMPL–NAME column does not use the NOREPEAT parm, for comparison

Figure 40. A report that blanks out repeating values

How to Change the Justification of Data within a Column

This section explains:

- how data is **normally justified** within a column
- how to specify that the data within a column should be **left–, center–, or right–justified**

Most of the features discussed in this section are illustrated in the sample report in **Figure 41**.

By default, Spectrum Writer justifies fields in the following manner:

TYPE OF DATA	DEFAULT JUSTIFICATION
Character	None
Numeric	Right–justified
Date	None
Time	Right–justified
Bit	None

To change the way data is justified within a column, simply specify a justification parm (LEFT, CENTER, or RIGHT) in parentheses immediately after the field name. (Do *not* leave a space between the field name and the first parenthesis.)

For example, the following statement specifies that the LAST–NAME field should be right–justified, the FIRST–NAME field should be center–justified, and the TOTAL–SALES field should be left–justified.

```
COLUMNS: LAST-NAME(RIGHT)  FIRST-NAME(CENTER)  TOTAL-SALES(LEFT)
```

Note: The default justification of "None" (see table above) is *not* the same as left-justification. Any leading blanks in a character field, for example, are left as is when there is no justification. When left-justification is specified, all leading blanks are removed.

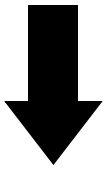
Note: You may also abbreviate LEFT, CENTER and RIGHT as LJ, CJ and RJ, respectively.

Note: The maximum width allowed for columns that are to be justified is 256 characters.

Note: The use of a large column heading or a large width parm can result in a report column that is bigger than the area actually needed to display the contents of character, date and bit fields. In such cases, the field's actual (smaller) display area is centered within the larger area reserved for the entire column. Justification, if any, is performed only within this smaller, centered area where the field’s contents actually appear.

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE LISTING'
COLUMNS: EMPL-NUM
         LAST-NAME(RIGHT)
         FIRST-NAME(CENTER)
         TOTAL-SALES(LEFT)
```



Produce this Report:

```
                  EMPLOYEE LISTING

EMPL      LAST            FIRST           TOTAL
NUM       NAME            NAME            SALES

036            JONES      JERRY      42,509.89
037          JOHNSON     THOMAS      86,999.24
039          JOHNSON      LINDA      75,023.55
040        MACDONALD     RICHARD     2,560.98
041          SIMPSON     TIMOTHY     8,723.88
042         MORRISON     MICHAEL     98,054.99
043   CHRISTOPHERSON     MELISSA     47,665.31
044            BAKER     VIVIAN      92,125.89
045           THOMAS     MARTIN      60,193.49


*** GRAND TOTAL (9 ITEMS)           513,857.22
```

Remarks:

- the EMPL–NUM column has no justification parm
- the LAST–NAME column is right–justified
- the FIRST–NAME column is center–justified
- the TOTAL–SALES column is left justified
- the Grand Total line uses the same justification for TOTAL–SALES as the regular report lines

Figure 41. Specifying how to justify data within the report columns

How to Specify Which Columns to Total

This section explains:

- how Spectrum Writer determines **which columns to print totals (and other statistics) for**
- **how to explicitly specify** that a column should or should not be included in total and statistics lines
- how to print **totals for time fields**

Most of the features discussed in this section are illustrated in the sample report in **Figure 42**.

There are a number of **statistical lines** that can be printed at the end of a report, as well as at control breaks. The total line is the most common statistical line. By default, a total line automatically prints at the end of the report (the "Grand Totals") and at each control break. The other statistical lines are:

- the average line
- the non–zero average line
- the maximum line
- the minimum line
- the non–zero minimum line

These other statistical lines do not print unless specifically requested (in either a SORT or a BREAK statement).

For a column to appear in any of the statistical lines, Spectrum Writer must **accumulate** information about it as the report is being produced. For example, it must accumulate the column's total value, its average value, etc. Each field that is accumulated automatically appears in *all* statistical lines printed.

Which fields are accumulated? By default, all **numeric columns** are accumulated. So, by default, all numeric columns appear in the total line, and any of the other statistical lines that are printed.

The one exception to this rule is numeric fields that are displayed using a PICTURE which contains **special characters**. (Special characters include such things as parentheses, imbedded dashes, asterisks, etc.) By default, numeric fields displayed with such a PICTURE are not accumulated and therefore do not appear in the total line and other statistical lines. To illustrate this exception, consider the following COLUMNS statement:

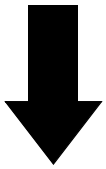
```
COLUMNS: TELEPHONE(PIC'(999) 999-9999')
```

The telephone number column in this report would not be accumulated, even though TELEPHONE is defined as a numeric field (see Appendix F, "Files Used in Examples" on page 648). The special characters in the PICTURE (namely the parentheses) suggest that totals, averages, etc. would not be appropriate for this field.

To state Spectrum Writer's default more precisely: all numeric columns *except those formatted with special characters* are accumulated and appear in the statistical lines of the report.

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE LISTING'
COLUMNS: EMPL-NUM
         LAST-NAME
         TELEPHONE(PIC'(999) 999-9999')
         TOTAL-SALES
         TOTAL-SALES(NOACCUM)
         NUM-ACCOUNTS(PIC'Z,ZZ9.9')
BREAK:  #GRAND AVERAGE
```



Produce this Report:

```
                              EMPLOYEE LISTING

EMPL      LAST                            TOTAL           TOTAL        NUM
NUM       NAME          TELEPHONE         SALES           SALES      ACCOUNTS

036  JONES           (415) 555-7653       42,509.89       42,509.89     78.0
037  JOHNSON         (602) 555-6654       86,999.24       86,999.24    128.0
039  JOHNSON         (415) 555-6785       75,023.55       75,023.55    104.0
040  MACDONALD       (415) 555-9887        2,560.98        2,560.98      6.0
041  SIMPSON         (818) 555-1887        8,723.88        8,723.88     16.0
042  MORRISON        (818) 555-4748       98,054.99       98,054.99    154.0
043  CHRISTOPHERSON  (602) 555-4556       47,665.31       47,665.31     65.0
044  BAKER           (415) 555-1209       92,125.89       92,125.89    147.0
045  THOMAS          (415) 555-1152       60,193.49       60,193.49    118.0


*** GRAND TOTAL (9 ITEMS)               513,857.22                    816.0
*** AVERAGE VALUE                        57,095.25                     90.7
```

Remarks:

- the TELEPHONE field is not accumulated by default, since its PICTURE includes special characters
- the first TOTAL–SALES column is accumulated by default, and appears in the total and average lines
- the second TOTAL–SALES is not accumulated (due to the NOACCUM parm) and does not appear in the total or average lines
- the NUM–ACCOUNTS column is displayed with a PICTURE that includes one decimal digit, so that the average line will also contain one decimal digit for that column
- the BREAK: #GRAND statement specifies that averages should print along with the Grand Totals at the end of the report

Figure 42. Specifying which columns to total

You may, however, override this default and explicitly state whether any numeric field is to be accumulated or not. Take as an example the DEPT–NUM field, which is defined as a numeric field (see Appendix F, "Files Used in Examples" on page 648). By default, the DEPT–NUM column would be accumulated since it is a numeric field. Yet, it makes no sense to total or to average the department number. In the case of this field you want to specify that the DEPT–NUM field should not be accumulated.

This is normally done when a field is first defined— in either a FIELD or a COMPUTE statement. Specifying the **NOACCUM parm** in those statements indicates that the field should not be accumulated. By specifying this parm when a field is first defined, you avoid having to specify NOACCUM in the COLUMNS statement of every report that uses that field. Here is how the DEPT–NUM field was defined so that it is not accumulated (and therefore does not appear in totals lines):

```
FIELD: DEPT-NUM  LENGTH(1)  TYPE(NUM)  NOACCUM
```

A similar parm is available in the COMPUTE statement to specify that a computed field should not be accumulated:

```
COMPUTE: NEW-DEPT-NUM(NOACCUM) = 900 + DEPT-NUM
```

There is also a similar **ACCUM parm** that can be specified when a field is defined. This parm explicitly specifies that a numeric field *should be* accumulated and appear in the total (and statistical) lines. Use this parm if you do wish to total a field that is formatted with special characters.

You may also explicitly state whether or not to accumulate a particular numeric field directly in the COLUMNS statement. Use the ACCUM or NOACCUM parm in parenthesis immediately after the field name. Such a parm in the COLUMNS statement overrides (for the current report only) any other parm that may have been specified in the FIELD or COMPUTE statement. For example:

```
COLUMNS: TOTAL-SALES(ACCUM)  DEPT-NUM(NOACCUM)
```

In the above example, the total sales column would be accumulated, and the department number field would not be accumulated, regardless of what was specified in their FIELD statements. Therefore, the TOTAL–SALES columns would appear in the total and other statistical lines. And the DEPT–NUM field would not appear in any of the statistical lines.

By default, Spectrum Writer does not total any **time fields**. However, if you have a time field which is a duration or interval (as opposed to a time of day), you may want to total it in your report. You can do this by specifying the ACCUM parm for your time field. For example:

```
COLUMNS: TIME-ON-PHONE(ACCUM)
```

The above statement would cause the TIME–ON–PHONE field to be totalled at the Grand Total line and at control breaks. It makes sense to total this time field, since it represents a duration (time spent on the telephone) rather than a time of day.

Note: To suppress the entire Grand Total line, use the NOGRANDTOTAL parm on the OPTIONS statement. For information on customizing the Grand Totals, see page 207.

Note: To suppress the entire total line at a control break, see page 185.

Note: The same display format used in formatting data for the regular report lines is also used to format the data in the total line, and in any other statistical lines requested. This means, for example, that if you want to see two decimal digits for a particular field in the average line, you should also specify that two decimal digits print in the regular report column. Do this by specifying a PICTURE that has two decimal digits in the COLUMNS statement. An example of this (but using only one decimal digit) is shown in **Figure 42** (page 149). (For more information on specifying PICTURES, see page 451.)

How to Produce Multi–Line Reports

This section explains:

- how to **print more than one report line** for each input file record
- how to write **more than one output record** to a PC file for each input file record

PC File Note: The following discussion of multi–line reports also applies to creating PC files. With reports, each COLUMNS statement results in one print line being printed in the report. With PC files, each COLUMNS statement results in one output record being written to the PC file.

Most of the techniques discussed in this section are illustrated in **Figure 43**.

All of our report examples until now have used a single COLUMNS statement. However, you are allowed to specify as many COLUMNS statements for a report as you like. Each COLUMNS statement results in one print line in the body of the report. Thus, a report with a single COLUMNS statement will produce a report having a single line for each record included in the report. A report with three COLUMNS statements will print three lines for each input record, and so on. The report lines will print in the same order that the COLUMNS statements appear in.

Note: To print a variable number of lines per input record, see page 249.

Reports with multiple COLUMNS statements are useful when you need to display a large amount of data from each record. They are also useful when a single record has several related fields that you want to print stacked on top of each other, rather than listed alongside each other.

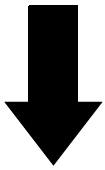
The following tips will help your multi–line reports look better.

Aligning Columns in Multi-Line Reports

To align the columns from the different COLUMNS statements neatly, you may need to use explicit **spacing factors and width parms**. (Spacing factors are discussed on page 128; width parms are discussed on page 135.) Consider the sample report in **Figure 43**. The first field listed on each COLUMNS statement is not the same size. If the spacing factors had not been used after the LAST–NAME, ADDRESS, CITY, and STATE field names, the subsequent columns on each line (the literal text and the quarterly sales figures) would have been skewed. The spacing factors compensated for the first columns' different widths and caused the subsequent columns to line up correctly.

These Control Statements:

```
OPTIONS: DOUBLE
INPUT:   EMPL-FILE

TITLE:   'EMPLOYEE ADDRESSES, WITH QUARTERLY SALES'
TITLE:
TITLE:   '       ADDRESS          QUARTER        SALES     ' /
TITLE:   '_____________________ _____________ _____________' /

COLUMNS: LAST-NAME 6 '1ST QUARTER:'  SALES-QTR1
COLUMNS: ADDRESS   1 '2ND QUARTER:'  SALES-QTR2
COLUMNS: CITY      6 '3RD QUARTER:'  SALES-QTR3
COLUMNS: STATE(2) 19 '4TH QUARTER:'  SALES-QTR4
```



Produce this Report:

```
        EMPLOYEE ADDRESSES, WITH QUARTERLY SALES

       ADDRESS          QUARTER          SALES

JONES                1ST QUARTER:          9,956.01
125 MAIN STREET      2ND QUARTER:         10,511.56
SAN FRANCISCO        3RD QUARTER:          8,698.07
CA                   4TH QUARTER:         13,334.25

JOHNSON              1ST QUARTER:         21,560.15
4000 LINDA VISTA     2ND QUARTER:         21,350.21
SCOTTSDALE           3RD QUARTER:         19,970.10
AZ                   4TH QUARTER:         24,118.78

JOHNSON              1ST QUARTER:         14,590.34
12 LINCOLN DRIVE     2ND QUARTER:         17,220.10
SANTA ROSA           3RD QUARTER:         20,100.08
CA                   4TH QUARTER:         23,113.12

                                       (other report lines not shown)

*** GRAND TOTAL (9 ITEMS)             122,989.16
                                      140,583.32
                                      124,677.23
                                      125,597.60
```

Remarks:

- the DOUBLE option is used to print a blank line between each input record's data
- a spacing factor is used before the second item in each COLUMNS statement, to force correct alignment of subsequent columns
- a width parm is used to make the STATE "column" only 2 bytes wide. Otherwise, its larger default column heading ("STATE") would have resulted in a 5–byte column.
- the use of multiple COLUMNS statements suppresses the printing of the default column headings
- the second TITLE statement puts a blank line between the real report title and the title line used to make column headings
- the third and fourth TITLE statements have a trailing slash, to left–align the column heading text
- the last TITLE statement is "overprinted," since it contains only underscores and spaces

Figure 43. Using multiple COLUMN statements to print multi-line reports

Use the **DOUBLE parm** of the OPTIONS statement (page 573) to double space the report after all the report lines for a particular input record have printed. Otherwise, it will be hard to tell which report lines are related to each other. The DOUBLE option tells Spectrum Writer to double space before printing a new *record's* data. It does not mean to double space *within* the report lines for the same input record. (To do that, use empty COLUMNS statements wherever you want a blank line to appear.)

Column Headings in Multi-Line Reports

Another thing to remember about reports with multiple COLUMNS statements: **column headings** are *not* automatically generated. To print column headings in a multi–line report, you have two options:

- use the MULTICOLHDG parm in an OPTIONS statement
- use TITLE statements to create your own column headings

Let's examine each of these options. The MULTICOLHDG option tells Spectrum Writer to create column headings as it normally would for the *first* COLUMNS statement. If those column headings would be appropriate for your report, this is the easiest method to use. Of course, you can also use column heading parms in that first COLUMNS statement to specify exactly the column headings you want.

If the column headings from the first COLUMNS statement would not be appropriate, you can use the second method to create column headings in a multi–line report. Use additional TITLE statements to supply your own headings (see **Figure 43**). After the regular TITLE statements, add a blank TITLE to cause a blank line to print. Then use one or more TITLE statements to specify your column headings.

To prevent these titles from being centered (and therefore not lining up correctly with the report columns) use a trailing slash. The trailing slash causes these title lines to be left–aligned, rather than centered (page 168).

If you want to **underline** your columns headings, use a final TITLE statement that contains nothing but underscores and blanks. Spectrum Writer will "overprint" any title line that contains only blanks and underscore characters.

You can also use literal texts within the COLUMNS statements as a sort of **row heading**, which works in conjunction with the more generalized column heading. (An example of a row heading is the literal text "1ST QUARTER" in the report in **Figure 43**.) Together, the row and column headings make clear exactly what each item of data in the report is.

Notice that the Grand total lines do *not* contain these literal texts ("1ST QUARTER", etc.) This is because only numeric columns appear in the Grand totals. To add such texts to the Grand Total lines, you could use several BREAK statement FOOTING parms, as discussed in the section beginning on page 207.

> **Tip:** By using a large of number of COLUMNS statements, you can create "reports" where each input record prints one entire page. Use this technique to print special forms. Specify one COLUMNS statement per line of the form, mixing literal text and field names as desired. Use empty COLUMNS statements where blank lines should appear. Use enough trailing blank COLUMNS statements to fill out the page.

How to Change the Report Margins

This section explains:

- how to increase the **left margin** in a report
- how to increase the **top margin** in a report
- how to change the **bottom margin** in a report

To shift the whole report (including titles, body, Grand Totals, etc.) to the right, use the LEFTMARGIN parm of the OPTIONS statement (discussed on page 565). For example:

```
OPTIONS: LEFTMARGIN(10)
```

The above statement would create a left margin of 10 blank spaces.

The first title in a report is always printed at the "top of form" position. (The exact location of the "top of form" line depends on the printer you are using.) Putting the first title on the "top of form" line at your shop may result in the titles printing too high on the page. To solve this problem, simply use one or more *blank* TITLE statements before the normal ones. This has the effect of increasing your report's **top margin**. The first few titles (which will still start printing at the "top of form" line) will only be blank lines. The following statements would cause the report title to print three lines down from where it would normally print:

```
TITLE:
TITLE:
TITLE:
TITLE: 'EMPLOYEE DIRECTORY'
```

Use the PAGELEN option (in the OPTIONS statement) to adjust the report's **bottom margin**. The PAGELEN value tells Spectrum Writer how many lines of each page to use when printing the report. The bottom margin of the report is simply the unused lines at the bottom of each sheet of paper.

The default PAGELEN value is 60. That means that 60 lines are used on each page. Specifying a smaller PAGELEN will **increase** the bottom margin in the report. Specifying a larger value will **decrease** the bottom margin. For example, the following statement will cause 5 additional blank lines to be left at the bottom of each page:

```
OPTIONS: PAGELEN(55)
```

How to Print Bar Graphs

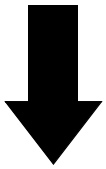
In "How to Change the Way Dates, Times and Numbers Are Formatted" (page 137), we learned how to specify a **display format** along with a field name in the COLUMNS statement. The display format specifies just how a field's data should be formatted in a report. One of the display formats you can use for numeric fields is called BARGRAPH (or just BAR). It specifies that the field should be formatted as a horizontal bar graph (or "histogram.") For example:

```
COLUMNS:  EMPL-NAME  CUSTOMER  AMOUNT(BARGRAPH)
```

The above statement specifies that the AMOUNT field should appear as a bar graph in the report. By default, bar graph columns are 20 characters wide. The column will contain a

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'BAR GRAPH OF FIRST QUARTER SALES'
COMPUTE: SALES-IN-THOUSANDS(0) = SALES-QTR1 / 1000
SORT:    SALES-QTR1(DESC)
COLUMNS: LAST-NAME  FIRST-NAME  SALES-QTR1
         SALES-IN-THOUSANDS  SALES-IN-THOUSANDS(BAR,30)
```



Produce this Report:

```
                          BAR GRAPH OF FIRST QUARTER SALES

                                                    SALES                SALES
     LAST            FIRST           SALES          IN                    IN
     NAME            NAME            QTR1        THOUSANDS             THOUSANDS

MORRISON         MICHAEL              25,014.19         25 *************************
JOHNSON          THOMAS               21,560.15         22 **********************
BAKER            VIVIAN               21,336.10         21 *********************
THOMAS           MARTIN               14,889.07         15 ***************
JOHNSON          LINDA                14,590.34         15 ***************
CHRISTOPHERSON   MELISSA              13,807.22         14 **************
JONES            JERRY                 9,956.01         10 **********
SIMPSON          TIMOTHY               1,287.58          1 *
MACDONALD        RICHARD                 548.50          1 *


*** GRAND TOTAL (9 ITEMS)            122,989.16        124
```

Remarks:

- the BAR display format (in the COLUMNS statement) causes the second SALES–IN–THOUSANDS column to be displayed as a bar graph
- the override column width of 30 causes the bar graph column to be 30 characters wide
- a COMPUTE statement is used to create a field whose value is between 0 and 30, to correspond with the width of the bar graph column
- the (0) parm in the COMPUTE statement results in SALES–IN–THOUSANDS having zero decimal digits

Figure 44. A report with a bar graph column

number of asterisks equal to the rounded value of the numeric field (up to a maximum of 20). For example, when the AMOUNT field is equal to 5.25, the column will contain 5 asterisks: when the AMOUNT field is equal to 17.89, the column will contain 18 asterisks.

Of course many fields will have values much larger than 20. The TOTAL–SALES field, for example, contains values into the tens of thousands. Use a COMPUTE statement to reduce large fields down to a value between 0 and 20. Then display that COMPUTE field using the BAR display format. This is illustrated in **Figure 44**

Also, you may use an override column width parm to increase (or decrease) the default column width of 20 characters. The report on page 155 shows a bar graph column that is 30 characters wide. (The use of the width parm was discussed beginning on page 135.)

How to Print Vertical Lines between Report Columns

Spectrum Writer normally leaves one blank space between each report column. You can use the COLSEP parm of the OPTIONS statement to specify some other "column separator" text. For example:

```
OPTIONS: COLSEP(' | ')
```

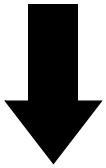
The above statement specifies a 3–character text that should appear between each column of the report. The text consists of a blank, a vertical bar character, and another blank. Using this OPTIONS statement results in a report with a vertical bar running down between the report columns. This gives the report a spreadsheet–like appearance.

The report in **Figure 45** shows a report that uses the above statement.

> **Note:** The vertical bar is the "Shift 1" key on most mainframe terminals. When working at a PC running terminal emulation software, you will probably not see a key with this symbol on it. Some terminal emulator programs use the "pipeline" key as the vertical bar key. Some others use the right–hand square bracket key "]" for this purpose.
>
> **PC File Note:** The COLSEP parm should not be used when creating PC files. Spectrum Writer will choose an appropriate column delimiter for your PC program.

These Control Statements:

```
OPTIONS: COLSEP(' | ')
INPUT:   EMPL-FILE
TITLE:   'DEMONSTRATION OF VERTICAL BARS BETWEEN COLUMNS'
COLUMNS: EMPL-NUM  LAST-NAME  FIRST-NAME  DEPT-NUM
         SEX       HIRE-DATE  TOTAL-SALES
```



Produce this Report:

```
                    DEMONSTRATION OF VERTICAL BARS BETWEEN COLUMNS

EMPL        LAST             FIRST          DEPT              HIRE          TOTAL
NUM         NAME             NAME           NUM     SEX       DATE          SALES

036   | JONES           | JERRY          |       2  | M  | 01/31/80  |      42,509.89
037   | JOHNSON         | THOMAS         |       1  | M  | 06/21/75  |      86,999.24
039   | JOHNSON         | LINDA          |       2  | F  | 11/25/79  |      75,023.55
040   | MACDONALD       | RICHARD        |       2  | M  | 07/04/82  |       2,560.98
041   | SIMPSON         | TIMOTHY        |       3  | M  | 12/01/82  |       8,723.88
042   | MORRISON        | MICHAEL        |       3  | M  | 11/30/79  |      98,054.99
043   | CHRISTOPHERSON  | MELISSA        |       1  | F  | 08/15/81  |      47,665.31
044   | BAKER           | VIVIAN         |       4  | F  | 06/04/82  |      92,125.89
045   | THOMAS          | MARTIN         |       4  | M  | 06/04/82  |      60,193.49


*** GRAND TOTAL (9 ITEMS)                                                 513,857.22
```

Remarks:

- the COLSEP option specifies a 3–character "column separator" text, consisting of a vertical bar surrounded by blanks

Figure 45. A report with vertical lines separating the columns

Including All Fields in the COLUMNS Statement

How can you get all of the fields from an input record into the COLUMNS statement without having to type each field name individually? Just put the *record name* in the COLUMNS statement. (An input's record name is, by default, the same as the file name.)

```
Example: OPTION:  PC
         INPUT:   SALES-FILE
         COLUMNS: SALES-FILE
```

The three statements above reformats the entire contents of the SALES-FILE into a comma-delimited "PC" file. **Figure 22** (page 89) shows a run that uses the above statements.

Record Name Parms

There are a number of optional parms you can specify when you use a record name in the COLUMNS statement. Use these parms to:

- specify how overlapping fields should be handled
- specify individual fields to exclude from the output
- specify what order the columns (fields) should appear in

Here is the full syntax of the record name option of the COLUMNS statement:

```
COLUMNS: record-name[( [exclude-field1 exclude-field2 ...]
                       [OUTER/INNER] [BYDEF/BYNAME/BYCOL] [LIST/NOLIST] )] ...
```

Excluding Duplicate Data from Overlapping Fields

Specifying a record name in the COLUMNS statement is a quick way to get all of the data from an input record into your output file. But in most cases there will be some fields that you don't really need or want in your output.

One common example of this are fields that overlap with other fields in the input record. For example, consider this definition of a date field:

```
Example: FIELD: SALES-DATE TYPE(YYMMDD)
         FIELD: SALES-YY   LEN(2) COLUMN(SALES-DATE)
         FIELD: SALES-MM   LEN(2)
         FIELD: SALES-DD   LEN(2)
```

The SALES-DATE field defines the whole 6-byte date field in the record. Then, the next three fields *redefine* the individual YY, MM and DD components of the same field.

By default, Spectrum Writer writes *all* of the fields defined for a file to the output. That means it will write all four of the above fields, even though it is a duplication of the same data.

If that is not what you want, specify either the OUTER or INNER parm.The **OUTER parm** tells Spectrum Writer to exclude outer fields when overlaps occur.

```
Example: COLUMNS: SALES-FILE(OUTER)
```

The above statement would result in SALES-YY, SALES-MM and SALES-DD being written to the output file, but not SALES-DATE.

The **INNER parm** does just the opposite. It excludes inner fields when overlaps are detected. In the above example, it would result in only SALES-DATE being written to the output file.

Excluding Individual Fields

There are times when you can not get the exact results you want with either the INNER or OUTER parm. This may happen when there are multiple levels of overlapping fields, or partially overlapping fields. Or when you want to exclude certain fields for some other reason. In such cases, you can name individual fields as **"exclude fields."** Any field name specified in the parms for a record name will *not* be included in the output. You can specify as many exclude fields as you like, in any order.

```
Example: COLUMNS: SALES-FILE(BACKUP-EMPL-NUM  COMMISSION-RATE  TIME-ON-PHONE)
```

The above statement would write out all fields from the SALES-FILE except for the three fields named in the parms as exclude fields.

> **Note:** if you have actual fields named INNER, OUTER or any of the other parm names, Spectrum Writer assumes you are naming the *field* by that name (as an exclude field), rather than naming the parm. If you have fields with the same name as a parm, you can indicate that you mean the *parm* by preceding it with a pound sign (#). For example:

```
Example: COLUMNS: SALES-FILE(#INNER #BYNAME)
```

Specifying the Field Order in the Output

By default, the fields appear in the output record in the order in which they were defined (the default BYDEF option). Specify the BYNAME parm if you want the fields to appear in the alphabetical order of the field names. Or specify BYCOL to put them in starting column order (that is, the order in which they occur in the input record).

```
Example: COLUMNS: SALES-FILE(BYCOL)
```

Listing the Fields in the Control Listing

By default, Spectrum Writer lists the names of all of the fields it is automatically including in the output (the default LIST option). This might be a long list for some files. If you prefer to suppress the list of field names in the control listing, use the NOLIST parm:

```
Example: COLUMNS: SALES-FILE(NOLIST)
```

Including All COMPUTE Fields

COMPUTE fields that are part of a file definition (that is, that are kept in the copy library along with the FIELD statements for a file) are generally included with the rest of the fields when a record name is specified in the COLUMNS statement.

But there are some COMPUTE fields that are not considered to be part of any file's definition. Examples of such COMPUTE fields are those which are computed without using any fields as operands (that is, they use only literals) and those which are defined out of sequence for an earlier file (while a different file is the "current" file being defined).

Spectrum Writer has a special "record name" called #COMPUTES that includes all COMPUTE fields that are not a part of a file definition. Use this special record name to output all COMPUTE fields that are not part of any file's definition. The syntax is:

```
Example: COLUMNS: #COMPUTES[( [exclude-field1 ...] [BYDEF/BYNAME] [LIST/NOLIST] )] ...
```

Note that the INNER, OUTER and BYCOL parms do not apply to this special record name.

What If You Run Out of Room?

The standard size of a report line is 132 characters. Therefore, the print expressions you specify (in COLUMNS statements, TITLE statements, etc.) must produce a line no longer than 132 characters. If it exceeds 132 characters, Spectrum Writer will truncate part of the line. If you have trouble fitting all the information you need into a report, try some of the following solutions:

If you are printing on a **laser printer:**

- try using a condensed font (or "form") that allows more than 132 characters per line. Also change the JCL (under OS/390) to specify a larger LRECL for the SWOUTPUT DD (page 417). Spectrum Writer will then allow your report to be as wide as the LRECL value that you specify. It will not be limited to 132 characters in that case.

  VSE Note: Increase the RECSIZE value in the OUTATTR parm and in the JCL to achieve the same result (page 431).

  Note: You may need to send a "setup string" to your laser printer at the beginning of the report in order to use the desired printer form. See the PRTSETUP option (page 572) for information on doing this.

If you are printing on a regular line printer:

- shorten long column headings, by abbreviating them, or by breaking the heading up into several lines (see **Figure 36** on page 131). See "How to Change the Column Headings" on page 130.
- shorten the width of one or more columns. See "How to Change the Width of a Column" on page 135.
- use smaller spacing factors between the report columns
- move constant information (information that does not change from page to page) *out* of the individual report lines and *into* the title lines or break lines. For an example of putting data in the title, see **Figure 54** (page 179).
- use multiple COLUMNS statements to create a report with more than one report line for each input file record. See "How to Produce Multi–Line Reports" on page 151.

Why Do I See ****X**** in My Report?

This section explains:

- why **asterisks** sometimes appear in your report

Sometimes an error prevents Spectrum Writer from being able to display the desired data in a report. Rather than abandon the whole report, Spectrum Writer prints a number of asterisks where that data should have appeared. A single letter will be imbedded in the

asterisks. That letter is an **error code** which tells you exactly what kind of error occurred. The following table lists these error codes. Appendix E, "Error Indicators" (page 644) discusses each of these errors in more detail, including suggestions for correcting the error. A discussion on propagating error conditions is also found in that Appendix.

Error Code	Meaning
****A****	Ambiguous reference to a field.
****E****	Error in field's definition.
****F****	Error computing a field's offset value.
****I****	Invalid data in the field.
****S****	Size error (not enough room to print all the digits).
****U****	Undefined field.
****V****	Overflow occurred.
****Z****	Divide by zero occurred.

Customizing the Report Titles

The following sections show various ways that you can customize the titles in a report. The following sections explain:

- how to include **file data** in a title (below)
- how to put the **page number, date and time** in your titles (page 165)
- how to change the **spacing and formatting** of data in the titles (page 165)
- how to split the title into **left aligned, centered, and right aligned** parts (page 168)
- various **special options** that relate to titles and column headings (page 175)

How to Include Data from a File in the Title

This section explains:

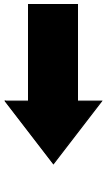
- how to print **literal texts** in a title
- how to print **data from an input file** in a title

The contents of the TITLE statement is simply a **print expression**. Print expressions tell Spectrum Writer how to build one print line that will be used in a report. The print expression in a TITLE statement specifies how to build a title line.

Note: the contents of the COLUMNS statement is also a print expression— one that tells how to build the report lines for the main body of the report. Thus, the contents

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE DIRECTORY -'  LAST-NAME
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
                    EMPLOYEE DIRECTORY - BAKER

     LAST            FIRST        HIRE
     NAME            NAME         DATE      ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the value used for LAST–NAME in the title is taken from the next report line to print
- by default, the literal text is separated from the LAST–NAME field by one blank
- by default, the title is centered over the report

Figure 46. A report title that includes data from a file

of a TITLE statement is very similar to the contents of a COLUMNS statement, which you are already familiar with.

As with other print expressions in Spectrum Writer, just list one or more **items** to print.

```
TITLE: item1 item2 item3 ...
```

Each **item** can be either a **literal text** or a **field name**.

To put a **literal** text in the title, simply enclose the text in either apostrophes or quotation marks. For example, the following statement causes the words EMPLOYEE DIRECTORY to appear in the title:

```
TITLE: 'EMPLOYEE DIRECTORY'
```

To put **data from an input file** in your title, simply list the desired field name. (Do *not* put the field name in apostrophes or quotation marks.) For example, the following statement causes the contents of the LAST–NAME field to appear in the report title.

```
TITLE: LAST-NAME
```

The data that appears in the title will be taken from the first record whose data prints on the new page.

By the way, the TITLE statement can refer to *any field* from the input file(s). You are not limited to just those fields that are listed in the COLUMNS statement. Field names used in the TITLE statement may be any of the following:

- any field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a computed field (created in a preceding COMPUTE statement)
- **a built–in** field. (See Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields.)

Figure 46 (page 162) shows an example of a title which uses one literal text and one data field from the input file. (Another example of printing data from a file in the title is shown in **Figure 54** on page 179.)

How to Include the Page Number, Date and Time in a Title

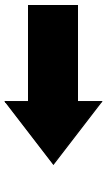
This section explains:

- how to include **data from built–in fields** in a title

Most reports will include the page number and the current date and time somewhere in the title. Spectrum Writer has a number of **built–in fields** that can be used for this purpose.

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE DIRECTORY'
TITLE:   #DAYNAME  #TODAY  #TIME
TITLE:   'PAGE'  #PAGENUM
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
                          EMPLOYEE DIRECTORY
                       FRIDAY    04/27/92  2:35 PM
                                 PAGE    1

     LAST           FIRST        HIRE
     NAME           NAME         DATE      ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  PHOENIX
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the #DAYNAME built–in field causes the day of the week to appear in the title
- the #TODAY built–in field causes the current date to appear in the title
- the #TIME built–in field causes the current time to appear in the title
- the #PAGENUM built–in field causes the page number to appear in the title

Figure 47. A title that shows the current day of the week, date, time and page number

You may use these fields in your TITLE statement just like real fields from input files. The built–in fields available are:

BUILT-IN FIELDS AVAILABLE IN THE TITLE STATEMENT	
BUILT-IN FIELD NAME	**DESCRIPTION**
#PAGENUM	a numeric field containing the current page number. (May also be abbreviated #PAGE.)
#TODAY	a date field containing the system date on which the program began execution
#COMDATE	(*VSE only*) a date field containing the date from the DATE JCL statement, if any
#DAYNAME	a character field containing the day of the week (Monday, etc.) on which the program began execution
#TIME	a character field containing the formatted time of day at which the program began execution (formatted in 12–hour format including AM or PM)
#TIME24	a character field containing the formatted time of day at which the program began execution (in 24–hour format)
#HHMMSS	a time field containing the time of day on which the program began execution
#JOBNAME	an 8–byte character field containing the jobname of the job executing Spectrum Writer

The sample report in **Figure 47** shows report titles that use several of these built–in fields.

The techniques discussed in the following sections of this chapter can be used to improve the appearance of the current date in your title. For example, you may want to spell out the name of the month in the current date. You may also want to align the date and page number with the left or right report margin.

> **Note:** These built–in fields can also be used in the FOOTNOTE statement. Use the FOOTNOTE statement when you want to print the date, page number, etc. at the *bottom of* your report pages. (See page 175.)

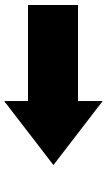
How to Change the Appearance of Items in the Title

This section explains how to:

- specify the **number of spaces** that should appear between items in a title
- specify the **width** of an item in the title
- specify the **display format** to use when formatting dates, times and numbers in the title

These Control Statements:

```
INPUT:    EMPL-FILE
TITLE:    'EMPLOYEE DIRECTORY -'  LAST-NAME(1)
TITLE:    #TODAY(LONG1)
TITLE:    #TODAY(CENTER,LONG1)
SORT:     LAST-NAME  FIRST-NAME
COLUMNS:  LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
                          EMPLOYEE DIRECTORY - B
                            JUNE 4, 1990
                               JUNE 4, 1990

     LAST           FIRST        HIRE
     NAME           NAME         DATE      ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the width of the LAST–NAME field in the first title has been shortened to 1 byte
- the LONG1 display format causes the current date (#TODAY) to be spelled out in the second and third titles
- the CENTER justification parm causes the current date to be correctly centered in the third title line

Figure 48. Using width, display format and justification parms in the title

- **justify** the contents of fields printed in the title
- specify that data should be **ASCII** instead of EBCDIC

As in other print expressions, you may customize the title line by using optional **spacing factors** and **parms**. So, the full syntax for the TITLE statement is this:

```
TITLE:  [n] item1(parms) [n] item2(parms) [n] item3(parms) ...
```

The optional **spacing factor** [n] is the number of blank spaces to leave between items in a title. If you omit the spacing factor, the default is for *one* blank space to appear between each item. (A spacing factor of zero is allowed if you want *no* spaces to appear between two items in a title.) For example, the following statement causes 5 blanks to appear between the literal text "EMPLOYEE DIRECTORY" and the contents of the LAST–NAME field in the title:

```
TITLE:  'EMPLOYEE DIRECTORY'  5  LAST-NAME
```

The optional **parms** are used to provide details about how to display data fields in a title. You may specify one or more parms, enclosed in parentheses, immediately following a field name. (Do *not* leave a space between the field name and the first parenthesis.) You may use any combination of parms, in any order. Separate the parms with a comma and/or blanks. For example, the following statement has both a width parm and a justification parm for the LAST–NAME field:

```
TITLE:  LAST-NAME(50, CENTER)
```

The following table shows the parms that are available in the TITLE statement. The sample report in **Figure 48** (page 166) illustrates the use of many of these parms.

TITLE STATEMENT PARMS	
PARM	**DESCRIPTION**

ASCII	Specifies that the field should be formatted in ASCII, rather than in EBCDIC
	
	`COMPUTE: TITLE-LIT = 'DATE: '`
	`TITLE: TITLE-LIT(ASCII) 0 SALES-DATE(ASCII)`
	
	See page 143 for more information on creating ASCII output files.

BIZ	Means "blank if zero." Specifies that the title area should be left blank whenever the numeric, date or time item contains zeros. The following example specifies that the SALES-DATE field should be left blank whenever its value is zero. `TITLE:  'DATE:'  SALES-DATE(BIZ)`
display-format	Specifies how to format a field in the title. A complete list of display formats is found in Appendix B, "Display Formats" on page 617. This parm works just like the display format parm in the COLUMNS statement, which is explained in more detail beginning on page 137. The following example specifies that the current date field (#TODAY) should be displayed in the LONG1 format — with the month name spelled out: `TITLE:  #TODAY(LONG1)`

TITLE STATEMENT PARMS (CONTINUED)	
PARM	**DESCRIPTION**
LEFT/ CENTER/ RIGHT	Specifies how to justify a field's data within the area reserved for it in the title. These parms work just like the justification parms in the COLUMNS statement, which are explained in more detail beginning on page 146. The section titled "How to Split the Title into Left Aligned, Centered, and Right Aligned Parts" (page 168) also illustrates the use of justification parms. The following example specifies that the contents of the current date field (#TODAY) should be center justified (as well as being formatted in the LONG1 display format). `TITLE: #TODAY(CENTER,LONG1)`
width	This numeric parm specifies how many characters should be reserved for an item in the title. This parm works just like the width parm in the COLUMNS statement, which is explained in more detail beginning on page 135. As an example, the following statement specifies that only one character of the LAST–NAME field should appear in the title: `TITLE: LAST-NAME(1)`

If a field is specified in a TITLE statement without any parms, Spectrum Writer chooses a default width, display format and justification.

Notice in the sample report in **Figure 48** that the #TODAY field in the second title line does not appear to be exactly centered over the report. This is because the contents of the #TODAY field does not fill the whole area reserved for it in the title. The default width reserved for a date in the LONG1 format is 18 characters — big enough to handle the largest possible value (for example "SEPTEMBER 31, 1999"). When a smaller value (for example "MAY 1, 1999") appears in this 18–character area with no justification, it is padded with blanks on the right. Therefore the date does not look like it is centered.

In other words, the *18–character area* reserved to display the #TODAY field is centered over the report. But, the *value within* the 18–character area is not centered. To correct this, a justification parm of CENTER was specified for the #TODAY field in the third title line of that report. The CENTER justification parm causes the contents of the 18–character #TODAY field to be centered.

To solve a similar problem that can arise when dates are lined up over the *right margin* of a report, see page 173.

How to Split the Title into Left Aligned, Centered, and Right Aligned Parts

This section explains:

- how to split the title into **left aligned, centered, and right aligned parts**

How to Split the Title into Left Aligned, Centered, and Right Aligned Parts

Until now, all of our TITLE statements have consisted of a single print expression. The contents of that print expression has been centered over the reports.

A TITLE statement is actually allowed to have up to *three* print expressions, separated with slashes (/).

```
TITLE:  print-expression1  [/ print-expression2]  [/ print-expression3]
```

> **Note:** Do not confuse multiple *items* within a single print expression with multiple *print expressions*. A single print expression may contain as many items (literal texts and field names) as you like. A new print expression begins only when a slash is encountered. See "How to Include Data from a File in the Title" on page 161 for a review of what a print expression is.

Each print expression is called a **title part**. Spectrum Writer aligns each title part differently, depending on how many parts there are. Here is how title parts are aligned:

Number of Title Parts	Alignment
1	the title is centered
2	the first part is left aligned, and the second part is right aligned
3	the first part is left aligned, the second part is centered, and the third part is right aligned

Thus, a simple TITLE statement with no slashes (and therefore with just a single part) will result in a title that is centered across the report. The sample reports in the preceding pages show examples of titles with only a single part.

A TITLE statement with two parts (separated by a slash) results in a title that has a left aligned part and a right aligned part. The report in **Figure 49** shows an example of such a title.

And a TITLE statement with three parts results in a title with: a left aligned part, a centered part, and a right aligned part. The report in **Figure 50** shows an example of a title that has three parts.

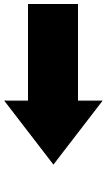
What if you want your whole title to be left aligned or right aligned, without splitting it into multiple parts? Use a leading or a trailing slash. This has the effect of creating a TITLE statement with two parts, but with one of the parts being an *empty* print expression. Since the TITLE statement has two parts, one will be left aligned and one will be right aligned. But the part that has no print expression will be all blank.

For example, a **trailing slash** causes a title to be left aligned. **Figure 51** (page 172) shows an example of this.

This use of a trailing slash to prevent the centering of a single title part is also helpful when creating column headings with the TITLE statement. An example of this appears in **Figure 43** (page 152).

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE DIRECTORY -'  LAST-NAME  / 'ABC COMPANY'
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
EMPLOYEE DIRECTORY - BAKER                                        ABC COMPANY

     LAST            FIRST        HIRE
     NAME            NAME         DATE       ADDRESS            CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

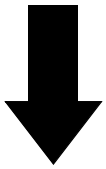
Remarks:

- the slash in the TITLE statement splits the title into left and right parts

Figure 49. A report with left and right title parts

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'ABC COMPANY'  /
         'EMPLOYEE DIRECTORY -'  LAST-NAME  /
         'SALES DEPARTMENT'
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
ABC COMPANY         EMPLOYEE DIRECTORY - BAKER              SALES DEPARTMENT

     LAST            FIRST        HIRE
     NAME            NAME         DATE       ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

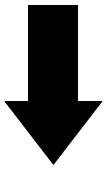
Remarks:

- the two slashes in the TITLE statement split the title into three parts
- the first title part is aligned with left margin of the report
- the second title part is centered
- the third title part is aligned with the right margin of the report

Figure 50. A report with left, center, and right title parts

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'DATE:'  #TODAY    /  'EMPLOYEE DIRECTORY'  /
TITLE:   'TIME:'  #TIME24   /
TITLE:   'PAGE:'  #PAGENUM  /
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
DATE: 04/27/92              EMPLOYEE DIRECTORY
TIME: 14:35
PAGE:    1

     LAST            FIRST        HIRE
     NAME            NAME         DATE       ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the built–in fields #TODAY, #TIME24, and #PAGENUM are included in the title
- using #TIME24 results in a 24–hour time, without the AM or PM
- the use of a trailing slash in the first title produces a left aligned and a centered title part
- the use of a trailing slash in the second and third titles produces a left aligned title

Figure 51. Titles with the date, 24-hour time, and page number on the left side of the report

You can also use a trailing slash in conjunction with a spacing factor to print a title in a certain column. For example, to print the text "REGION" in column 62 of the title, you would use this statement:

```
TITLE: 61 'REGION'  /
```

The above statement specifies that 61 blanks should be left before the first item in the title. Therefore, the word "REGION" would begin in column 62. The trailing slash prevents Spectrum Writer from trying to center the title.

On the other hand, you can use a **leading slash** to force the whole title to be aligned on the *right side* of the report. **Figure 52** (page 174) shows an example of this.

The reports on page 172 and page 174 also illustrate one other possibility. By using an empty print expression in the appropriate place, you can also create titles that have a left and a center aligned part, but no right aligned part. Or, you can create a title with a center and a right aligned part, but with no left aligned part.

Correcting Right Alignment Problems

You may sometimes specify a right aligned title only to find that the last character in the title does not line up with the last character of the body of the report. Two things can cause this to occur:

- the body of the report may be smaller than the total length of the title parts. By necessity the title will extend beyond the right margin of the report.
- the last field listed in the title may not have completely filled the area reserved for it. Thus, there would be trailing blanks within the last field in the title, and the title would not appear to be right aligned. In other words, while the end of the *field* lined up with the right edge of the report, the *data within* the field did not extend to its last character. You should right–justify the contents of the last field by specifying the RIGHT parm for that field. This will make the last characters in the title line up with the right edge of the report. **Figure 52** (page 174) shows a sample report that uses this technique to correctly right align the current date in a title.

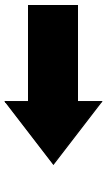
Correcting Centering Problems

A similar problem can occur with centered title parts. Sometimes they do not appear to be centered correctly. Two things can cause this to occur:

- this can happen when the *contents* of a centered field does not completely fill the area reserved for it in the title. In that case, the *field* may be centered correctly, but the *data within* the field may not be centered. Use the CENTER parm to center the contents of the field. The second title line in the report in **Figure 48** (page 166) exhibits this problem. The third title line in that same report uses the CENTER parm to correct the problem.
- sometimes correctly centering a title part would cause it to *overlap* with long title parts that are aligned over the left or right margins. In these cases, Spectrum Writer shifts the center title part to prevent the titles parts from overlapping.

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   /  'EMPLOYEE DIRECTORY'  /  #TODAY(LONG1,RIGHT)
TITLE:   /  #TIME
TITLE:   /  'PAGE:'  #PAGENUM(2)
SORT:    LAST-NAME  FIRST-NAME
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE  ADDRESS  CITY
```



Produce this Report:

```
                          EMPLOYEE DIRECTORY               APRIL 27, 1992
                                                                  2:35 PM
                                                                PAGE:  1

     LAST            FIRST        HIRE
     NAME            NAME         DATE       ADDRESS          CITY

BAKER           VIVIAN          06/04/82 667 CRESTHAVEN BLVD  WALNUT CREEK
CHRISTOPHERSON  MELISSA         08/15/81 61752 TIMBERIDGE RD  TORRANCE
JOHNSON         LINDA           11/25/79 12 LINCOLN DRIVE     SANTA ROSA
JOHNSON         THOMAS          06/21/75 4000 LINDA VISTA     SCOTTSDALE
JONES           JERRY           01/31/80 125 MAIN STREET      SAN FRANCISCO
MACDONALD       RICHARD         07/04/82 525 FOOTHILL DRIVE   PLEASANTON
MORRISON        MICHAEL         11/30/79 98 SOUTH LAKESIDE DR GLENDALE
SIMPSON         TIMOTHY         12/01/82 89876 WEST 53 STREET ARCADIA
THOMAS          MARTIN          06/04/82 77812 S. HUNTINGTON  CONCORD


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the built–in fields #TODAY, #TIME, and #PAGENUM are displayed in the titles
- the system date field (#TODAY) is displayed using the LONG1 format, and is right–justified
- the page number field (#PAGENUM) is only 2 characters wide
- the use of a leading slash in the first title produces a centered and a right aligned title part
- the use of a leading slash in the second and third titles produces a right aligned title

Figure 52. A title with date (spelled out), time, and page number on the right side of report

Special Options Related to Titles

The following table summarizes the options that affect the titles. (For a similar list of options that affect column headings, see page 133) Use an OPTIONS statement to specify these options (page 555).

OPTIONS RELATED TO TITLES	
OPTION	**DESCRIPTION**
COLHDGONCE	Suppresses titles. Prints the column headings just once at the beginning of the report. Report will have no page breaks.
NOCOLHDGS	Suppresses column headings, but does not affect titles. Report will have normal page break processing.
NOTITLES	Suppresses all titles and column headings. Report will have no page breaks.
TITLEONCE	Prints titles (and any column headings) just once at the beginning of the report. Report will have no page breaks.

How to Print "Titles" at the Bottom of Each Page

To print "titles" at the *bottom of* each page of the report, use the FOOTNOTE statement. The FOOTNOTE statement works just like the TITLE statement, except that the footnote lines print at the bottom of each page, rather than at the top. For example:

```
FOOTNOTE: 'THE INFORMATION IN THIS REPORT IS CONFIDENTIAL'
FOOTNOTE: 'PAGE' #PAGENUM
```

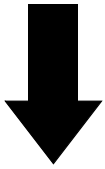
The two FOOTNOTE statements above cause two lines to print at the bottom of each page of the report. The first footnote line contains the literal text ("THE INFORMATION IN THIS REPORT IS CONFIDENTIAL") centered under the report. The second footnote line has the word "PAGE", followed by the page number. **Figure 53** shows a sample report which uses these two FOOTNOTE statements. FOOTNOTE statements may appear anywhere after the INPUT statement.

All of the features allowed in TITLE statements are also allowed in FOOTNOTE statements. (Using the TITLE statement is discussed beginning on page 161.) Specifically, you can:

- include the current date, time, page number, etc. in the footnote, by using the built–in fields #TODAY, #DAYNAME, #TIME, #TIME24, #HHMMSS and #PAGENUM. (page 163)
- separate the footnote line into left, center, and right aligned parts, by using slashes within the FOOTNOTE statement. (page 168)
- include data from the input file(s) in your footnote line. Just list the desired field name in the FOOTNOTE statement. The data that will appear in the footnote will be the field's value from the *previous* report record. (page 161)
- specify exactly how data should be formatted in the footnote, by using the width, display–format, and justification parms. (page 165)

These Control Statements:

```
INPUT:    EMPL-FILE
TITLE:    'ABC COMPANY -- EMPLOYEE DIRECTORY'
SORT:     LAST-NAME  FIRST-NAME
COLUMNS:  LAST-NAME  FIRST-NAME  EMPL-NUM  SEX  DEPT-NUM
          HIRE-DATE  CITY  STATE
FOOTNOTE: 'THE INFORMATION IN THIS REPORT IS CONFIDENTIAL'
FOOTNOTE: 'PAGE'  #PAGE
```



Produce this Report:

```
                    ABC COMPANY -- EMPLOYEE DIRECTORY

     LAST           FIRST      EMPL     DEPT   HIRE
     NAME           NAME       NUM  SEX NUM    DATE      CITY          STATE

BAKER           VIVIAN          044   F   4   06/04/82 WALNUT CREEK     CA
CHRISTOPHERSON  MELISSA         043   F   1   08/15/81 PHOENIX          AZ
JOHNSON         LINDA           039   F   2   11/25/79 SANTA ROSA       CA
JOHNSON         THOMAS          037   M   1   06/21/75 SCOTTSDALE       AZ
JONES           JERRY           036   M   2   01/31/80 SAN FRANCISCO    CA
MACDONALD       RICHARD         040   M   2   07/04/82 PLEASANTON       CA
MORRISON        MICHAEL         042   M   3   11/30/79 GLENDALE         CA
SIMPSON         TIMOTHY         041   M   3   12/01/82 ARCADIA          CA
THOMAS          MARTIN          045   M   4   06/04/82 CONCORD          CA


*** GRAND TOTAL (9 ITEMS)









              THE INFORMATION IN THIS REPORT IS CONFIDENTIAL
                                PAGE    1
```

Remarks:

- the report has two footnote lines that correspond to the two FOOTNOTE statements
- since no slashes are used, each footnote is centered under the report

Figure 53. Using the FOOTNOTE statement to add footnotes to a report

Customizing the Control Breaks

This section discusses:

- using the **SORT statement** to request a control break
- using the **BREAK statement** to request a control break
- some of the parms available for **customizing control breaks**

The easiest way to request a control break is to specify a break parm after a field name right in the SORT statement. For example, the TOTAL parm in the following SORT statement requests that a control break occur whenever the REGION field changes value:

```
SORT:  REGION(TOTAL)
```

At a control break, the following things happen by default:

- a **total line prints**, showing the number of items in the control group, as well as the totals for all numeric columns in the report
- **two blank lines print**, before continuing with the report

Another way to request a control break is to use the BREAK statement. The BREAK statement names a sort field and makes that field a control break field. Only a field named in an earlier SORT statement can appear in a BREAK statement. For example, the following two statements have the same effect as the above SORT statement.

```
SORT:  REGION
BREAK: REGION
```

We could also have included the TOTAL parm on the BREAK statement. However, since TOTAL is the default, it was not necessary.

There are several advantages to using a BREAK statement. The BREAK statement has parms that gives you complete control over what prints at control breaks. These parms are discussed in the sections that follow:

- how to specify the **report spacing** at a control break with the SPACE parm (page 178)
- how the **default total line** looks, and tips on getting the most out of it (page 180)
- how to print **control-group-wide percentages and ratios** in the total line (page 202)
- how to **customize the total line** using the TOTAL parm (page 182)
- how to **suppress totals** at a control break (page 185)
- how to **print statistical lines** using the AVERAGE, MAXIMUM, MINIMUM, NZAVERAGE and NZMINIMUM parms (page 186)
- how to print **customized "footing" lines** at the end of a control group using the FOOTING parm (page 188)
- how to print the **number of items contained** in a control group (page 198)

- how to print **customized "heading" lines** at the beginning of a control group using the HEADING parm (page 200)

How to Change the Control Break Spacing

This section explains:

- the **default control break** spacing in a report
- how to specify **your own control break spacing** in a report
- the **SPACE parm** in the BREAK statement

By default, Spectrum Writer prints two blank lines whenever a control break occurs. (These blank lines print *after* any footing lines, total lines and statistical lines for the control break have printed.) For example, the sample report in **Figure 13** (page 66) uses default spacing at control breaks.

If you want something other than two blank lines, specify a **spacing option** in either the SORT or the BREAK statement. (A complete list of spacing options is shown on page 180.) By coding the appropriate value for this parm, you can request that a different number of blank lines print (including zero lines), or you can request one of several types of "page breaks."

If you only want to customize the spacing of a control break, you do not need to use a BREAK statement. All break spacing options can be specified directly in the SORT statement. Simply put the spacing parm in parentheses immediately after the appropriate field name. For example, the following SORT statement requests that five blank lines print whenever the REGION field changes value:

```
SORT:  REGION(5)
```

The mere presence of the break spacing factor in the SORT statement above implies that REGION should be a control break field. The following SORT statement requests a **page break**. That is, whenever a new region starts printing, it will begin on a new page.

```
SORT:  REGION(PAGE)
```

In a BREAK statement, use the SPACE parm to specify the desired control break spacing. The following statements specify that 5 blank lines should print whenever the REGION field changes value:

```
SORT:  REGION
BREAK: REGION  SPACE(5)
```

And the following statements request a page break for the REGION field.

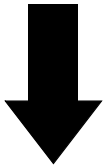
```
SORT:  REGION
BREAK: REGION  SPACE(PAGE)
```

Figure 54 shows a sample report that uses a similar BREAK statement to request a page break.

There are other spacing options that are especially useful for reports that are printed on the front and back of the paper. You may want to distribute the individual pages of your report to, for example, a company's various regions. To do this, the different regions must print

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES FOR REGION:'  REGION /  'PAGE'  #PAGENUM
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION SPACE(PAGE1)
```



Produce this Report:

```
SALES FOR REGION: EAST                                        PAGE    1

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/92 STAR MARKET            44.35       2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY         29.65       1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI          14.99       0.90
EAST   SIMPSON    04/30/92 J & S LUMBER           23.87       1.43
*** TOTAL FOR EAST  (4 ITEMS)                    112.86       6.77
```

```
SALES FOR REGION: NORTH                                       PAGE    1

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

NORTH  JOHNSON    04/01/92 VILLA HOTEL           234.45      14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES          9.98       0.60
NORTH  JONES      04/15/92 EZ GROCERY             10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN               10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN              121.76       7.31
*** TOTAL FOR NORTH (5 ITEMS)                    386.69      23.22
```

```
SALES FOR REGION: SOUTH                                       PAGE    1

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL        101.38       6.09

                        (other report lines not shown)
```

Remarks:

- specifying PAGE1 (in the BREAK statement) causes the report to skip to a new page whenever the REGION field changes value, and also resets the page number to 1
- since we printed the REGION in the title of each page, we could now eliminate the REGION column making room in the report for other data

Figure 54. A BREAK statement that requests a page break and resets the page number

on separate *sheets* of paper, not just on a new *page*. (A new page might only be the back side of the same sheet of paper where another region printed.) The NEWSHEET spacing option does this.

There are also spacing options that will reset the page number after a control break. When skipping to a new page after a control break, you may also want to start the page numbering over again with page one. This is especially useful when you will be distributing the various sections of the report to different people, and you want each section to start with page one. The PAGE1, NEWSHEET1 and ODDPAGE1 options do this.

The following table lists the control break spacing options available:

SPACING OPTIONS AVAILABLE AT CONTROL BREAK	
SPACING OPTION	**DESCRIPTION**
n	Skips this number of **blank lines**.
PAGE	Skips to the top of the **next page** of the report.
PAGE1	Works like PAGE, but also resets the page number to "one."
NEWSHEET	Skips to a **new sheet** of paper. In order for this feature to work, you must also use the OPTIONS statement's PRTSHEET parm to specify a character string that can be sent to your printer to tell it to skip to a new sheet of paper. (See page 572.)
NEWSHEET1	Works like NEWSHEET, but also resets the page number to "one."
ODDPAGE	Skips to the next **odd numbered** page. This parm accomplishes the same thing as the NEWSHEET parm, but can be used even if you do not have a character string to send to your printer to force it to skip to a new sheet. However, for this option to work you must ensure that the first page of your report prints on the front side of a sheet of paper. As long as page 1 of your report prints on the front side of a sheet of paper, all other odd numbered pages will also be on front sides.
ODDPAGE1	Works like ODDPAGE, but also resets the page number to "one."

PC File Note: Only the **n** spacing parm (meaning "n" blank lines) is allowed when creating PC files. Since PC files do not have "pages," the other spacing parms are meaningless for PC files.

How a Default Total Line Looks

This section explains:

- how the **default total line** looks
- **tips** on making the default total line look its best

Before we examine the various custom lines that we can print at a control break, let's look at what happens by default at a control break.

By default, Spectrum Writer prints one total line at every control break. The report in **Figure 54** (page 179) shows an example of the default total lines. They look something like this:

```
*** TOTAL FOR EAST  (    4 ITEMS)                112.86       6.79
```

Default total lines contain the following information:

- a number of **asterisks** (three, in this example) which serve to set the total line off from the regular report lines. The asterisks also serve as a visual indicator of the "level" of the break. The higher the break level, the more asterisks that print. (Break levels are discussed in "Reports with Multiple Control Breaks" on page 204.)
- the words TOTAL FOR, which identifies this as the *total* line
- the value of the **break field** in the control group that just ended (in this example EAST).
- the **number of items** that were included in the control group (in this example 4). The number of items is the number of primary input file records included in the control group. Usually, it is also the number of report lines printed for the control group.
- the control group **total** for each numeric column in the report (in this example the AMOUNT and TAX columns). (For more information on exactly which columns are totalled, see "How to Specify Which Columns to Total" on page 148.)

Split Total Lines

Sometimes the text at the beginning of the total line will extend into the area where the first column total should print. This normally happens when the first numeric column is fairly close to the left margin of the report. When the total line text would overlap with one or more actual column totals, Spectrum Writer uses **two lines** to print the totals. The first line contains the initial total line text (including the number of items). The second line then shows the actual totals for all of the numeric columns.

To prevent this splitting of the total line, design your reports so that the first numeric column is well away from the left margin of the report. You might do this by printing large character fields (such as names, descriptions, etc.) in the first columns of the report, and putting the numeric columns after that. That is what we have done for most examples in this manual. Or, you can use an initial spacing factor in the COLUMNS statement to shift all columns to the right, like this:

```
COLUMNS:  40  AMOUNT  TAX
```

The report in **Figure 65** (page 210) uses a COLUMNS statement with a large initial spacing factor.

To prevent splitting the total line, you could also specify a shorter text to being the total line with. Use the TOTAL parm to specify a shorter text (page 182).

Size Errors in Total Lines

When printing large reports you may see a number of asterisks in the total line. For example, you might see a total line that looks like this:

```
*** TOTAL FOR EAST  (    4 ITEMS)          ******S******       6.79
```

The "**size**" **error indicator** (*****S*****) indicates that there wasn't enough room to display all of the digits in a number. In this case, the report column is not wide enough to display the total value. Use a **width parm** in the COLUMNS statement to make the column wider (see page 135). For example, the following COLUMNS statement makes the AMOUNT column 20 characters wide, so that even huge numbers will fit in the total line:

```
COLUMNS: REGION  EMPL-NAME  SALES-DATE  CUSTOMER  AMOUNT(20)  TAX
```

If there is a very large number of records in a control group, there may not be enough room to print the *number of items* in the total line. In that case you might see something like this:

```
*** TOTAL FOR EAST  (**S** ITEMS)                 112.86       6.79
```

To correct this problem, specify your own total line text using the TOTAL parm (see page 182). Be sure to specify a width parm that leaves plenty of room to display the #ITEMS built–in field, like this:

```
BREAK: REGION
       TOTAL('*** TOTAL FOR'  REGION  #ITEMS(10)  'ITEMS')
```

The built–in field #ITEMS is discussed on page 198.

How to Customize the Total Line at a Control Break

This section explains:

- how to **customize** the total line at a control break
- how to use the **TOTAL parm** in the BREAK statement

Spectrum Writer automatically prints a total line at the end of each control group. As we saw earlier, the default total line begins with a text something like this:

```
*** TOTAL FOR EAST  (    4 ITEMS)
```

This text is then followed by the actual totals for each numeric column. You may prefer to print **your own text** at the beginning of the total line. Use the TOTAL parm of the BREAK statement to do that.

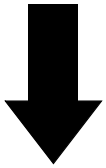
Here is an example of a BREAK statement with a TOTAL parm:

```
BREAK: REGION
       TOTAL('REGION TOTALS')
```

When you specify a text in a TOTAL parm, Spectrum Writer uses your text, rather than the default text, in its total line. The above statement specifies that the total line should begin with the words REGION TOTALS. After that, the actual totals appear, lined up under the appropriate report columns. **Figure 55** shows a sample report that uses the above BREAK statement.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES BY REGION'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION  TOTAL('REGION TOTALS')
```



Produce this Report:

```
                          SALES BY REGION

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

EAST      MORRISON 03/29/92 STAR MARKET         44.35      2.66
EAST      MORRISON 03/30/92 A1 PHOTOGRAPHY      29.65      1.78
EAST      SIMPSON  04/01/92 EUROPEAN DELI       14.99      0.90
EAST      SIMPSON  04/30/92 J & S LUMBER        23.87      1.43
REGION TOTALS                                  112.86      6.77


NORTH     JOHNSON  04/01/92 VILLA HOTEL        234.45     14.07
NORTH     JOHNSON  04/05/92 MARYS ANTIQUES       9.98      0.60
NORTH     JONES    04/15/92 EZ GROCERY          10.25      0.62
NORTH     JONES    04/15/92 TOY TOWN            10.25      0.62
NORTH     JONES    04/15/92 TOY TOWN           121.76      7.31
REGION TOTALS                                  386.69     23.22


SOUTH     JOHNSON  03/12/92 ACE ELECTRICAL     101.38      6.09
SOUTH     JOHNSON  04/16/92 ACME BUILDING      500.00     30.00
REGION TOTALS                                  601.38     36.09


WEST      BAKER    03/26/92 JACKS CAFE         137.00      8.22
WEST      BAKER    04/12/92 JACKS CAFE         135.75      8.15
WEST      THOMAS   04/14/92 YOGURT CITY          9.98      0.60
REGION TOTALS                                  282.73     16.97



****** GRAND TOTAL (14 ITEMS)                1,383.66     83.05
```

Remarks:

- the total line now begins with the text "REGION TOTALS" as specified in the TOTAL parm of the BREAK statement

Figure 55. A report with a customized total line at the control breaks

The contents of the TOTAL parm is actually a **print expression**. Print expressions tell Spectrum Writer how to build one print line to use in a report. In the TOTAL parm, the print expression tells how to build the first part of the total line.

> **Note:** The contents of the COLUMNS statement is also a print expression— one that tells how to build the report lines for the main body of the report. Thus, the contents of the TOTAL parm is very similar to the contents of a COLUMNS statement, which you are already familiar with.

Briefly, the TOTAL parm print expression can contain literal text, data from input records, data from built–in fields, and certain statistical values for numeric data fields. The section titled "How to Print Customized Footing Lines at a Control Break" (page 188) describes in detail how to write a FOOTING parm print expression. Those same rules apply to writing TOTAL parm print expressions.

Here is an example of a TOTAL print expression which consists of one literal item and one field name:

```
BREAK: REGION
       TOTAL('TOTALS FOR REGION:'  REGION)
```

The total line produced by the statement above would begin with:

```
TOTALS FOR REGION: xxxxx
```

where xxxxx would be the name of the region that had just finished printing.

You may also put a blank print expression in the TOTAL parm, like this:

```
BREAK: REGION TOTAL(' ')
```

The example above results in a total line with no beginning text— just the actual numeric totals themselves.

Only one TOTAL parm is allowed in the BREAK statement. If you need to print more than one line at a control break, use one or more FOOTING parms along with the TOTAL parm. (FOOTING parms are discussed beginning on page 188.) For example:

```
BREAK: REGION
       FOOTING('END OF REGION:' REGION)
       FOOTING('VERIFY THE FOLLOWING TOTALS WITH ACCOUNTING')
       TOTAL('TOTAL SALES')
```

The statement above would cause three lines to print at the control break: the two footing lines first, followed by the total line. The total line would begin with the text TOTAL SALES, followed by the numeric totals.

The total line at a control break always prints immediately after the last footing line (if any), regardless of where the TOTAL parm is specified in the BREAK statement.

If you want the total line to be *separated* from the footing lines, (or from the last detail report line) use a blank FOOTING parm, like this:

```
BREAK: REGION
       FOOTING('END OF REGION' REGION')
       FOOTING('VERIFY THE FOLLOWING TOTALS WITH ACCOUNTING')
       FOOTING(' ')
       TOTAL('TOTAL SALES AS OF' #TODAY)
```

This will cause a blank footing line to print after the first two footings and before the total line.

Notice in the above statement that we used the built–in field #TODAY to print the current date in the total line.

Note: To customize the Grand Totals line, see page 207.

How to Suppress the Total Line at a Control Break

This section explains:

- how to **suppress the total line** at a control break
- the **NOTOTAL parm** in the BREAK and SORT statements

Even when a report has no numeric columns, a total line still prints at control breaks. That is because the total line contains other useful information such as the value of the break field, and the number of items in the control group.

To suppress the total line at a control break, specify NOTOTAL in the SORT or BREAK statement. For example, if you did not want to see region totals at the REGION control break, you would write:

```
BREAK: REGION NOTOTAL
```

The above example would still result in a control break whenever the REGION field changed value. But region totals would *not* print at the break. Two blank lines (the default spacing option) is all that would print at the control break.

You can also use the NOTOTAL parm directly in the SORT statement, either alone or in combination with a break spacing parm. Here are two examples:

```
SORT: REGION(NOTOTAL)
SORT: REGION(PAGE, NOTOTAL)
```

The first example causes a control break to occur whenever the REGION field changes value, but prevents region totals from printing. (The presence of the NOTOTAL parm implies that a control break should occur.) The default spacing of two blank lines will be printed at the control break.

The second example above also causes a control break on the REGION field, but specifies that each new region should start printing on a new page. Again, no region totals would print at the control break.

Note: To suppress totals just for particular columns, see page 148.

Note: To suppress the Grand Total line, see the section beginning on page 207.

How to Customize the Statistical Lines at a Control Break

This sections explains:

- how to print **statistical lines** at a control break
- how these statistical lines look by **default**
- how to **customize** the statistical lines
- the **AVERAGE**, **MAXIMUM**, **MINIMUM**, **NZAVERAGE** and **NZMINIMUM** parms in the SORT and BREAK statements

The sample report in **Figure 56** illustrates most of the features discussed in this section.

There are a number of **statistical lines** that can be printed at a control break. The **total line** is the most common statistical line. By default, the total line automatically prints at each control break, as well as at the end of the report. The other statistical lines do not appear unless specifically requested. You may request them by specifying the appropriate parm in either the BREAK statement or the SORT statement. The statistical parms (and their abbreviations) are shown in the following table.

STATISTICAL PARMS AVAILABLE IN THE BREAK STATEMENT	
PARM	**STATISTIC LINE**
AVERAGE **AVG**	average line
NZAVERAGE **NZAVG**	non–zero average line. A non–zero average is the average obtained when zero values are excluded from the calculation. This value may be useful when the data in some records is missing.
MAXIMUM **MAX**	maximum line
MINIMUM **MIN**	minimum line
NZMINIMUM **NZMIN**	non–zero minimum line. A non–zero minimum is the minimum value, not considering zero values. This value may be useful when the data in some records is missing.

The following example requests that a line showing averages and a line showing maximum values be printed at the control break. (Of course, the total line will also print, since the NOTOTAL parm was not specified to suppress it.)

```
BREAK: REGION  AVERAGE  MAXIMUM
```

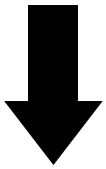
It is also possible to request the same thing directly in the SORT statement:

```
SORT: REGION(AVERAGE, MAXIMUM)
```

The presence of the statistical parms in the above SORT statement imply that REGION should be a break field.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES STATISTICS BY REGION'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION
         TOTAL('--- TOTAL SALES FOR REGION:' REGION)
         AVERAGE('--- AVERAGE SALE IN REGION')
         MAXIMUM('--- BIGGEST SALE IN REGION')
         MINIMUM('--- SMALLEST SALE IN REGION')
```



Produce this Report:

```
                    SALES STATISTICS BY REGION

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/92 STAR MARKET           44.35      2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY        29.65      1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI         14.99      0.90
EAST   SIMPSON    04/30/92 J & S LUMBER          23.87      1.43
--- TOTAL SALES FOR REGION: EAST                112.86      6.77
--- AVERAGE SALE IN REGION                        28.22      1.69
--- BIGGEST SALE IN REGION                        44.35      2.66
--- SMALLEST SALE IN REGION                       14.99      0.90


NORTH  JOHNSON    04/01/92 VILLA HOTEL          234.45     14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES         9.98      0.60
NORTH  JONES      04/15/92 EZ GROCERY            10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN              10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN             121.76      7.31
--- TOTAL SALES FOR REGION: NORTH               386.69     23.22
--- AVERAGE SALE IN REGION                       77.34      4.64
--- BIGGEST SALE IN REGION                      234.45     14.07
--- SMALLEST SALE IN REGION                       9.98      0.60

                  (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                1,383.66     83.05
****** AVERAGE VALUE                            98.83      5.93
****** MAXIMUM VALUE                           500.00     30.00
****** MINIMUM VALUE                             9.98      0.60
```

Remarks:

- the print expression in parentheses after each statistical parm determines the initial wording of the statistical lines
- to similarly customize the Grand Total statistical lines, we could add another BREAK statement (see page 207)

Figure 56. A report that prints statistical lines (average, maximum, minimum) at control breaks

When the average line prints at a control break, it begins with the text AVERAGE VALUE, followed by the averages themselves lined up under the numeric columns. Just as with the total line, you can change the beginning text to be anything you like. Simply specify a **print expression** in parentheses immediately after the AVERAGE parm:

```
BREAK: REGION  AVG('AVERAGES FOR REGION:' REGION)
```

The other statistical lines (maximum, minimum, etc.) begin with similar texts (MAXIMUM VALUE, MINIMUM VALUE, etc.) You can override the text for any of these lines in the same way as for total or average lines:

```
BREAK: REGION  MAXIMUM('BIGGEST SALE IN REGION:' REGION)
               MINIMUM('SMALLEST SALE IN REGION:' REGION)
```

As with the TOTAL parm discussed earlier, the contents of these additional statistical parms is simply a **print expression**. Briefly, the print expression can contain literal text, data from input records, data from built–in fields, and certain statistical values for numeric and time fields. The section titled "How to Print Customized Footing Lines at a Control Break" (page 188) describes in detail how to write a FOOTING parm print expression. Those same rules apply to writing print expressions for the statistical parms.

Any statistical lines requested at a control break will print *after* all footing lines have printed. The statistical lines always print in the following order:

- the total line
- the average line
- the non–zero average line
- the maximum line
- the minimum line
- the non–zero minimum line

Note: For information on *which columns* receive averages and other statistics, see page 148.

Note: Notice the statistical lines after the Grand Totals on page 187. They still begin with the default wording (****** AVERAGE VALUE, etc.) You can also customize these Grand Totals lines to match the statistics lines at the control breaks. (See page 207.)

How to Print Customized Footing Lines at a Control Break

This section explains:

- how to specify **customized "footing" lines** to print at the end of a control group
- the **detailed syntax for print expressions** used within the BREAK statement's FOOTING, TOTAL, AVERAGE, MAXIMUM, MINIMUM, NZAVERAGE and NZMINIMUM parms

PC File Note: This section discusses the FOOTING parm as it is used when creating reports. Some of this discussion does not apply to creating PC files. The use of the FOOTING parm for PC files is discussed on page 108.

Spectrum Writer automatically prints a total line at the end of each control group. You may want to print certain lines of your own at a control break (either in place of, or in addition to, the total line). Use the FOOTING parm of the BREAK statement to print such lines.

The FOOTING parm of the BREAK statement lets you specify a control break "footing line." This line prints just before the totals line (if any) at a control break. This line can contain literal text, data from input records, data from built–in fields, and certain statistical values for numeric and time fields.

Here is an example of a BREAK statement with a simple FOOTING parm:

```
BREAK: REGION
       FOOTING('END OF SALES IN REGION:'  REGION)
```

This FOOTING parm causes a line reading END OF SALES IN REGION: xxxxx to print immediately after the last report line in each region (where xxxxx is the name of the region). The report in **Figure 57** uses the above BREAK statement.

> **Note:** The following discussion of the BREAK statement's FOOTING parm syntax also applies to the TOTAL, AVERAGE, MAXIMUM, MINIMUM, NZAVERAGE, and NZMINIMUM parms (discussed in the sections beginning on page 182 and page 186). In addition, the syntax of the HEADING parm is almost identical— the only differences are explained in the section on the HEADING parm, beginning on page 200.

The contents of the FOOTING parm is simply a **print expression**. Print expressions tell Spectrum Writer how to build one print line to use in a report. In a FOOTING parm, the print expression tells how to build a line to print at a control break.

> **Note:** The contents of the COLUMNS statement is also a print expression— one that tells how to build the report lines for the main body of the report. Thus, the contents of the FOOTING parm is very similar to the contents of a COLUMNS statement, which you are already familiar with.

As with other print expressions in Spectrum Writer, just list one or more **items** to print.

```
FOOTING( item1 item2 item3 ... )
```

Each **item** can be either a **literal text** or a **field name**.

To include a **literal** text in a footing line, simply enclose the text in either apostrophes or quotation marks. For example the following statement causes the words END OF SALES IN REGION: to appear in the footing line:

```
BREAK: REGION FOOTING('END OF SALES IN REGION:')
```

To put **data from an input file** in your footing line, simply list the desired field name. (Do *not* put the field name in apostrophes or quotation marks.) For example the following statement causes the contents of the REGION field to appear in the footing line:

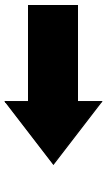
```
BREAK: REGION FOOTING(REGION)
```

Field names used in the FOOTING parm may be any of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES BY REGION'
TITLE:   'EXAMPLE OF A SINGLE FOOTING LINE'
SORT:    REGION
BREAK:   REGION FOOTING('END OF SALES IN REGION:'  REGION)
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
```



Produce this Report:

```
                          SALES BY REGION
                   EXAMPLE OF A SINGLE FOOTING LINE

           EMPL     SALES
REGION     NAME     DATE     CUSTOMER          AMOUNT        TAX

EAST   MORRISON   03/29/92 STAR MARKET             44.35       2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY          29.65       1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI           14.99       0.90
EAST   SIMPSON    04/30/92 J & S LUMBER            23.87       1.43
END OF SALES IN REGION: EAST
*** TOTAL FOR EAST  (4 ITEMS)                     112.86       6.77


NORTH  JOHNSON    04/01/92 VILLA HOTEL            234.45      14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES           9.98       0.60
NORTH  JONES      04/15/92 EZ GROCERY              10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN                10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN               121.76       7.31
END OF SALES IN REGION: NORTH
*** TOTAL FOR NORTH (5 ITEMS)                     386.69      23.22


SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL          101.38       6.09
SOUTH  JOHNSON    04/16/92 ACME BUILDING           500.00      30.00
END OF SALES IN REGION: SOUTH
*** TOTAL FOR SOUTH (2 ITEMS)                     601.38      36.09

                  (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                   1,383.66      83.05
```

Remarks:

- the footing line (specified in the BREAK statement) prints before the total line at each control break

Figure 57. Using the FOOTING parm to print a customized line at a control break

- a **built–in** field (see Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields)

By default, the data that appears in the footing line will be the field's value from the *last record* of the preceding control group. For numeric and time fields you may use a statistical parm to cause the field's **total** value, **average** value, etc. to print in the footing line. Statistical parms are discussed later in this section.

Figure 57 shows an example of a footing line that uses one literal text and one data field from the input file.

As in other print expressions, you may customize your footing line by using optional **spacing factors** and **parms**. So, the full syntax for the FOOTING parm is this:

```
FOOTING( [n] item1(parms)  [n] item2(parms)  [n] item3(parms) ... )
```

The optional **spacing factor** [n] is the number of blank spaces to leave between items in the footing line. If you omit the spacing factor, the default is for *one* blank space to appear between each item. (A spacing factor of zero is allowed if you want *no* spaces to appear between two items in a footing.) The following statement causes 5 blanks to appear between the literal text END OF SALES IN REGION: and the contents of the REGION field:

```
BREAK: REGION FOOTING('END OF SALES IN REGION:' 5 REGION)
```

The optional parms are used to provide details about how to display data fields in the footing. You may specify one or more parms, enclosed in parentheses, immediately following a field name. (Do *not* leave a space between the field name and the open parenthesis.) You may use any combination of parms, in any order. Separate the parms with a comma and/or blanks. For example, the following FOOTING parm uses both a statistical parm and a display format parm for the AMOUNT field:

```
BREAK: REGION
       FOOTING('AVERAGE SALE FOR REGION:'  AMOUNT(AVG,DOLLAR))
```

The following table shows the parms that may be used in BREAK statement print expressions:

BREAK STATEMENT PRINT EXPRESSION PARMS

PARM	DESCRIPTION

ASCII	Specifies that the field should be formatted in ASCII, rather than in EBCDIC
	
	`COMPUTE: BREAK-LIT = 'TOTAL AMOUNT IS '`
	`BREAK: REGION`
	`FOOTING(BREAK-LIT(ASCII) 0 AMOUNT(TOTAL,ASCII))`
	
	See page 143 for more information on creating ASCII output files.

AVERAGE
AVG
	Allowed only with numeric and time fields. Specifies that the field's average value for the control group should be printed. The following example specifies that the average value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('AVERAGE AMOUNT IS' AMOUNT(AVG))`

BREAK STATEMENT PRINT EXPRESSION PARMS (CONTINUED)

PARM	DESCRIPTION

BIZ	Means "blank if zero." Specifies that a field in the footing should be left blank whenever the numeric, date or time item contains zeros. The following example specifies that the HIRE-DATE field should be left blank whenever its value is zero.
	
	`BREAK: HIRE-DATE`
	`FOOTING('END OF EMPLOYEES HIRED'`
	`HIRE-DATE(BIZ))`

display–format	Specifies how to format a field in the footing. A complete list of display formats appears in Appendix B, "Display Formats" (page 617). This parm works just like the display format parm in the COLUMNS statement, which is explained in more detail beginning on page 135. The following example specifies that the HIRE–DATE field should be displayed in the LONG1 format— with the month name spelled out:
	
	`BREAK: HIRE-DATE`
	`FOOTING('END OF EMPLOYEES HIRED'`
	`HIRE-DATE(LONG1))`

LEFT/CENTER/ RIGHT	Specifies how to justify a field's data within the area reserved for it in the footing. These parms work just like the justification parms in the COLUMNS statement, which are explained in more detail beginning on page 146. The following example specifies that the contents of the HIRE–DATE field should be center justified (as well as being formatted in the LONG1 display format):
	
	`BREAK: HIRE-DATE`
	`FOOTING(HIRE-DATE(LONG1, CENTER))`

MAXIMUM MAX	*Allowed only with numeric and time fields.* Specifies that the field's maximum value in the control group should be printed. The following example specifies that the maximum value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('MAXIMUM AMOUNT IS' AMOUNT(MAX))`

MINIMUM MIN	*Allowed only with numeric and time fields.* Specifies that the field's minimum value in the control group should be printed. The following example specifies that the minimum value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('MINIMUM AMOUNT IS' AMOUNT(MIN))`

BREAK STATEMENT PRINT EXPRESSION PARMS (CONTINUED)	
PARM	**DESCRIPTION**

NZAVERAGE
NZAVG
	Allowed only with numeric and time fields. Specifies that the field's non–zero average value for the control group should be printed. (A non–zero average is the average obtained when zero values are excluded from the calculation.) The following example specifies that the non–zero average value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('AVERAGE AMOUNT IS' AMOUNT(NZAVG))`

NZMINIMUM
NZMIN
	Allowed only with numeric and time fields. Specifies that the field's non–zero minimum value in the control group should be printed. (A non–zero minimum is the minimum value, not considering zero values.) The following example specifies that the non–zero minimum value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('MINIMUM AMOUNT IS' AMOUNT(NZMIN))`

TOTAL
TOT
	Allowed only with numeric and time fields. Specifies that the field's total value for the control group should be printed. The following example specifies that the total value of the AMOUNT field should print in the footing line:
	
	`BREAK: REGION`
	`FOOTING('TOTAL AMOUNT IS' AMOUNT(TOTAL))`
	
	Note: When using TOTAL with computed fields defined with the DIVTOTS parm, be aware that the "total" value is not simply the sum of each individual value. Instead, the total value of the compute expression's numerator is divided by the total value of its denominator. This control-group-wide calculation is used whenever the "total" value of such fields is called for.

width	This numeric parm specifies how many characters should be reserved for an item in the footing. This parm works just like the width parm in the COLUMNS statement, which is explained in more detail beginning on page 135. As an example, the following statement specifies that only one character of the REGION field should appear in the footing:
	
	`BREAK: REGION`
	`FOOTING('END OF SALES IN REGION:' REGION(1))`

The ASCII, BIZ, display–format, justification and width parms all specify *how* a data field looks in the footing line. The other **statistical parms** determine *what value* will appear in the footing line. Normally when a field is used as an item in a footing print expression, the

value for the field is taken from the last record in the control group. By using one of the statistical parms (TOTAL, AVERAGE, etc.) for a numeric field, you can print a statistical value for the field, instead of its value from the last record.

Consider the following example:

```
BREAK: REGION
       FOOTING('AVERAGE SALE FOR'  REGION  'REGION IS'  AMOUNT(AVG))
```

This footing print expression consists of 4 items: two literals, and two field names. Here is how each item will be processed:

- the two literals (AVERAGE SALE FOR and REGION IS) appear in the footing line just as they are.
- the first field (REGION) has no parms in parentheses after it. Therefore, the value used for REGION in the footing line will be taken from the REGION field in the last record of the control group. Since REGION is the break field, *all* records in the control group have the same value for region. So in this case, taking the value from the last record is fine.
- the second field in the print expression (AMOUNT) has the AVG parm in parentheses after it. This means that the *average* of all AMOUNT fields in the control group will appear in the footing line. For this field, it would have been meaningless to simply print the AMOUNT field from the last record in the control group.

Figure 58 shows a sample report which uses the above statement.

Notice that the statistical keywords (TOTAL, AVERAGE, MAXIMUM, etc.) can be used in two different ways:

- We have just discussed their use as a parm within parentheses after a specific field name. When used this way, they specify what value to print for a particular field in a print line at a control break. For example:

  ```
  BREAK: REGION  FOOTING('REGION TOTAL IS'  AMOUNT(AVERAGE))
  ```

- The other use is as a BREAK statement parm (similar to the FOOTING parm). In that use, the single keyword causes a whole line of totals, averages, maximum values, etc. to print at the control break. (See page 182 and page 186 for more information on this.) For example:

  ```
  BREAK: REGION AVERAGE
  ```

Let's look at some more examples of FOOTING parms. Here's an example of using three parms with the AMOUNT field.

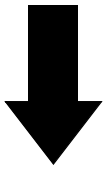
```
BREAK: REGION
       FOOTING('AVERAGE SALE FOR'  REGION  'REGION IS'
              AMOUNT(AVERAGE, PIC'$$$,$$$', LEFT))
```

The AVERAGE parm tells Spectrum Writer to print the average value of AMOUNT for the control group.

The PIC'$$$,$$$' parm shows how to format the average sales amount in the footing line. It specifies that a floating dollar sign should be used, and that only whole dollars be

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES BY REGION'
TITLE:   'EXAMPLE OF PRINTING AVERAGES IN FOOTING LINES'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION  FOOTING('AVERAGE SALE FOR'
                         REGION
                         'REGION IS'
                         AMOUNT(AVG))
```



Produce this Report:

```
                          SALES BY REGION
              EXAMPLE OF PRINTING AVERAGES IN FOOTING LINES

           EMPL     SALES
REGION     NAME     DATE     CUSTOMER          AMOUNT        TAX

EAST   MORRISON   03/29/92 STAR MARKET            44.35       2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY         29.65       1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI          14.99       0.90
EAST   SIMPSON    04/30/92 J & S LUMBER           23.87       1.43
AVERAGE SALE FOR EAST  REGION IS          28.22
*** TOTAL FOR EAST  (4 ITEMS)                    112.86       6.77


NORTH  JOHNSON    04/01/92 VILLA HOTEL           234.45      14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES          9.98       0.60
NORTH  JONES      04/15/92 EZ GROCERY             10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN               10.25       0.62
NORTH  JONES      04/15/92 TOY TOWN              121.76       7.31
AVERAGE SALE FOR NORTH REGION IS          77.34
*** TOTAL FOR NORTH (5 ITEMS)                    386.69      23.22


SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL         101.38       6.09
SOUTH  JOHNSON    04/16/92 ACME BUILDING          500.00      30.00
AVERAGE SALE FOR SOUTH REGION IS         300.69
*** TOTAL FOR SOUTH (2 ITEMS)                    601.38      36.09

                  (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                 1,383.66      83.05
```

Remarks:

- the footing line contains the AMOUNT field's average value for each region
- the example on page 197 shows how to remove the excess space that appears between the text and the average value in the footing line

Figure 58. A report which prints a field's average value in a footing line

displayed. The size of the PICTURE (7 characters) also determines how many characters are reserved in the footing line for that field.

The LEFT justification parm specifies that the average AMOUNT field should be left–justified within the 7 characters reserved for it in the footing line. This eliminates the extra blank spaces that appeared between the literal text and the actual amount in **Figure 58**. **Figure 59** (page 197) shows an example of a footing line that uses the LEFT parm.

Here is another example of a FOOTING parm. In this example, we print a footing line *instead* of a total line at the control break. The footing line will contain the total sales amount, the average sales amount, and the maximum sales amount for a region.

```
BREAK: REGION   NOTOTAL
       FOOTING('SALES STATISTICS FOR'  REGION  5
               'TOTAL:' AMOUNT(TOT,LEFT)
               'AVG:'   AMOUNT(AVG,LEFT)
               'MAX:'   AMOUNT(MAX,LEFT))
```

There are several things to notice about this example:

- the NOTOTAL parm prevents the normal total line from printing at the control break
- within the FOOTING print expression, the spacing factor of 5 helps separate the REGION field from the statistics that follow.
- the LEFT parm used along with the statistical parms (TOT, AVG, and MAX) causes the statistical value to be left justified. This arranges each value closer to its "identifier" in the footing line.

The sample report on page 197 uses a BREAK statement similar to the one above.

You may specify as many FOOTING parms as you like in a single BREAK statement. Each FOOTING parm describes one footing line. At the control break, the footing lines will print in the order of their occurrence in the BREAK statement.

The first footing line always prints immediately after the last regular report line of the control group. If you want the first footing line to be separated from the regular report lines, specify a blank footing line in your first FOOTING parm, like this:

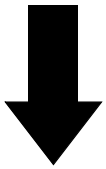
```
BREAK: REGION
       FOOTING(' ')
       FOOTING('END OF REGION:'  REGION)
       FOOTING('AVERAGE SALE:'  AMOUNT(AVG))
```

The example above will cause a blank footing line to print immediately after the last regular report line, followed by the other two footing lines. See **Figure 59** for a sample report that uses a blank FOOTING parm.

Note: In the FOOTING line, you may print statistical values for *any* numeric or time field in the input file(s). You are not limited to just those fields that appear in the COLUMNS statement.

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES BY REGION'
TITLE:   'EXAMPLE OF A FOOTING LINE WITH STATISTICS'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION  NOTOTAL
         FOOTING(' ')
         FOOTING('SALES STATISTICS FOR'  REGION  5
                 'TOTAL:' AMOUNT(TOT,LEFT)
                 'AVG:'   AMOUNT(AVG,LEFT)
                 'MAX:'   AMOUNT(MAX,LEFT))
```



Produce this Report:

```
                          SALES BY REGION
               EXAMPLE OF A FOOTING LINE WITH STATISTICS

          EMPL     SALES
REGION    NAME     DATE     CUSTOMER          AMOUNT      TAX

EAST   MORRISON   03/29/92 STAR MARKET          44.35      2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY       29.65      1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI        14.99      0.90
EAST   SIMPSON    04/30/92 J & S LUMBER         23.87      1.43

SALES STATISTICS FOR EAST      TOTAL: 112.86      AVG: 28.22        MAX: 44.35


NORTH  JOHNSON    04/01/92 VILLA HOTEL         234.45     14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES        9.98      0.60
NORTH  JONES      04/15/92 EZ GROCERY           10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN             10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN            121.76      7.31

SALES STATISTICS FOR NORTH     TOTAL: 386.69      AVG: 77.34        MAX: 234.45


SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL      101.38      6.09
SOUTH  JOHNSON    04/16/92 ACME BUILDING       500.00     30.00

SALES STATISTICS FOR SOUTH     TOTAL: 601.38      AVG: 300.69       MAX: 500.00

                    (other report lines not shown)
```

Remarks:

- the blank FOOTING parm causes a blank line to print before the real footing line
- the NOTOTAL parm in the BREAK statement suppresses the normal total line at the control break
- the footing line now displays the total, average, and maximum values for the AMOUNT field
- the LEFT justification parm causes the numeric values to be left justified, and therefore closer to their respective identifiers

Figure 59. Printing a field's total, average, and maximum values on a single line

How to Print the Number of Items in a Control Group

This section explains:

- how to use the **special built–in fields** that are available for use in the BREAK statement

We saw earlier that the default total line shows the number of items that appear in a control group. If you choose to specify a custom total line, you may also want to show the number of items that are in a control group. The special built–in field #ITEMS allows you to do this. There are also some other related built–in fields that you may wish to use in BREAK statement print expressions. These are:

BUILT-IN FIELD NAME	DESCRIPTION
#ITEMS	this numeric field contains the number of records included in the *current* control group.
#ITEM–ENDING	This 1-byte character field contains either the letter "S", or a blank, depending on the value of #ITEMS. When #ITEMS equals one, #ITEM–ENDING is a blank. Otherwise, #ITEM–ENDING is an "S". This field can be concatenated to another word to form the proper plural or singular ending for that word.
#COUNTER	this numeric field always contains the total number of records included in the report so far. It is similar to #ITEMS except that it is not reset to zero after a control break.

You can use these built–in fields just like real data fields in the print expressions for the FOOTING parm, TOTAL parm, AVERAGE parm, etc. For example:

```
BREAK: REGION
       TOTAL(REGION  'REGION HAS'  #ITEMS  'SALES')
```

As with other fields, you may also include a parm list in parentheses after the built–in field name. The following example requests that only 2 bytes be reserved in the footing line for displaying the number of items in the control group:

```
BREAK: REGION
       TOTAL(REGION  'REGION HAS'  #ITEMS(2)  'SALES')
```

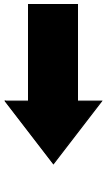
Note that if a control group only contains one record, the preceding total line would read "xxxxx REGION HAS 1 SALES" (which "ain't" good English). We can use the #ITEM–ENDING built–in field to so that the word SALE appears in the text when the control group contains only 1 record, and the word SALES appears when the control group contains multiple records. Notice that we use a spacing factor of zero, to prevent a blank space from appearing between "SALE" and the ending "S".

```
BREAK: REGION
       TOTAL(REGION  'REGION HAS'  #ITEMS(2)  'SALE'  0  #ITEM-ENDING)
```

Figure 60 shows a sample report that uses the above BREAK statement.

These Control Statements:

```
INPUT:    SALES-FILE
TITLE:    'SALES BY REGION'
COLUMNS:  REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:     REGION
BREAK:    REGION
          TOTAL(REGION  'REGION HAS' #ITEMS(2)
                'SALE' 0 #ITEM-ENDING)
```



Produce this Report:

```
                          SALES BY REGION

           EMPL     SALES
REGION      NAME     DATE     CUSTOMER        AMOUNT       TAX

EAST     MORRISON  03/29/92 STAR MARKET          44.35      2.66
EAST     MORRISON  03/30/92 A1 PHOTOGRAPHY       29.65      1.78
EAST     SIMPSON   04/01/92 EUROPEAN DELI        14.99      0.90
EAST     SIMPSON   04/30/92 J & S LUMBER         23.87      1.43
EAST   REGION HAS  4 SALES                      112.86      6.77


NORTH    JOHNSON   04/01/92 VILLA HOTEL         234.45     14.07
NORTH    JOHNSON   04/05/92 MARYS ANTIQUES        9.98      0.60
NORTH    JONES     04/15/92 EZ GROCERY           10.25      0.62
NORTH    JONES     04/15/92 TOY TOWN             10.25      0.62
NORTH    JONES     04/15/92 TOY TOWN            121.76      7.31
NORTH REGION HAS   5 SALES                      386.69     23.22


SOUTH    JOHNSON   03/12/92 ACE ELECTRICAL      101.38      6.09
SOUTH    JOHNSON   04/16/92 ACME BUILDING       500.00     30.00
SOUTH REGION HAS   2 SALES                      601.38     36.09

                    (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                1,383.66     83.05
```

Remarks:

- the customized total line uses the #ITEMS field to show the number of records included in the control group
- the width parm after #ITEMS causes only two spaces to be reserved for the number of items
- the #ITEM–ENDING built–in field contains the proper ending for the word "SALE" in the total line
- the spacing factor of 0 in the TOTAL parm puts zero spaces between the word "SALE" and the contents of the #ITEM–ENDING built–in field

Figure 60. A report that prints the number of items in a control group

Note: The special built–in fields discussed in this section may not be used in HEADING print expressions. Since the heading lines print *before* a control group, the number of items that the control group will contain is not yet known.

How to Print Header Lines at the Beginning of a Control Group

This section explains:

- how to print header lines at the **beginning** of a control group
- how to print header lines at the **top of each page**
- how to use the **HEADING and REPEAT parms** of the BREAK statement

In earlier sections we learned how to print lines at the end of a control group. You may also want to print one or more lines of text at the *beginning* of a control group. For example, you might want to print EAST REGION SALES FOLLOW at the beginning of the report lines for the East region. Use the HEADING parm of the BREAK statement to accomplish this. For example:

```
BREAK: REGION
       HEADING(REGION 'REGION SALES FOLLOW')
```

Figure 61 shows a sample report that uses the above BREAK statement.

You may have as many HEADING parms in a BREAK statement as you like. Each HEADING parm describes one heading line that will print at the beginning of a control group. The heading lines will print in the order of their occurrence in the BREAK statement.

The contents of the HEADING parm is simply a **print expression**. Print expressions tell Spectrum Writer how to build one print line to use in a report. In the HEADING parm, the print expression tells how to build a line that will print at the beginning of a new control group.

Note: The contents of the COLUMNS statement is also a **print expression**— one that tells how to build the report lines for the main body of the report. Thus, the contents of the HEADING parm is very similar to the contents of a COLUMNS statement, which you are already familiar with.

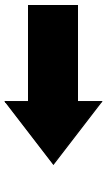
Briefly, the HEADING print expression can contain literal text and data from input records. The section titled "How to Print Customized Footing Lines at a Control Break" on page 188 describes how to write a FOOTING parm print expression in detail. Most of the same rules apply to writing HEADING parm print expressions.

There are, however, certain restriction on the print expression allowed in a HEADING parm. The special built–in fields #ITEMS, #COUNTER, and #ITEM–ENDING *may not* be used in a HEADING parm. Similarly, the statistical parms (TOTAL, AVERAGE, MAXIMUM, etc.) *may not* be used in the HEADING parm's print expression. The reason, of course, is that Spectrum Writer will not know what those values are until all of the records in the control group have been processed.

The value used for all fields appearing in a heading line will be taken from the *first* record of the control group that follows. If you want the heading lines for a control group to be

These Control Statements:

```
INPUT:    SALES-FILE
TITLE:    'SALES BY REGION'
COLUMNS:  5 REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:     REGION
BREAK:    REGION
          HEADING(REGION  'REGION SALES FOLLOW')
```



Produce this Report:

```
                          SALES BY REGION

              EMPL     SALES
     REGION   NAME     DATE     CUSTOMER          AMOUNT      TAX

EAST  REGION SALES FOLLOW
     EAST   MORRISON   03/29/92 STAR MARKET          44.35      2.66
     EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY       29.65      1.78
     EAST   SIMPSON    04/01/92 EUROPEAN DELI        14.99      0.90
     EAST   SIMPSON    04/30/92 J & S LUMBER         23.87      1.43
*** TOTAL FOR EAST  (4 ITEMS)                       112.86      6.77


NORTH REGION SALES FOLLOW
     NORTH  JOHNSON    04/01/92 VILLA HOTEL         234.45     14.07
     NORTH  JOHNSON    04/05/92 MARYS ANTIQUES        9.98      0.60
     NORTH  JONES      04/15/92 EZ GROCERY           10.25      0.62
     NORTH  JONES      04/15/92 TOY TOWN             10.25      0.62
     NORTH  JONES      04/15/92 TOY TOWN            121.76      7.31
*** TOTAL FOR NORTH (5 ITEMS)                       386.69     23.22


SOUTH REGION SALES FOLLOW
     SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL      101.38      6.09
     SOUTH  JOHNSON    04/16/92 ACME BUILDING       500.00     30.00
*** TOTAL FOR SOUTH (2 ITEMS)                       601.38     36.09

                    (other report lines not shown)


****** GRAND TOTAL (14 ITEMS)                     1,383.66     83.05
```

Remarks:

- the text specified in the HEADING parm (of the BREAK statement) prints at the beginning of each control group
- the data used for the REGION field in the heading line comes from the first record in the following control group
- the spacing factor of 5 in the COLUMNS statements shifts the report columns to the right, so that the heading lines and total lines stand out

Figure 61. A report that prints control group headings

printed at the top of **each page** of the report, add the REPEAT ("repeat headings") parm to the BREAK statement:

```
BREAK: REGION REPEAT
       HEADING('SALES IN REGION' REGION)
```

The above statement specifies a heading line to print at the beginning of each region's control group. If any such control group is large enough to print on multiple pages, the heading line will also be printed at the top of each subsequent page for that control group. Such heading lines print after the report titles and column headings, and before the first detail line of the report. The value used for a field appearing in a repeated heading line is taken from the *next* detail record after the heading line.

Computing True Percentages and Ratios at Control Breaks

By default, Spectrum Writer prints the total value of each numeric column at control breaks. For some computed fields this is not what is really desired. Consider the following COMPUTE statement:

```
COMPUTE: PERCENT-TAX = TAX / AMOUNT
```

The above statement computes a field called PERCENT–TAX, which is computed by dividing the amount of the tax by the amount of the sale. At control breaks, it is probably not helpful to see the *sum* of all of the PERCENT–TAX percentages. Instead it would be helpful to see the PERCENT–TAX percentage for the entire control group. To get this value, we need to divide the control group's *total value* for TAX by the control group's *total value* for AMOUNT.

You can do this by specifying the DIVTOTS ("divide totals") parm in the COMPUTE statement, like this:

```
COMPUTE: PERCENT-TAX(DIVTOTS) = TAX / AMOUNT
```

The above statement tells Spectrum Writer to divide the total value of the numerator by the total value of the denominator at control breaks. In this case the total value of TAX will be divided by the total value of AMOUNT. This control-group-wide percentage is what will appear in the total line at the control breaks and in the Grand Total line. You may also abbreviate DIVTOTS as DT.

Figure 62 shows a report that uses the DIVTOTS parm.

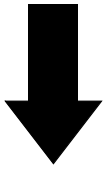
DIVTOTS may only be specified for COMPUTE statements that meet all of the following requirements:

- At its highest level, the expression must consist of a single division operation. The numerator and/or denominator themselves, however, can be expressions within parentheses. All of the following statements qualify as consisting of a "single high level division":

  ```
  COMPUTE: A = B / C
  COMPUTE: A = B / (C + D + E)
  COMPUTE: A = (B + C) / (D + E)
  COMPUTE: A = (B/C) / (D/E)
  ```

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'COMPUTING BREAK-WIDE PERCENTAGES'
COMPUTE: PERC-TAX            = TAX / AMOUNT
COMPUTE: PERCENT-TAX(DIVTOTS) = TAX / AMOUNT
SORT:    REGION(TOTAL)
COLUMNS: EMPL-NAME REGION CUSTOMER TAX AMOUNT
         PERC-TAX  PERCENT-TAX
```



Produce this Report:

COMPUTING BREAK-WIDE PERCENTAGES

EMPL NAME	REGION	CUSTOMER	TAX	AMOUNT	PERC TAX	PERCENT TAX
MORRISON	EAST	STAR MARKET	2.66	44.35	0.059977	0.059977
MORRISON	EAST	A1 PHOTOGRAPHY	1.76	29.65	0.060034	0.060034
SIMPSON	EAST	EUROPEAN DELI	0.90	14.99	0.060040	0.060040
SIMPSON	EAST	J & S LUMBER	1.43	23.87	0.059908	0.059908
*** TOTAL FOR EAST	(4 ITEMS)		6.77	112.86	0.239959	0.059986
JOHNSON	NORTH	VILLA HOTEL	14.07	234.45	0.060013	0.060013
JOHNSON	NORTH	MARYS ANTIQUES	0.60	9.98	0.060120	0.060120
JONES	NORTH	EZ GROCERY	0.62	10.25	0.060488	0.060488
JONES	NORTH	TOY TOWN	0.62	10.25	0.060488	0.060488
JONES	NORTH	TOY TOWN	7.31	121.76	0.060036	0.060036
*** TOTAL FOR NORTH	(5 ITEMS)		23.22	386.69	0.301145	0.060048
		(other report lines not shown)				
*** GRAND TOTAL	(14 ITEMS)		83.05	1,383.66	0.841332	0.060022

Remarks:

- The PERC–TAX field is computed by dividing TAX by AMOUNT.
- The PERCENT–TAX is computed the same way, but has the DIVTOTS parm.
- The total lines show the sum of the PERC–TAX field, which is meaningless for a percentage.
- The DIVTOTS parm means the PERCENT–TAX value in the total lines is computed by dividing the region's total TAX by the region's total AMOUNT.
- The PERCENT–TAX field in the Grand Total line is similarly computed by dividing the Grand Total TAX by the Grand Total AMOUNT.

Figure 62. Using the DIVTOTS parm to get accurate percentages at control breaks

- Neither the numerator nor the denominator may be literal values. Each must be either a field or an expression. Thus, DIVTOTS would *not* be allowed for the following:

```
COMPUTE: A = B / 100
```

  Computations involving division by a literal value (like the one above) are not ratios or percentages. A regular total for such fields is more appropriate at control breaks. If you need a literal in a DIVTOTS COMPUTE statement for some reason, assign the literal value to a field and then refer to that field in the COMPUTE statement:

```
COMPUTE: HUNDRED= 100
COMPUTE: A(DIVTOTS) = B / HUNDRED
```

- Only simple COMPUTE statements may use the DIVTOTS parm. It is not allowed in conditional COMPUTE statements. (Conditional COMPUTE statements are those that use the WHEN and ASSIGN parms to assign different values to a field.) However, either or both of the numerator and the denominator can be COMPUTE fields that may have been computed with conditional COMPUTE statements.

Reports with Multiple Control Breaks

This section explains:

- what **break levels** are
- what happens when a **higher level break** occurs

You may have more than one control break in a report. Spectrum Writer allows an unlimited number of control breaks. Just remember that each of the break fields *must* be a sort field.

When a report has more than one control break, each break is thought of as having a "level." The order in which the break fields are listed in the SORT statement determines each break's level. The break field appearing first in the SORT statement is considered the "highest" level break field. The break field appearing next in the SORT statement is considered the "next highest" level break field, and so on to the lowest level break field. For example, consider the following SORT statement:

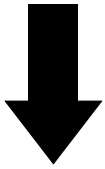
```
SORT: REGION(TOTAL)  EMPL-NAME(TOTAL)  CUSTOMER
```

This SORT statement contains three sort fields. The TOTAL parm after the first two fields makes them control break fields. REGION is the higher level break field, since it appears first in the SORT statement. EMPL–NAME is the lower level break field.

Even when BREAK statements are used to identify break fields, it is still the order of the fields in the *SORT statement* that determines the level of the break fields. The order in which

These Control Statements:

```
INPUT:   SALES-FILE
TITLE:   'SALES BY EMPLOYEE WITHIN REGION'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:    REGION(3)  EMPL-NAME(1)  CUSTOMER
```



Produce this Report:

```
                    SALES BY EMPLOYEE WITHIN REGION

           EMPL     SALES
REGION     NAME     DATE     CUSTOMER          AMOUNT       TAX

EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY          29.65      1.78
EAST   MORRISON   03/29/92 STAR MARKET             44.35      2.66
*** TOTAL FOR MORRISON   (    2 ITEMS)             74.00      4.44

EAST   SIMPSON    04/01/92 EUROPEAN DELI           14.99      0.90
EAST   SIMPSON    04/30/92 J & S LUMBER            23.87      1.43
*** TOTAL FOR SIMPSON    (    2 ITEMS)             38.86      2.33

****** TOTAL FOR EAST  (    4 ITEMS)             112.86      6.77



NORTH  JOHNSON    04/05/92 MARYS ANTIQUES           9.98      0.60
NORTH  JOHNSON    04/01/92 VILLA HOTEL            234.45     14.07
*** TOTAL FOR JOHNSON    (    2 ITEMS)           244.43     14.67

NORTH  JONES      04/15/92 EZ GROCERY              10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN                10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN               121.76      7.31
*** TOTAL FOR JONES      (    3 ITEMS)           142.26      8.55

****** TOTAL FOR NORTH (    5 ITEMS)             386.69     23.22


                    (other report lines not shown)


********* GRAND TOTAL (    14 ITEMS)           1,383.66     83.05
```

Remarks:

- the total line for EMPL–NAME, the lower level break, begins with three asterisks
- the total line for REGION begins with six asterisks, indicating its higher level
- the SORT statement specifies that 3 blank lines should print after the REGION totals, and only 1 blank line after the EMPL–NAME totals

Figure 63. A report with two levels of control breaks

the BREAK statements appear is not significant. (All BREAK statements must, however, appear after the SORT statement.) Consider the following statements:

```
SORT:  REGION  EMPL-NAME  CUSTOMER
BREAK: EMPL-NAME
BREAK: REGION
```

The preceding statements produce the very same result as the earlier example that used a SORT statement alone. REGION will be the high level break field, and EMPL–NAME will be a lower level break field (due to their relative position in the SORT statement).

Here is why a break's level is important: whenever a control break occurs for a particular break field, all *lower* level breaks are "forced." That is, a control break is automatically processed for all lower level control breaks, whether or not the contents of those break fields changed value.

For example, consider the report shown in **Figure 63** which uses a SORT statement to request two levels of control breaks. By making both REGION and EMPL–NAME break fields, the report shows the totals sales for each employee within a region, as well as for each region.

Consider what happens as Spectrum Writer is printing the report and the REGION field changes value. The control break for REGION must be processed, with region totals being printed. But, there is a lower level break than REGION, namely EMPL–NAME. So, Spectrum Writer will first process the EMPL–NAME control break, printing the sales totals for the last employee within the region. Then the control break for REGION will be processed, with the sales totals being printed for the whole region.

Now consider a place in the report, where the EMPL–NAME field changes, but the REGION field does not change. In this case Spectrum Writer will process only the EMPL–NAME control break, because there are no lower level breaks to be forced.

As a means of helping you visualize the level of the control breaks, Spectrum Writer uses a slightly different total line for each level of control break. For the lowest level control break, the total line begins with three asterisks. The total line for the next higher level break begins with six asterisks. Each higher level control break gets three additional asterisks. This helps when you are scanning a report for a particular level of break totals. Just scan down the left side of the report looking for the total line with the appropriate number of asterisks.

When more than one control break is used in a report, it is often desirable to use a larger spacing factor for the higher level break(s). For example we might want to just skip 1 line whenever the EMPL–NAME changes, but skip to a whole new page whenever the REGION changes. This would be specified by using a break spacing parm in either the SORT statement or the BREAK statement (see page 178). For example:

```
SORT:  REGION  EMPL-NAME  CUSTOMER
BREAK: REGION     SPACE(PAGE)
BREAK: EMPL-NAME  SPACE(1)
```

Or, to specify the same spacing parms directly in the SORT statement:

```
SORT: REGION(PAGE)  EMPL-NAME(1)  CUSTOMER
```

How to Customize the Grand Totals

This section explains:

- how the Grand Totals are processed by **default**
- how to print **additional statistical lines** (average, maximum and minimum) at the Grand Total
- how to **customize** the Grand Total lines
- how to **suppress** the Grand Totals

Spectrum Writer treats the end of a report like one final "control break". The "control group" for this break includes the entire report. As with any other control break, Spectrum Writer prints a total line at this special control break. This total line is what appears as the "Grand Total" line in your report.

You can customize the Grand Total control break, just like you do for regular control breaks. Just use the special field name **#GRAND** in a BREAK statement. For example:

```
BREAK: #GRAND  AVERAGE  MAXIMUM  MINIMUM
```

In the above statement the field name #GRAND specifies that the information on this BREAK statement pertains to the Grand Total break at the end of the report. The AVERAGE parm specifies that a line of averages should print at the control break (that is, at the end of the report). The MAXIMUM and MINIMUM parms specify that a line of maximums and a line of minimums should also print. **Figure 64** shows a sample report that uses this BREAK statement.

You may use all of the normal BREAK statement parms (except for SPACE) in the BREAK statement for #GRAND. See the section titled "Customizing the Control Breaks" (page 177) to learn what all you can do with a BREAK statement.

Here is another example of a #GRAND BREAK statement:

```
BREAK: #GRAND  TOTAL(#ITEMS 'SALES LISTED IN REPORT')
               AVERAGE('AVERAGE SALE IN REPORT')
```

The above statement uses the TOTAL parm to specify a custom total line. The text "nnn,nnn SALES LISTED IN REPORT" will now appear in the Grand Total line rather than the usual "*** GRAND TOTAL (nnnnn ITEMS)". The AVERAGE parm causes a line of averages to print at the end of report. It also specifies what text the average line should begin with ("AVERAGE SALE IN REPORT").

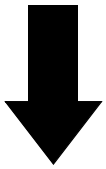
The FOOTING parm may also be specified in the #GRAND BREAK statement. Footing lines print at the end of a control group. The entire report is the "control group" for the Grand Total control break. Therefore, any footing lines specified in this statement will print only once — at the end of the report.

Note: If you want to a line at the bottom of *every* page, use a FOOTNOTE statement.

The HEADING parm may also be used in the #GRAND BREAK statement. Any HEADING lines specified will print once at the very beginning of the report (after the title lines and column

These Control Statements:

```
INPUT:    SALES-FILE
TITLE:    'SALES BY REGION'
TITLE:    'SHOWING COMPANY-WIDE STATISTICS'
COLUMNS:  REGION EMPL-NAME SALES-DATE CUSTOMER AMOUNT TAX
SORT:     REGION EMPL-NAME SALES-DATE
BREAK:    #GRAND  AVERAGE  MAXIMUM  MINIMUM
```



Produce this Report:

```
                          SALES BY REGION
                    SHOWING COMPANY-WIDE STATISTICS

           EMPL     SALES
REGION     NAME     DATE     CUSTOMER          AMOUNT       TAX

EAST   MORRISON   03/29/92 STAR MARKET            44.35      2.66
EAST   MORRISON   03/30/92 A1 PHOTOGRAPHY         29.65      1.78
EAST   SIMPSON    04/01/92 EUROPEAN DELI          14.99      0.90
EAST   SIMPSON    04/30/92 J & S LUMBER           23.87      1.43
NORTH  JOHNSON    04/01/92 VILLA HOTEL           234.45     14.07
NORTH  JOHNSON    04/05/92 MARYS ANTIQUES          9.98      0.60
NORTH  JONES      04/15/92 EZ GROCERY             10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN               10.25      0.62
NORTH  JONES      04/15/92 TOY TOWN              121.76      7.31
SOUTH  JOHNSON    03/12/92 ACE ELECTRICAL        101.38      6.09
SOUTH  JOHNSON    04/16/92 ACME BUILDING         500.00     30.00
WEST   BAKER      03/26/92 JACKS CAFE            137.00      8.22
WEST   BAKER      04/12/92 JACKS CAFE            135.75      8.15
WEST   THOMAS     04/14/92 YOGURT CITY             9.98      0.60


*** GRAND TOTAL (14 ITEMS)                    1,383.66     83.05
*** AVERAGE VALUE                                98.83      5.93
*** MAXIMUM VALUE                               500.00     30.00
*** MINIMUM VALUE                                 9.98      0.60
```

Remarks:

- the BREAK statement for #GRAND specifies how to process the Grand Total "control break"
- the AVERAGE, MAXIMUM and MINIMUM parms cause those statistical lines to print along with the Grand Total line
- the TOTAL parm was not needed, since total lines print at control breaks by default

Figure 64. A report with customized Grand Totals

headings). If the REPEAT parm is also specified, these HEADING lines will also be repeated at the top of each page of the report.

As mentioned earlier, a total line prints at the Grand Total control break by default. In addition, any other statistical lines that printed at a standard control break will also print by default at the Grand Total control break. Thus, for example, if an average line and a maximum line printed at a REGION control break, an average line and maximum line will also print at the Grand Total control break. As shown in the previous example, you may also explicitly request any of these statistical lines, even if no other control break specified them.

The SPACE parm in a BREAK statement is used to specify the spacing to perform *after* a control break. Since there is no more report following the Grand Total control break, any SPACE parm specified for it will be ignored.

Spacing *before* the Grand Total break is determined as follows. If any other control break specified a SPACE parm of NEWSHEET, then the Grand Totals will also be printed on a new sheet of paper. Otherwise, if any other control break specified ODDPAGE, then the Grand Total will also go on the next odd page. Otherwise, if any other control break specified PAGE, then the Grand Totals will go on a new page. (If the NEWSHEET1, ODDPAGE1, or PAGE1 variation of these parms was used, then the Grand Total page will be numbered page 1 as well.)

If no real control breaks used any of the page spacing options, then the Grand Totals will be printed after skipping two blank lines.

To **suppress the Grand Total line** altogether, you can do one of two things.

You can use the NOGRANDTOTAL parm in an OPTIONS statement, like this:

```
OPTIONS: NOGRANDTOTAL
```

Figure 66 (page 213) uses the above statement.

Or, you can use a BREAK statement for the #GRAND break and specify the NOTOTAL parm, like this:

```
BREAK: #GRAND NOTOTAL
```

How to Produce Summary Reports

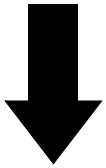
This section explains:

- what a summary report is
- how to convert a regular report into a summary report

A summary report is one which does not show detail information about every record included in the report. Instead, the detail information is *summarized*, with just the totals actually appearing in the report. Chapter 2, "How to Request a Report" included a lesson on creating summary reports (page 73). And a lesson in "How to Request a PC File" on page 83 showed how to create summary PC files (page 113).

These Control Statements:

```
OPTION:  SUMMARY
INPUT:   SALES-FILE
TITLE:   'EMPLOYEE SALES SUMMARY'
COLUMNS: 40  AMOUNT  TAX
SORT:    REGION(TOTAL)  EMPL-NAME(TOTAL)
```



Produce this Report:

```
                    EMPLOYEE SALES SUMMARY

                                          AMOUNT      TAX

*** TOTAL FOR MORRISON   (    2 ITEMS)          74.00       4.44
*** TOTAL FOR SIMPSON    (    2 ITEMS)          38.86       2.33
****** TOTAL FOR EAST  (    4 ITEMS)          112.86       6.77

*** TOTAL FOR JOHNSON    (    2 ITEMS)         244.43      14.67
*** TOTAL FOR JONES      (    3 ITEMS)         142.26       8.55
****** TOTAL FOR NORTH (    5 ITEMS)          386.69      23.22

*** TOTAL FOR JOHNSON    (    2 ITEMS)         601.38      36.09
****** TOTAL FOR SOUTH (    2 ITEMS)          601.38      36.09

*** TOTAL FOR BAKER      (    2 ITEMS)         272.75      16.37
*** TOTAL FOR THOMAS     (    1 ITEM )           9.98       0.60
****** TOTAL FOR WEST  (    3 ITEMS)          282.73      16.97


********* GRAND TOTAL (    14 ITEMS)        1,383.66      83.05
```

Remarks:

- no regular report lines print— only the total lines from the two levels of control breaks
- the total line for EMPL–NAME, the lower level break, begins with three asterisks
- the total line for REGION begins with six asterisks, indicating its higher level
- the spacing factor of 40 (in the COLUMNS statement) move the AMOUNT column over 40 spaces, leaving room for the total line text to print on the same line as the totals themselves
- note that it is okay to sort a report on fields which do not appear in the COLUMNS statement

Figure 65. A summary report that uses two levels of control breaks

In each case, an OPTIONS statement with the SUMMARY parm was used:

```
OPTIONS: SUMMARY
```

The SUMMARY parm causes two things to happen:

- it specifies that *zero* detail lines (per control group) will print. This is the same as specifying:

  ```
  OPTIONS: DETAIL(0)
  ```

  The only lines that print in such a report are the lines that are associated with control breaks: heading lines, footing lines, totals line, average lines, etc.

- it sets the break spacing value for the lowest level break to *zero* blank lines (instead of the normal default of two blank lines). This prevents two blank lines from appearing between every line in the summary report.

Figure 65 shows an example of a summary report. This report contains *two* levels of breaks. It is very similar to the detail report shown earlier in **Figure 63** (page 205). The main difference is that in **Figure 65** the detail lines have been suppressed and only the EMPL–NAME and REGION total lines are printed.

Notice that in summary reports **only numeric columns** are filled in. That is because only numeric columns can be totalled, or "summarized." Therefore, in this report we removed the non–numeric columns (REGION, EMPL–NAME, SALES–DATE and CUSTOMER) from the COLUMNS statement. We added a spacing of 40 to the COLUMNS statement ahead of the first field in order to push that field 40 spaces over in the report. That was necessary to prevent overlap between the total line text ("*** TOTAL FOR...") and the first actual total (in the AMOUNT column). If we had not done that, the control break total lines would have split onto two lines, making a less attractive report. (See page 181.)

Note: If you request a SUMMARY report and do not specify any control breaks, your report will contain only the Grand Total line. This is useful when you want to summarize *all* of the detail lines in the entire report.

Printing a "Line Number" in Your Report

You have already seen how to use the #ITEM built–in field in BREAK statements (page 198). In the BREAK statement, #ITEM represents the total number of records in a control group. This is the same value that appears in the default total line printed at control breaks.

You can also specify #ITEM as a field in your COLUMNS statement. It's value will be an ascending, sequential "item number" representing the number of items included in the control group *so far*. That is, it will be "1" for the first item printed in a control group, "2" for the next item and so on. #ITEM's value is reset to zero after each control break. It then begins again numbering the items in the next control group. (Of course, if your report has no control breaks, the value of #ITEM will not be reset.)

Using #ITEM in your COLUMNS statement allows you to print a "rank" or a "line number" for each record printed in your report.

You might also want to print an "item number" and *not* have it reset at each control break. To allow this, there are additional built–in fields named #ITEM2, #ITEM3, and so on through #ITEM9. #ITEM2 is similar to #ITEM, but is *not reset* at the lowest level of control break. However, if you have *two* levels of control breaks in your report, #ITEM2 will be reset to zero whenever the higher level control break occurs. Similarly, #ITEM3 is not reset at the two lowest level control breaks, but is reset when the third level of control break occurs. By using the appropriate #ITEM built–in field, you can print item numbers and have them reset whenever you like for reports with up to 9 levels of control breaks.

The report in **Figure 66** (page 213) uses the #ITEM built–in field.

Note: #ITEM may also be spelled #ITEM1.

How to Create "Top 10" Type Reports

This section explains:

- how to create **"Top 10"** type reports
- how to use the **DETAIL parm** in the OPTIONS statement

The DETAIL(nnn) option tells Spectrum Writer to print only a limited number of detail records in the report for each control group. We saw in an earlier section that specifying the SUMMARY option causes the DETAIL(0) option to be in effect. DETAIL(0) requests that *no* detail records be printed for each control group in the report.

To produce a "Top 5" or "Top 10" type of report, use the DETAIL parm with whatever value is appropriate for your report. For example:

```
OPTIONS: DETAIL(3)
```

In the above example we request that only 3 detail lines print for each control group. That will cause just the first 3 records in each control group to print in our report.

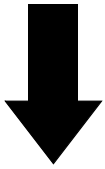
Consider the "Top 3 Sales" report in **Figure 66** which uses the above statement. This report is sorted first in REGION order, and then in *descending* AMOUNT order. We also made REGION a control break. The result is that within each REGION, the largest sale prints first, the next largest sale prints next, and so on. By using the DETAIL(3) option, our report shows only the 3 largest sales in each region.

Here are a few other things to note about this kind of report:

- the DETAIL option specifies the **maximum** number of records to print per control group. If a control group does not contain that many records, all records for that control group are printed. (In **Figure 66**, the "SOUTH" region is an example of this. There are only two sales for that region.)
- the control group **totals line** will still contain the total value of the entire control group — not just the total of those detail records that are actually printed. You can use the NOTOTALS parm in the BREAK statement to suppress the totals if you prefer (as we did in **Figure 66**).

These Control Statements:

```
OPTIONS: DETAIL(3)  NOGRANDTOTAL
INPUT:   SALES-FILE
TITLE:   'TOP 3 SALES IN EACH REGION'
SORT:    REGION  AMOUNT(DESC)
BREAK:   REGION  NOTOTALS
COLUMNS: #ITEM('RANK')
         REGION  EMPL-NAME  SALES-DATE  CUSTOMER  AMOUNT  TAX
```



Produce this Report:

```
                        TOP 3 SALES IN EACH REGION

                  EMPL     SALES
 RANK  REGION     NAME      DATE     CUSTOMER        AMOUNT      TAX

    1  EAST     MORRISON   03/29/95  STAR MARKET        44.35     2.66
    2  EAST     MORRISON   03/30/95  A1 PHOTOGRAPHY     29.65     1.78
    3  EAST     SIMPSON    04/30/95  J & S LUMBER       23.87     1.43


    1  NORTH    JOHNSON    04/01/95  VILLA HOTEL       234.45    14.07
    2  NORTH    JONES      04/15/95  TOY TOWN          121.76     7.31
    3  NORTH    JONES      04/15/95  TOY TOWN           10.25     0.62


    1  SOUTH    JOHNSON    04/16/95  ACME BUILDING     500.00    30.00
    2  SOUTH    JOHNSON    03/12/95  ACE ELECTRICAL    101.38     6.09


    1  WEST     BAKER      03/26/95  JACKS CAFE        137.00     8.22
    2  WEST     BAKER      04/12/95  JACKS CAFE        135.75     8.15
    3  WEST     THOMAS     04/14/95  YOGURT CITY         9.98     0.60
```

Remarks:

- the DETAIL(3) option causes a maximum of 3 detail lines per control group to print
- the #ITEM built–in field (in the COLUMNS statement) lets us print a "rank" for each detail record
- the NOTOTALS parm (in the BREAK statement) suppresses the control break totals (which would not be the sum of the detail records printed)
- the NOGRANDTOTAL option suppresses the Grand Totals, which would not be the sum of the detail records printed

Figure 66. "Top 3 Sales in Region" report

- if a report with a DETAIL(nnn) option does not have any control breaks, the whole report is treated as a single control group. In that case, just the first "nnn" records of the entire report will print.

How to Count "Occurrences" in a File

This section explains:

- how to count the **number of times a certain value occurs** in a file

Say that we wanted to know how many of the employees in the EMPL–FILE are based in California. Or, what if we wanted to know the count of male and female employees. To get statistics like these from a file, we use a special type of summary report. **Figure 67** and **Figure 68** (page 216) show examples of such reports.

In these reports, we first create a number of new fields using conditional COMPUTE statements. These fields are used as "counter" fields. They count the number of times that a certain field contains a particular value. For example, the NUMBER–OF–MALE field counts the number of times that the SEX field in the EMPL–FILE contains "M". Consider the following statement:

```
COMPUTE: NUMBER-OF-MALE = WHEN(SEX='M')  ASSIGN(1)
```

After each record is read from the input file, the value of the NUMBER–OF–MALE field is computed. Its value will always be either 1 or 0. When the SEX field contains the value "M", the NUMBER–OF–MALE field will contain a 1. Otherwise, the NUMBER–OF–MALE field will contain a 0 (the default value when no WHEN expressions are true). By adding up all of the NUMBER–OF–MALE fields in the report, we can get a total count of the records whose SEX field contained an "M".

We set up a similar counter field for each statistic that we are interested in. These counter fields are then listed in the COLUMNS statement. The Grand Total line shows us the total value for each of these "counters".

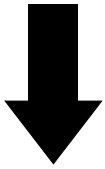
You would normally use the SUMMARY option to suppress all of the detail lines leaving just the statistics. In **Figure 67** we printed the detail lines to better illustrate how the counter fields work.

You can break your statistics down further by simply adding one or more **control breaks** to such a report. For example, by sorting and breaking on the DEPT–NUM field, we can get the same statistics *by department number*. That is, we can see the number of males and females in each department. The sample report in **Figure 68** (page 216) shows an example of printing statistics by department number. In this report we used the SUMMARY option to suppress the individual detail lines. We also removed from the COLUMNS statement those fields which do not print in the total lines.

> **Note:** Another way to get "count" statistics is to simply sort the report on the item you want to count (the STATE field, for instance), and make it a control break. Each time the STATE field changes value, a control break will occur and the number of "items" in that state will print. The disadvantage of this method is that only one "thing" can be counted at a time.

These Control Statements:

```
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE FILE COUNTS'

COMPUTE: NUMBER-OF-MALE       = WHEN(SEX='M')    ASSIGN(1)
COMPUTE: NUMBER-OF-FEMALE     = WHEN(SEX='F')    ASSIGN(1)
COMPUTE: NUMBER-IN-CALIFORNIA = WHEN(STATE='CA') ASSIGN(1)
COMPUTE: NUMBER-IN-ARIZONA    = WHEN(STATE='AZ') ASSIGN(1)
COMPUTE: NUMBER-OF-FULLTIME   = WHEN(FULL-TIME)  ASSIGN(1)

COLUMNS: LAST-NAME  FIRST-NAME  DEPT-NUM  STATE
         NUMBER-OF-MALE         NUMBER-OF-FEMALE
         NUMBER-IN-CALIFORNIA   NUMBER-IN-ARIZONA
         NUMBER-OF-FULLTIME
```



Produce this Report:

```
                                EMPLOYEE FILE COUNTS

                                                  NUMBER  NUMBER    NUMBER   NUMBER   NUMBER
     LAST            FIRST      DEPT              OF      OF        IN       IN       OF
     NAME            NAME       NUM  STATE        MALE    FEMALE    CALIFORNIA ARIZONA  FULLTIME

JONES            JERRY            2    CA          1       0          1       0        1
JOHNSON          THOMAS           1    AZ          1       0          0       1        1
JOHNSON          LINDA            2    CA          0       1          1       0        1
MACDONALD        RICHARD          2    CA          1       0          1       0        0
SIMPSON          TIMOTHY          3    CA          1       0          1       0        1
MORRISON         MICHAEL          3    CA          1       0          1       0        1
CHRISTOPHERSON   MELISSA          1    AZ          0       1          0       1        1
BAKER            VIVIAN           4    CA          0       1          1       0        1
THOMAS           MARTIN           4    CA          1       0          1       0        1


*** GRAND TOTAL (9 ITEMS)                          6       3          7       2        8
```

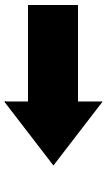
Remarks:

- several "counter" fields are created using conditional COMPUTE statements
- the counter fields are totalled at the end of the report, giving us our statistics
- you would normally use an OPTIONS: SUMMARY statement to suppress the detail lines from such a report

Figure 67. Counting how many times various values occur in a file

These Control Statements:

```
OPTIONS: SUMMARY
INPUT:   EMPL-FILE
TITLE:   'EMPLOYEE FILE COUNTS, BY DEPARTMENT'

COMPUTE: NUMBER-OF-MALE       = WHEN(SEX='M')    ASSIGN(1)
COMPUTE: NUMBER-OF-FEMALE     = WHEN(SEX='F')    ASSIGN(1)
COMPUTE: NUMBER-IN-CALIFORNIA = WHEN(STATE='CA') ASSIGN(1)
COMPUTE: NUMBER-IN-ARIZONA    = WHEN(STATE='AZ') ASSIGN(1)
COMPUTE: NUMBER-OF-FULLTIME   = WHEN(FULL-TIME)  ASSIGN(1)

SORT:    DEPT-NUM
BREAK:   DEPT-NUM  TOTAL('COUNTS FOR DEPARTMENT' DEPT-NUM)

COLUMNS: 40
         NUMBER-OF-MALE        NUMBER-OF-FEMALE
         NUMBER-IN-CALIFORNIA  NUMBER-IN-ARIZONA
         NUMBER-OF-FULLTIME
```



Produce this Report:

```
                        EMPLOYEE FILE COUNTS, BY DEPARTMENT

                                      NUMBER  NUMBER    NUMBER   NUMBER   NUMBER
                                        OF      OF        IN       IN       OF
                                      MALE    FEMALE  CALIFORNIA ARIZONA FULLTIME

COUNTS FOR DEPARTMENT 1                    1       1          0       2        2
COUNTS FOR DEPARTMENT 2                    2       1          3       0        2
COUNTS FOR DEPARTMENT 3                    2       0          2       0        2
COUNTS FOR DEPARTMENT 4                    1       1          2       0        2


****** GRAND TOTAL (9 ITEMS)               6       3          7       2        8
```

Remarks:

- this report is similar to the report in the preceding figure
- in this report we added a control break for DEPT–NUM, giving us department totals as well as Grand Totals
- the OPTIONS: SUMMARY statement suppressed all detail lines from the report
- the COLUMNS statement only lists the counter fields, since no detail records are printed
- the initial spacing factor of 40 (in the COLUMNS statement) moves the first column 40 spaces to the right, leaving room for the total line text to print

Figure 68. Breaking down “count” statistics further

How to Break Totals Down into Categories

This section explains:

- how to break a total down "by category" (such as "by sex")

In the preceding section, we saw how to count the number of males and females in a control group. Now let's take that a step further. What if we wanted to calculate the total sales made by males and females? We are no longer simply counting occurrences, but accumulating some field's total by category.

Of course, one way to do that is to sort and break on the SEX field. That would cause all records for each sex to be grouped and printed together, with control break totals printed for each group. If we listed TOTAL–SALES in the report, the control break totals would show the total sales for each sex. But assume we want such totals by sex *without* having to sort on the SEX field? And assume we want to see the male and female totals together in the same line, rather than in separate total lines.

There is another technique we can use to accomplish this. Again, we use a conditional COMPUTE statement:

```
COMPUTE: MALE-SALES = WHEN(SEX='M')  ASSIGN(TOTAL-SALES)
```

After each new record is read from the input file, the value of MALE–SALES will be computed. Its value will always be either zero or the employee's total sales amount (from the TOTAL–SALES field). When the SEX field contains an "M" the MALE–SALES field will contain the TOTAL–SALES value. Otherwise, the MALE–SALES field will contain a zero. By adding up all of the MALE–SALES fields in the report, we can get the total sales made by all males.

To get the amount sold by females, we use a similar statement:

```
COMPUTE: FEMALE-SALES = WHEN(SEX='F')  ASSIGN(TOTAL-SALES)
```

Figure 69 shows a report that uses the above statements. We put the MALE–SALES and FEMALE–SALES field in the COLUMNS statement. Those fields are then automatically totalled and printed at each control break, as well as at the Grand Totals.

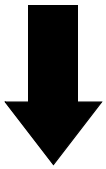
By adding the SUMMARY option, we could suppress the detail lines and see just the total lines.

This technique can often be used to total a field by category, instead of just getting a single total for it. Use one COMPUTE statement for each possible value of the "category" field. Of course, this technique cannot be used if all of the possible values of the category field are not known in advance.

> **Note:** **Figure 69** is an example of a "crosstab report." Crosstab reports are discussed in more depth in the following sections.

These Control Statements:

```
INPUT:    EMPL-FILE
TITLE:    'SALES TOTALS, BY GENDER'

COMPUTE:  MALE-SALES   =  WHEN(SEX='M')    ASSIGN(TOTAL-SALES)
COMPUTE:  FEMALE-SALES  = WHEN(SEX='F')    ASSIGN(TOTAL-SALES)

SORT:     DEPT-NUM
BREAK:    DEPT-NUM  TOTAL('SALES IN DEPARTMENT' DEPT-NUM)

COLUMNS:  LAST-NAME  FIRST-NAME  DEPT-NUM  SEX
          TOTAL-SALES(12)  MALE-SALES(12)  FEMALE-SALES(12)
```



Produce this Report:

```
                          SALES TOTALS, BY GENDER

     LAST            FIRST       DEPT          TOTAL         MALE         FEMALE
     NAME            NAME         NUM   SEX    SALES        SALES        SALES

JOHNSON         THOMAS                1  M     86,999.24    86,999.24         0.00
CHRISTOPHERSON  MELISSA               1  F     47,665.31         0.00    47,665.31
SALES IN DEPARTMENT       1                   134,664.55    86,999.24    47,665.31


JONES           JERRY                 2  M     42,509.89    42,509.89         0.00
JOHNSON         LINDA                 2  F     75,023.55         0.00    75,023.55
MACDONALD       RICHARD               2  M      2,560.98     2,560.98         0.00
SALES IN DEPARTMENT       2                   120,094.42    45,070.87    75,023.55


SIMPSON         TIMOTHY               3  M      8,723.88     8,723.88         0.00
MORRISON        MICHAEL               3  M     98,054.99    98,054.99         0.00
SALES IN DEPARTMENT       3                   106,778.87   106,778.87         0.00


BAKER           VIVIAN                4  F     92,125.89         0.00    92,125.89
THOMAS          MARTIN                4  M     60,193.49    60,193.49         0.00
SALES IN DEPARTMENT       4                   152,319.38    60,193.49    92,125.89


****** GRAND TOTAL (9 ITEMS)                   513,857.22   299,042.47   214,814.75
```

Remarks:

- in the detail lines, MALE–SALES and FEMALE–SALES contain either zero or the value from the TOTAL–SALES field.
- the totals for those fields show the total sales made by male and female employees

Figure 69. Subtotaling fields by a category (such as gender)

How to Make “Crosstab” Reports

This section explains

- what crosstab reports are
- a method of formatting crosstab reports with Spectrum Writer

Crosstab reports show information in a tabular format, based on two groupings of data. One grouping runs down the left side of the report; the other grouping runs across the top of the report. The intersections of these two grouping contains the report data — the data applicable to that combination of group values.

Spectrum Writer does not have an automatic "crosstab" report option. However, in many cases you can create a report in crosstab format yourself.

The next sections illustrate a method of making reports in crosstab format. There are two requirements for using this method:

- you must know in advance the number of different values that the “across the top” field might contain
- and you must know in advance what all of those values will be

If the crosstab report you want to produce meets these two conditions, then use the method described in the following examples.

A Simple Crosstab Report

Let’s start with a simple example. Assume that we want to produce a crosstab report from the sales file. We want to group the sales by month going down the page and by region going across the page. This report is shown in **Figure 70**.

Note that this report meets the requirements mentioned above. The “across” field in this case is REGION. And we do know both the number of different values it can have (four) and what those values will be (EAST, NORTH, SOUTH and WEST).

We first use COMPUTE statement to create four region “buckets” to hold the sales data. These correspond to the four values that the REGION field can have. As each record is read from the input file, we assign its AMOUNT field to one of those four buckets, depending on the value of the REGION field.

```
COMPUTE: EAST-AMT  = WHEN(REGION = “EAST”)  ASSIGN(AMOUNT)
COMPUTE: NORTH-AMT = WHEN(REGION = “NORTH”) ASSIGN(AMOUNT)
COMPUTE: SOUTH-AMT = WHEN(REGION = “SOUTH”) ASSIGN(AMOUNT)
COMPUTE: WEST-AMT  = WHEN(REGION = “WEST”)  ASSIGN(AMOUNT)
```

We don’t want to print detail records for this report. We only to print to print a summary report showing one line of totals for each month. Therefore we do not need a COLUMNS statement. We also specify the SUMMARY option. We use TITLE statements to produce our “column headings.” The trailing slashes in these statements prevent these "titles" from centered. (See page 153 for further explanation. Column headings are not produced automatically for this report, since there is no COLUMNS statement.)

These Control Statements:

```
OPT: SUMMARY DATEDELIM('-') NOGRAND
INPUT: SALES-FILE
COMPUTE: EAST-AMT  = WHEN(REGION='EAST')  ASSIGN(AMOUNT)
COMPUTE: NORTH-AMT = WHEN(REGION='NORTH') ASSIGN(AMOUNT)
COMPUTE: SOUTH-AMT = WHEN(REGION='SOUTH') ASSIGN(AMOUNT)
COMPUTE: WEST-AMT  = WHEN(REGION='WEST')  ASSIGN(AMOUNT)
COMPUTE: MONTH = #LEFT(#FORMAT(SALES-DATE, YYYY-MM-DD),7)
TITLE: 'CROSSTAB SALES REPORT'
TITLE: 'REGIONAL TOTALS BY MONTH'
TITLE: ' '
TITLE: ' MONTH            EAST          NORTH          SOUTH           W
 EST' /
TITLE: '_______   ___________   ___________    __________   _________
 ___' /
SORT: MONTH
BREAK: MONTH NOTOTALS
 FOOTING(MONTH EAST-AMT(TOTAL)  NORTH-AMT(TOTAL)
               SOUTH-AMT(TOTAL) WEST-AMT(TOTAL) )
```

Produce this Report:

```
                        CROSSTAB SALES REPORT
                       REGIONAL TOTALS BY MONTH

 MONTH        EAST         NORTH        SOUTH        WEST

1995-03        74.00          0.00       101.38       137.00
1995-04        38.86        386.69       500.00       145.73
```

Remarks:

- this crosstab report has months running down the page and regions running across the page

Figure 70. A Simple Crosstab Report

We use a COMPUTE statement to extract the month from the SALES-DATE field. (Note that we also include the year along with the month. That allows this report to work even if the input data spans multiple years.)

```
COMPUTE: MONTH = #LEFT(#FORMAT(SALES-DATE, YYYY-MM-DD),7)
```

This statement first formats the SALES-DATE in YYYY-MM-DD format. (We used the DATEDELIM option to cause dates to be formatted with a dash, instead of the standard slash.) The #LEFT function then takes just the YYYY-MM portion of that date. We then use this MONTH field as our sort field and our break field.

As Spectrum Writer processes all of the sales records for a given month, it simply distributes the AMOUNT fields into the correct region buckets. No report lines are printed.

Then, when the MONTH changes value, we are ready to print one line of our crosstab report. Since this is a summary report, we don't use a COLUMNS statement to describe this print line. Instead we describe the line using the BREAK statement FOOTING parm:

```
BREAK: MONTH NOTOTALS FOOTING(MONTH EAST-AMT(TOTAL)  NORTH-AMT(TOTAL)
                                   SOUTH-AMT(TOTAL) WEST-AMT(TOTAL) )
```

As you can see in the FOOTING parm above, our report lines will start with the contents of the MONTH field that just ended. That takes care of the "down the page" group in our crosstab report. Then we print the total values of the four regional buckets where we earlier distributed the AMOUNT field. This provides the "across the page" grouping. The result is one line of our crosstab report. (The NOTOTAL parm in the above statement prevents the default total line from printing at the control break.)

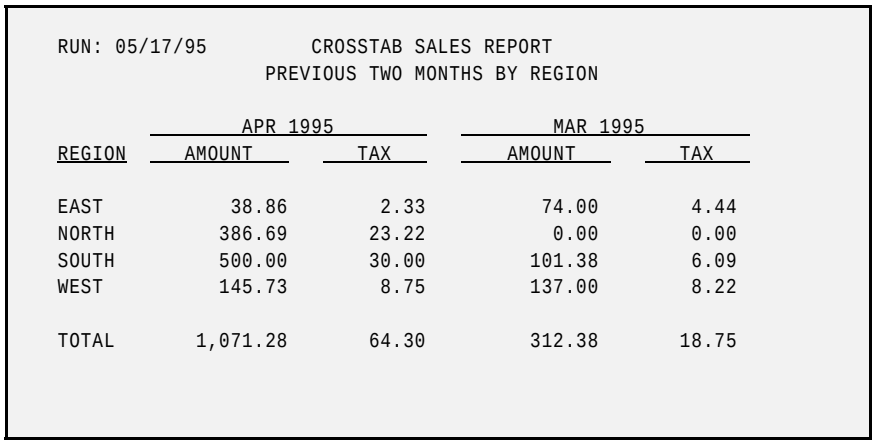
> **Note:** The small SALES-FILE only contains data for two months. Therefore, the crosstab report only has two lines of data. But this report would work equally well for a file containing many months (even years) of data.

Another Crosstab Report

Our second crosstab example is similar to the previous example. This time, however, we switch the "down" and "across" groups (see **Figure 71**). In addition, we made some other changes to illustrate additional features and variations. The changes in this report example are:

- The regions now run down the page, and the months run across the page
- we show *two* totals at each intersection (AMOUNT and TAX), instead of just one
- instead of "hardcoding" the two months to appear as the "across" values, we code the report to always report on the "previous two months" (based on the system date of the run)
- this report also shows Grand Totals

Following are some remarks about the control statements used to produce this report. (The control statements are shown in **Figure 72** on page 223.)



```
RUN: 05/17/95          CROSSTAB SALES REPORT
                    PREVIOUS TWO MONTHS BY REGION

          ___________APR 1995___________   ___________MAR 1995___________
REGION    ___AMOUNT___   ____TAX____   ___AMOUNT___   ____TAX____

EAST           38.86         2.33          74.00         4.44
NORTH         386.69        23.22           0.00         0.00
SOUTH         500.00        30.00         101.38         6.09
WEST          145.73         8.75         137.00         8.22

TOTAL       1,071.28        64.30         312.38        18.75
```

Figure 71. Another sample crosstab report

First we needed to determine the month number of the two previous months. Note that the #MONTHNUM built-in function returns a number from 1 to 12 for a given date. In this case, it returns the month number of the system date (#TODAY).

```
COMPUTE: SYSTEM-MONTH-NUM = #MONTHNUM(#TODAY))
COMPUTE: PREV1-MONTH-NUM = WHEN(SYSTEM-MONTH-NUM = 1) ASSIGN(12)
                           ELSE    ASSIGN(SYSTEM-MONTH-NUM - 1)
COMPUTE: PREV2-MONTH-NUM = WHEN(PREV1-MONTH-NUM = 1) ASSIGN(12)
                           ELSE    ASSIGN(PREV1-MONTH-NUM - 1)
```

Similarly, we compute the year associated with each of the two previous months. (They might be in the same year as the system date, or one or both of them could be in the previous year.) The #YEARNUM built-in function returns the numeric year of a given date.

```
COMPUTE: SYSTEM-YEAR-NUM = #YEARNUM(#TODAY))
COMPUTE: PREV1-YEAR-NUM = WHEN(SYSTEM-MONTH-NUM > 1) ASSIGN(SYSTEM-YEAR-NUM)
                          ELSE    ASSIGN(SYSTEM-YEAR-NUM - 1)
COMPUTE: PREV2-YEAR-NUM = WHEN(SYSTEM-MONTH-NUM > 2) ASSIGN(SYSTEM-YEAR-NUM)
                          ELSE    ASSIGN(SYSTEM-YEAR-NUM - 1)
```

Since this report has the previous two months as the "across" group, we need two buckets for months (PREV1-AMT and PREV2-AMT). We compare the previous months' numeric month and year values to the sales date to determine which monthly bucket to assign the amount value to.

```
COMPUTE: RECORD-MONTH-NUM = #MONTHNUM(SALES-DATE)
COMPUTE: RECORD-YEAR-NUM  = #YEARNUM(SALES-DATE)
COMPUTE: PREV1-AMT = WHEN(RECORD-MONTH-NUM = PREV1-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV1-YEAR-NUM)
                     ASSIGN(AMOUNT)
COMPUTE: PREV2-AMT = WHEN(RECORD-MONTH-NUM = PREV2-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV2-YEAR-NUM)
                     ASSIGN(AMOUNT)
```

You will see in **Figure 72** that we also used two similar monthly "buckets" to hold the TAX data. This allows us to print two data values for each intersection in the crosstab report.

```
OPTIONS: SUMMARY
INPUT: SALES-FILE

COMPUTE: SYSTEM-MONTH-NUM = #MONTHNUM(#TODAY)
COMPUTE: PREV1-MONTH-NUM = WHEN(SYSTEM-MONTH-NUM = 1) ASSIGN(12)
                           ELSE    ASSIGN(SYSTEM-MONTH-NUM - 1)
COMPUTE: PREV2-MONTH-NUM = WHEN(PREV1-MONTH-NUM = 1) ASSIGN(12)
                           ELSE    ASSIGN(PREV1-MONTH-NUM - 1)

COMPUTE: SYSTEM-YEAR-NUM = #YEARNUM(#TODAY)
COMPUTE: PREV1-YEAR-NUM =
                  WHEN(SYSTEM-MONTH-NUM > 1) ASSIGN(SYSTEM-YEAR-NUM)
                  ELSE                       ASSIGN(SYSTEM-YEAR-NUM - 1)
COMPUTE: PREV2-YEAR-NUM =
                  WHEN(SYSTEM-MONTH-NUM > 2) ASSIGN(SYSTEM-YEAR-NUM)
                  ELSE                       ASSIGN(SYSTEM-YEAR-NUM - 1)

COMPUTE: RECORD-MONTH-NUM = #MONTHNUM(SALES-DATE)
COMPUTE: RECORD-YEAR-NUM  = #YEARNUM(SALES-DATE)

COMPUTE: PREV1-AMT = WHEN(RECORD-MONTH-NUM = PREV1-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV1-YEAR-NUM)
                     ASSIGN(AMOUNT)
COMPUTE: PREV2-AMT = WHEN(RECORD-MONTH-NUM = PREV2-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV2-YEAR-NUM)
                     ASSIGN(AMOUNT)

COMPUTE: PREV1-TAX = WHEN(RECORD-MONTH-NUM = PREV1-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV1-YEAR-NUM)
                     ASSIGN(TAX)
COMPUTE: PREV2-TAX = WHEN(RECORD-MONTH-NUM = PREV2-MONTH-NUM AND
                          RECORD-YEAR-NUM  = PREV2-YEAR-NUM)
                     ASSIGN(TAX)

COMPUTE: PREV1-DATE = #MAKEDATE(#FORMAT(PREV1-YEAR-NUM, P'9999') +
                                #FORMAT(PREV1-MONTH-NUM, P'99') + "01")
COMPUTE: PREV1-COL-HDG = #SUBSTR(#FORMAT(PREV1-DATE, SHORT2), 4, 8)

COMPUTE: PREV2-DATE = #MAKEDATE(#FORMAT(PREV2-YEAR-NUM, P'9999') +
                                #FORMAT(PREV2-MONTH-NUM, P'99') + "01")
COMPUTE: PREV2-COL-HDG = #SUBSTR(#FORMAT(PREV2-DATE, SHORT2), 4, 8)

TITLE: 'RUN:' #TODAY / 'CROSSTAB SALES REPORT' /
TITLE: 'PREVIOUS TWO MONTHS BY REGION'
TITLE: ' '
TITLE: 16 PREV1-COL-HDG 19 PREV2-COL-HDG /
TITLE: '        _________________________   _________________________' /
TITLE: 'REGION     AMOUNT         TAX          AMOUNT         TAX' /
TITLE: '______  ___________   _________   ___________   _________' /

SORT: REGION
BREAK: REGION NOTOTALS
 FOOTING(REGION PREV1-AMT(TOTAL) PREV1-TAX(TOTAL)
                PREV2-AMT(TOTAL) PREV2-TAX(TOTAL) )
BREAK: #GRAND NOTOTALS
 FOOTING('TOTAL' PREV1-AMT(TOTAL) PREV1-TAX(TOTAL)
                 PREV2-AMT(TOTAL) PREV2-TAX(TOTAL) )
```

Figure 72. Control statements to produce the crosstab report on page 222

In this report, we wanted to show the previous months' abbreviated *names* (and not just month numbers) in the column headings. To do that we used these compute statements:

```
COMPUTE: PREV1-DATE = #MAKEDATE(#FORMAT(PREV1-YEAR-NUM, P'9999') +
                                #FORMAT(PREV1-MONTH-NUM, P'99') + "01")
COMPUTE: PREV1-COL-HDG = #SUBSTR(#FORMAT(PREV1-DATE, SHORT2), 4, 8)
```

The first COMPUTE statement above creates a date field based on the previous month's year number, month number and a constant "01" for the day number. The second COMPUTE statement then formats this date into "DD MMM YYYY" format and extracts just the last 8 bytes. We used this field (and a similar one for the second previous month's name) in the TITLE statement for our column headings.

The NOTOTALS parm in the BREAK statement for REGION tells Spectrum Writer not to print its own default total line. Instead we printed our own custom footing line, using the FOOTING parm.

At the Grand Totals, we wanted to print the monthly totals for all the regions. So we added a BREAK statement for the special #GRAND control break. It is identical to the BREAK statement for the REGION break, except that it prints the word "TOTAL" instead of the name of a region.

Working With Multiple Input Files

The following sections discuss various topics involving runs that use multiple input files. The topics discussed are:

- how to use **multiple READ statements** for the same auxiliary input file (page 224)
- how to use a field from one auxiliary input file as the **READKEY for another auxiliary file** (page 226)
- how to assign and use **record names** (page 228)
- how **"missing" records** are handled (page 229)
- how to **test for missing records** (page 230)
- how **I/O errors** are handled (page 230)
- how to read records using **generic and KGE** (key greater than or equal) keys (page 230)
- how to perform **"one–to–many"** reads by reading more than one record for each READKEY value (page 232)

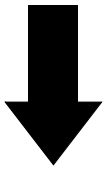
Using Multiple READ Statements for the Same File

This section explains:

- how to read more than one record (each with a different read key) from the same auxiliary input file

These Control Statements:

```
INPUT:   SALES-FILE
READ:    EMPL-FILE  READKEY(EMPL-NUM)         RECNAME(SALESMAN)
READ:    EMPL-FILE  READKEY(BACKUP-EMPL-NUM)  RECNAME(BACKUP)

COMPUTE: PRODKEY = 'P' + PRODUCT-CODE
READ:    PRODUCT-FILE  READKEY(PRODKEY)

TITLE:   'LISTING OF RECENT SALES, WITH BACKUP EMPLOYEE INFO'
COLUMNS: EMPL-NAME
         SALES-FILE.EMPL-NUM
         SALESMAN.HIRE-DATE
         BACKUP-EMPL-NUM
         BACKUP.LAST-NAME
         BACKUP.HIRE-DATE
         PRODUCT-CODE
         PRODUCT-DESC
```



Produce this Report:

```
                LISTING OF RECENT SALES, WITH BACKUP EMPLOYEE INFO


           SALES
           FILE  SALESMAN BACKUP     BACKUP       BACKUP
   EMPL    EMPL    HIRE    EMPL       LAST         HIRE   PRODUCT       PRODUCT
   NAME    NUM     DATE    NUM        NAME         DATE   CODE          DESC

JOHNSON     037  06/21/75  041   SIMPSON         12/01/82   952   PENCILS (NO. 1)
BAKER       044  06/04/82  045   THOMAS          06/04/82   978   HOLE PUNCHERS
MORRISON    042  11/30/79  036   JONES           01/31/80   907   INKPADS
MORRISON    042  11/30/79  045   THOMAS          06/04/82   919   GREEN PENS
SIMPSON     041  12/01/82  039   JOHNSON         11/25/79   916   RED PENS
JOHNSON     039  11/25/79  036   JONES           01/31/80   926   DESK CALENDARS
JOHNSON     039  11/25/79  044   BAKER           06/04/82   997   MAILING LABELS
BAKER       044  06/04/82  037   JOHNSON         06/21/75   916   RED PENS
THOMAS      045  06/04/82  037   JOHNSON         06/21/75   997   MAILING LABELS
JONES       036  01/31/80  042   MORRISON        11/30/79   977   PAPER CLIPS
JONES       036  01/31/80  039   JOHNSON         11/25/79   907   INKPADS
JONES       036  01/31/80  039   JOHNSON         11/25/79   977   PAPER CLIPS
JOHNSON     037  06/21/75  042   MORRISON        11/30/79   976   CHAIRS
SIMPSON     041  12/01/82  042   MORRISON        11/30/79   916   RED PENS


*** GRAND TOTAL (14 ITEMS)
```

Remarks:

- for every SALES–FILE record read, two records are read from the EMPL–FILE
- each EMPL–FILE record has a different name, assigned by the RECNAME parm (in the READ statement)
- the COLUMNS statement uses a record name to prefix each field name from the EMPL–FILE (to eliminate ambiguity)

Figure 73. A report with multiple READ statements for the same file

In Chapter 2, "How to Request a Report" we learned how to produce a report using two auxiliary input files. (See **Figure 20** on page 81.) We used two fields from the primary input file (SALES–FILE) as keys to read records from other files. The SALES–FILE contains yet another field that could be used as a read key for an auxiliary input file. That is the BACKUP–EMPL–NUM field, which is the employee number of the backup salesperson for a sale. This field can be used as a read key to the EMPL–FILE.

But our report already has one READ statement for the EMPL–FILE. That READ statement uses the EMPL–NUM field as the read key. This is no problem. Spectrum Writer allows you to have an unlimited number of READ statements for the same file. The sample report in **Figure 73** shows the addition of a second READ statement for the EMPL–FILE.

The second READ statement uses a different read key from the earlier READ statement, in order to read a different record from the EMPL–FILE. This means that two different EMPL–FILE records will be available for use in subsequent control statements. The first READ statement will read the employee file record for the main salesperson. The second READ statement will read the employee file record for the backup salesperson.

There is one thing to be careful about when you use more than one READ statement from the same file. All of the data fields from that auxiliary input file will now exist multiple times — once in each record. You can't simply specify HIRE–DATE, for example, in the COLUMNS statement now, because there are *two* such fields.

To solve this problem of ambiguous field names, we used the RECNAME parm in each of the READ statements for the EMPL–FILE. This parm assigns unique names to the two records. The record read using the EMPL–NUM field as the read key is named SALESMAN. The record read using the BACKUP–EMPL–NUM field as the read key is named BACKUP.

In the COLUMNS statement, we qualified all references to fields from the EMPL–FILE with one of these two record names. The use of the record name made it clear which record's data was intended in each instance.

How to Chain READ Statements

This section explains:

- how to use fields from one auxiliary input file to read a record from another auxiliary input file

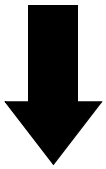
The sample report in the previous section used all of the fields in the primary input file that could be used as read keys to other files. But we can *still* read another record from an auxiliary input file. How? By using a field from an existing auxiliary input file as the key to *another* auxiliary input file. This is called "file chaining."

File chaining is when one auxiliary file contains the key used to read a record from another auxiliary input file, which may contain the key to yet another auxiliary input file, and so on. Spectrum Writer allows file chaining to any level.

Let's look at an example of file chaining. In the sample report in **Figure 73**, the EMPL–FILE is an auxiliary input file. The EMPL–FILE contains the address of the employee, including his 2–byte STATE. But the STATE field can be used as a key to read from another auxiliary input file — the STATE–FILE (described in Appendix F, "Files Used in Examples" on

These Control Statements:

```
INPUT:   SALES-FILE
READ:    EMPL-FILE   READKEY(EMPL-NUM)
READ:    STATE-FILE  READKEY(STATE)

TITLE:   'LISTING OF RECENT SALES'
TITLE:   'WITH EMPLOYEE ADDRESS INFORMATION'
COLUMNS:  EMPL-NAME
          CUSTOMER
          SALES-DATE
          SALES-FILE.EMPL-NUM('EMPL|NUM')
          CITY
          STATE
          STATE-NAME
```



Produce this Report:

```
                    LISTING OF RECENT SALES
                 WITH EMPLOYEE ADDRESS INFORMATION

   EMPL                      SALES   EMPL                          STATE
   NAME      CUSTOMER        DATE    NUM      CITY         STATE   NAME

JOHNSON    ACE ELECTRICAL  03/12/92 037  SCOTTSDALE       AZ   ARIZONA
BAKER      JACKS CAFE      03/26/92 044  WALNUT CREEK     CA   CALIFORNIA
MORRISON   STAR MARKET     03/29/92 042  GLENDALE         CA   CALIFORNIA
MORRISON   A1 PHOTOGRAPHY  03/30/92 042  GLENDALE         CA   CALIFORNIA
SIMPSON    EUROPEAN DELI   04/01/92 041  ARCADIA          CA   CALIFORNIA
JOHNSON    VILLA HOTEL     04/01/92 039  SANTA ROSA       CA   CALIFORNIA
JOHNSON    MARYS ANTIQUES  04/05/92 039  SANTA ROSA       CA   CALIFORNIA
BAKER      JACKS CAFE      04/12/92 044  WALNUT CREEK     CA   CALIFORNIA
THOMAS     YOGURT CITY     04/14/92 045  CONCORD          CA   CALIFORNIA
JONES      EZ GROCERY      04/15/92 036  SAN FRANCISCO    CA   CALIFORNIA
JONES      TOY TOWN        04/15/92 036  SAN FRANCISCO    CA   CALIFORNIA
JONES      TOY TOWN        04/15/92 036  SAN FRANCISCO    CA   CALIFORNIA
JOHNSON    ACME BUILDING   04/16/92 037  SCOTTSDALE       AZ   ARIZONA
SIMPSON    J & S LUMBER    04/30/92 041  ARCADIA          CA   CALIFORNIA


*** GRAND TOTAL (14 ITEMS)
```

Remarks:

- a record is read from the EMPL–FILE using the employee number from the primary input file as the key
- the STATE field from the EMPL–FILE is then used to read an additional record from the STATE–FILE
- an override column heading is specified for EMPL–NUM in the COLUMNS statement (for aesthetic purposes only)

Figure 74. A report with chained READ statements

page 648). By reading the STATE–FILE record we can obtain the full state name for use in our report. **Figure 74** shows a report that does this.

When chaining files, the order of the READ statements is important. Be sure to follow the rule that the READKEY field specified in each READ statement must already be available to Spectrum Writer in an existing input file record. For that reason, the READ statement to the EMPL–FILE must come before the READ statement to the STATE–FILE. The field used as the READKEY to the STATE–FILE isn't available until after the read to the EMPL–FILE.

How to Name the Input File Records

This section explains:

- what **record names** are
- the **default record name** assigned to each input file
- how to **assign your own record name** to an input file

Spectrum Writer assigns a name to the records that it reads from each input file. These are called **record names**. By default, records from a file are given the same name as the file itself. For example:

```
INPUT: SALES-FILE
```

Since no record name was explicitly stated in the above statement, the record name for records from the SALES–FILE file will also be "SALES–FILE."

Record names are necessary to distinguish between fields that have the same name but are in different input files (or, perhaps, different records from the same input file). For example, a field named EMPL–NUM exists in both the EMPL–FILE and in the SALES–FILE. If a particular report uses *both* of these files as inputs, simply specifying EMPL–NUM as a field name would be ambiguous. You would need to prefix EMPL–NUM with a record name to indicate which record's EMPL–NUM field you are referring to. (Prefixing a field name with a record name and a period is called **qualifying** a field name.) Consider the following statements:

```
INPUT:   SALES-FILE
READ:    EMPL-FILE  READKEY(EMPL-NUM)
COLUMNS: EMPL-NUM
         SALES-FILE.EMPL-NUM
         EMPL-FILE.EMPL-NUM
```

The above COLUMNS statement would have the following result. The first column (EMPL–NUM by itself) would result in an error message — the name is ambiguous since such a field exists in more than one of the input files. The first column in the report would contain only the "ambiguous reference" error indicator (that is, ***A***). The second column would contain the EMPL–NUM field from the SALES–FILE file, since the field name was qualified with that record name. The third column, similarly, would contain the EMPL–NUM field from the EMPL–FILE file.

> **Note:** you may be wondering why we did not need to use a record name in the READKEY parm of the above READ statement. The answer is that at that point in the control statements, only one EMPL-NUM field was known to Spectrum Writer (the one

in the SALES-FILE). It is only *after* the READ statement that more than one EMPL-NUM field are present in the input files.

If you want to specify a record name other than the file name, use the RECNAME parm of the INPUT or READ statement. For example:

```
INPUT: SALES-FILE RECNAME(SALESMAN)
```

The above statement would make SALESMAN the record name for the SALES–FILE file. To specify the EMPL–NUM from the SALES–FILE in this case, you would use:

```
COLUMNS: SALESMAN.EMPL-NUM
```

If you *do* specify a RECNAME parm (in an INPUT or READ statement), it is *not required* that you always use it when referring to fields from that file. Just use it whenever necessary to avoid ambiguity.

The ability to specify your own record names is especially important in reports where the same file is used in both the INPUT and a READ statement, or in multiple READ statements. In that case, since the same file is serving as multiple inputs to the report, just using the file name to qualify a field would still result in an ambiguous name.

You can qualify fields with record names in any control statement— not just the COLUMNS statement. Here are examples of qualifying field names in other control statements:

```
TITLE:     'EMPLOYEE DIRECTORY -- '  SALES-FILE.EMPL-NUM
COMPUTE:   MAILING-CODE = EMPL-FILE.EMPL-NUM + LAST-NAME
INCLUDEIF: EMPL-FILE.EMPL-NUM > '040'
```

The report in **Figure 73** (page 225) illustrates the use of record names.

How Missing Records Are Handled

Sometimes the auxiliary input file will not contain a record with a key equal to the read key's value. When this happens, Spectrum Writer assigns a default value to each of the fields in the "missing record." The default value depends on the data type of the field, as shown in the following table:

DEFAULT VALUES ASSIGNED TO FIELDS IN MISSING RECORDS

FIELD TYPE	DEFAULT VALUE
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

Testing for Missing Records

How can you tell when no record is found for a given read key? One easy way is to compare the contents of the key field (in the auxiliary input record) with the contents of the read key field (specified in the READKEY parm). If the values are not the same, it means that the record was missing (and thus its key field was assigned a default value of blanks).

Here is an example of testing for a successful read:

```
INPUT: SALES-FILE
READ:  EMPL-FILE READKEY(SALES-FILE.EMPL-NUM)
COMPUTE: EMPL-REC-FOUND = WHEN(EMPL-FILE.EMPL-NUM = SALES-FILE.EMPL-NUM) ASSIGN(#ON)
COMPUTE: START-DATE = WHEN(EMPL-REC-FOUND) ASSIGN(EMPL-FILE.HIRE-DATE)
                      ELSE                 ASSIGN(1/1/1990)
```

In the above statements, we set a bit field named EMPL-REC-FOUND to "on" (true) when the record from the EMPL-FILE is successfully read. (That is, when its key field equals the read key value.) This bit field is then used to help assign a value to a field called START-DATE. When the EMPL-FILE record is found, we assign its HIRE-DATE field to START-DATE. When the EMPL-FILE record is missing, we assign a default date of 1/1/1990 to START-DATE.

How I/O Errors Are Handled

By default, when an I/O error occurs on an auxiliary input file, Spectrum Writer prints a warning message in the control listing and then continues the run without reading from that file again. (All records from that file are treated as "missing".)

Sometimes, however, the data from the file in error may be so important to the report that it is pointless (or worse, misleading) to continue the run without it. Or, you may want to continue the run, but raise the job completion code to indicate that a problem exists. Use the READ statement's ONIOERROR parm for such situations.

Specify ONIOERROR(ERROR) to change the control listing message from a warning to an error (which also sets the job completion code to 8).

Or, specify ONIOERROR(STOP) to have Spectrum Writer halt the run immediately when an I/O error occurs on the file. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: "Missing" records are not considered I/O errors.

Note: You can also specify a ONIOERROR parm in an OPTIONS statement, if you want it to apply to *all* of the input files used in the run.

Using Generic and KGE Keys

This section explains:

- how to use **generic** READKEYs
- how to read records whose keys are **greater than or equal** to the READKEY

By default, Spectrum Writer assumes that the value in the READKEY parm specifies a complete, exact key. When performing the READ, Spectrum Writer looks for a record on the file that has that exact value as its full key. If no key in the file contains the exact value of the READKEY parm, the record is considered to be "missing."

Sometime you may know a portion, but not all, of the key in the record that you want to read. The READ statement has two parms that can be useful in such cases.

The **GENERIC parm** means that your READKEY value may be shorter than the key length defined for the VSAM file. Thus, it may not contain the complete key of the record you want to read, but only a leading portion of the desired key. When GENERIC is specified, Spectrum Writer reads the first record from the file which has an exact match on that portion of the key specified in the READKEY parm.

For example, assume a VSAM file contains records with the following 3–byte keys:

```
A01
A02
A16
A17
C01
C12
```

Given a file with the above keys, the READ statements below would give the result indicated in the following table.

RESULTS OF USING THE GENERIC PARM

STATEMENT	KEY OF RECORD READ
READ: FILE-X READKEY('A') GENERIC	**A01**
READ: FILE-X READKEY('A1') GENERIC	**A16**
READ: FILE-X READKEY('A13') GENERIC	**"missing"**
READ: FILE-X READKEY('B') GENERIC	**"missing"**

A related parm is the **KGE ("key greater or equal") parm**. This parm can be used with either a complete key or a generic key. It tells Spectrum Writer that, if no record on the file has a key (or partial key) that is exactly equal to the READKEY value, to use the first record whose key (or partial key) is greater than the READKEY value.Given a file with the same keys shown above, the READ statements below would give the indicated result:

RESULTS OF USING THE GENERIC AND KGE PARMS

STATEMENT	KEY OF RECORD READ
READ: FILE-X READKEY('A') GENERIC KGE	**A01**
READ: FILE-X READKEY('A1') GENERIC KGE	**A16**
READ: FILE-X READKEY('A13') GENERIC KGE	**A16**
READ: FILE-X READKEY('A13') KGE	**A16**
READ: FILE-X READKEY('B') GENERIC KGE	**C01**

Note: The GENERIC and KGE parms may only be used in READ statements that have a READKEY parm. Thus, they may not be used in READ statements for DB2 tables.

How to Perform "One–to–Many" Reads

This section explains:

- how to perform **"one–to–many"** reads by reading multiple records for a single READKEY value (or WHERE parm condition)

By default, each time Spectrum Writer reads a new record from the primary input file, it also attempts to read a single record from each file named in a READ statement.

However, there are times when there may be more than one record in an auxiliary input file for a given READKEY value. For example, this is often the case when reading from an **alternate index path** (where duplicate alternate key values can occur). Also, when using a **generic** READKEY there may be more than one record in a file that matches that generic key. And, when reading from a DB2 table, there may be more than one row that satisfies the conditions in your WHERE parm.

Use the MULTI parm in your READ statement if you want Spectrum Writer to read *all* of the records that match your READKEY value (or WHERE parm conditions). For example:

```
INPUT: EMPL-FILE
READ:  SALES-AIX  READKEY(EMPL-NUM)  MULTI
```

The INPUT statement above makes EMPL–FILE the primary input to our report. That file contains one record per employee. We then use a READ statement to read a record from the SALES–AIX file. The SALES–AIX file is actually a path to the SALES–FILE through an alternate index. The key for this alternate index is the EMPL–NUM field in the SALES–FILE. But we know that some employees have more than one record in the SALES–FILE. Without the MULTI parm, Spectrum Writer would simply read the first record for a given EMPL–NUM from the SALES–AIX file and use that record in the report. It would then proceed to read the next primary input file record and continue in the normal way.

By specifying the MULTI parm in the READ statement above, Spectrum Writer will now read *all* of the SALES–AIX records that match the EMPL–NUM from the EMPL–FILE record. The report in **Figure 75** uses the above statements.

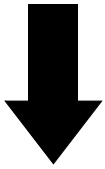
Here's how Spectrum Writer processed the input files in **Figure 75**. It reads the first record from the primary input file, EMPL–FILE. That record had an EMPL–NUM of 036.

Spectrum Writer then read the first record from the SALES–AIX file that had a key of 036. Using these two records as one "logical input record," Spectrum Writer then produced one line of the report.

Then, *before* reading the next record from the EMPL–FILE, Spectrum Writer read an additional record from the SALES–AIX file. It then used this "logical input record" (consisting of the original EMPL–FILE record and the second matching SALES–AIX record) in the report. This process continued until there were no more records in the SALES–AIX file with a key of 036. At that point, Spectrum Writer proceeded to read the next record from the primary input file. Using the EMPL–NUM from this new record (037), it then read each SALES–AIX file record with a key of 037, and so on.

These Control Statements:

```
INPUT:   EMPL-FILE
READ:    SALES-AIX  READKEY(EMPL-NUM)  MULTI
TITLE:   'EMPLOYEE LISTING, WITH RECENT SALES'
COLUMNS: LAST-NAME  FIRST-NAME  HIRE-DATE
         EMPL-FILE.EMPL-NUM
         SALES-AIX.EMPL-NUM
         SALES-DATE  AMOUNT
```



Produce this Report:

EMPLOYEE LISTING, WITH RECENT SALES

LAST NAME	FIRST NAME	HIRE DATE	EMPL FILE EMPL NUM	SALES AIX EMPL NUM	SALES DATE	AMOUNT
JONES	JERRY	01/31/80	036	036	04/15/95	10.25
JONES	JERRY	01/31/80	036	036	04/15/95	121.76
JONES	JERRY	01/31/80	036	036	04/15/95	10.25
JOHNSON	THOMAS	06/21/75	037	037	03/12/95	101.38
JOHNSON	THOMAS	06/21/75	037	037	04/16/95	500.00
JOHNSON	LINDA	11/25/79	039	039	04/01/95	234.45
JOHNSON	LINDA	11/25/79	039	039	04/05/95	9.98
MACDONALD	RICHARD	07/04/82	040		00/00/00	0.00
SIMPSON	TIMOTHY	12/01/82	041	041	04/01/95	14.99
SIMPSON	TIMOTHY	12/01/82	041	041	04/30/95	23.87
MORRISON	MICHAEL	11/30/79	042	042	03/29/95	44.35
MORRISON	MICHAEL	11/30/79	042	042	03/30/95	29.65
CHRISTOPHERSON	MELISSA	08/15/81	043		00/00/00	0.00
BAKER	VIVIAN	06/04/82	044	044	03/26/95	137.00
BAKER	VIVIAN	06/04/82	044	044	04/12/95	135.75
THOMAS	MARTIN	06/04/82	045	045	04/14/95	9.98

*** GRAND TOTAL (16 ITEMS) 1,383.66

Remarks:

- the MULTI parm in the READ statement causes Spectrum Writer to read multiple records from the SALES–AIX file for each record read from the EMPL–FILE

Figure 75. A "one-to-many" report using the MULTI parm in a READ statement

For a more complete description of how Spectrum Writer processes MULTI–type READ statements, see the Notes section of the READ statement in Chapter 10, "Control Statement Syntax" (page 591).

Speed-up Tip: READ statements with the MULTI parm are less efficient than regular READ statements. To reduce CPU and I/O usage, do not specify MULTI if you know that a file contains unique keys. (In other words, do not specify MULTI if you know the READKEY will only find one matching record in the file.)

Speed-up Tip: If you have some READ statements that use the MULTI parm and some that do not, put the READ statement(s) *without* the MULTI parm ahead of the other READ statements (when possible). In some cases this will reduce the amount of I/O that is performed.

Working with "Batched" Input Files

Some input files are organized as "batches" of data. Each batch begins with a header record and is followed by a number of detail records. A trailer record may also appear at the end each batch. The COMPUTE statement's RETAIN feature is useful when working with "batches" of records.

The RETAIN parm lets you save information from the header record in such files. You can then use this saved information along with the information in the detail records to produce your Spectrum Writer report or PC file.

Here is an example of using the RETAIN parm in a COMPUTE statements:

```
COMPUTE: SAVE-NAME = WHEN(REC-TYPE = 'A')  ASSIGN(EMP-NAME)
                     ELSE RETAIN
```

The above statement creates a new field called SAVE–NAME. As with all computed fields, Spectrum Writer assigns a value to SAVE–NAME each time it reads a new record from the input file. Assume that our input file has two types of records. The header records begin with an "A" in column 1. These header records contain the name of the employee whose data follows. The second type of record contains a "B" in column 1. These are the detail records. Each detail record contains the date and the amount of a sale made by the employee. When Spectrum Writer processes a header record, the WHEN condition in the above statement will be true (REC–TYPE will equal "A") and SAVE–NAME will be assigned the value of the EMP–NAME field. Otherwise (when the input record is a detail record), Spectrum Writer does not change the contents of the SAVE–NAME field. It just "retains" whatever value it already has. (Note that if ELSE RETAIN had not been specified, Spectrum Writer would set the SAVE–NAME field to blanks whenever the REC–TYPE field was not equal to "A".)

Figure 76 shows a sample "batch" type file with header and detail records. The lower box on that page shows the Spectrum Writer definition statements for the file. **Figure 77** (page 236) shows a PC file produced from this sample batch file.

Raw Input File:

```
AJOHNSON
B010492 1008.98
B033092  987.00
B050192  698.50
AMORRISON
B020892  345.99
B020992  900.17
ACLARK
B010192 1209.87
B022992  872.77
B060292  100.00
```

File Definition Statements for the Above File:

```
FILE: SALES-LOG DDNAME(SALELOG)

**** NOTE: THE FOLLOWING FIELD EXISTS IN ALL RECORD TYPES
FLD:  REC-TYPE  COL(1)  LEN(1)

**** NOTE: THE FOLLOWING FIELD EXISTS ONLY IN "A" RECORDS
FLD:  EMP-NAME  COL(2)  LEN(10)

**** NOTE: THE FOLLOWING FIELDS EXIST ONLY IN "B" RECORDS
FLD:  SALE-DATE COL(2)  TYPE(MMDDYY)
FLD:  SALE-AMT          TYPE(NUM)    LEN(8)  DEC(2)
```

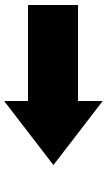
Remarks:

- The input file (shown in the top box) has two types of records
- Header records begin with the letter "A" and contain only an employee name
- Detail records begin with the letter "B" and contain the date and the amount of a sale
- Any number of detail records may follow a header record
- The Spectrum Writer definition statements (lower box) define the fields in both types of records
- Comment lines indicate which fields can be found in which records

Figure 76. An "batched" input file (with header and detail records) and its definition statements

These Control Statements:

```
OPTION:     PC
INPUT:      SALES-LOG
COMPUTE:    SAVE-NAME = WHEN(REC-TYPE = 'A') ASSIGN(EMP-NAME)
                        ELSE RETAIN
INCLUDEIF:  REC-TYPE = 'B'
COLUMNS:    SAVE-NAME  SALE-DATE  SALE-AMT
```



Produce this PC File:

```
"JOHNSON   ","01/04/92",      1008.98
"JOHNSON   ","03/30/92",       987.00
"JOHNSON   ","05/01/92",       698.50
"MORRISON  ","02/08/92",       345.99
"MORRISON  ","02/09/92",       900.17
"CLARK     ","01/01/92",      1209.87
"CLARK     ","02/29/92",       872.77
"CLARK     ","06/02/92",       100.00
```

Remarks:

- The PC file above contains one line for each detail record in the input file
- Each line includes "retained" data from the previous header record
- The COMPUTE statement saves the EMP–NAME field from the header records in a new field called SAVE–NAME
- The INCLUDEIF statement selects just the detail records to appear in the PC file
- The COLUMNS statement creates a column in the PC file for the SAVE–NAME field taken from the header record, as well as for the two fields from the detail records

Figure 77. A PC file produced from a batched input file (with header and details records)

Here are some general points to follow whenever using a header/detail type of input file:

- use FIELD statements to define all the fields in both the header records and the detail records. (Spectrum Writer allows you to define more than one field with the same starting column.)
- use one COMPUTE statement for each field that you want to retain from the header records.
  - use the WHEN parm to identify the header records in the input file
  - use the ASSIGN parm to name the header record field whose data you want to save
  - use ELSE RETAIN so that the field's value is not changed when the subsequent detail records are processed
- use an INCLUDEIF statement to select only the *detail* records for your Spectrum Writer report (or PC file). This is because you don't want to write out a report line containing only data from the header record. You just want to save data from the header records as they go by, and only write out report lines for each of the detail records in the input file. (Of course you can add further conditions to your INCLUDEIF statement if you want to include only certain detail records from the input file.)
- in your COLUMNS statement, you can refer to the retained data from the header records (that is, the COMPUTE fields) as well as all of the fields from the detail records
- note that information from any "trailer" record cannot be used with this technique. As the detail records are being processed, Spectrum Writer has not yet seen the trailer record. Therefore no data from that record is available. The conditions in the INCLUDEIF statement should ensure that the trailer records are not included in the report.

Working With Arrays

When working in a low level language, programmers often process arrays by using index variables, program loops, etc. As a non-procedural, 4GL report writer, Spectrum Writer does not have "procedural" elements such as "go to" or looping instructions. Instead, non-procedural methods are used to process arrays with Spectrum Writer.

Two methods for working with arrays are discussed:

- file "**normalization**" (below)
- **other strategies** that do not use normalization (page 249)

Using Normalization to Process Arrays

Often, the best way to process an array with Spectrum Writer is to "normalize" the array-containing records. During normalization, Spectrum Writer turns each *physical* input

record into one or more *logical* records. Each logical record corresponds to one "occurrence" of the array in the physical record. These logical records are all identical to the physical record, *except* for the part of the record that contains the first occurrence of the array. That part of the record is overlaid, successively, by the second, third, fourth, etc. occurrence of the array.

In effect, Spectrum Writer loops through the array for you, building logical records by moving, one at a time, each occurrence of the array into the first position. The result is that instead of a single record containing multiple occurrences of an array, you end up with multiple records that each contain a single occurrence of the array in a fixed location.

Figure 78 shows an example of a file that contains an array. As you can see in the Cobol record layout, the SALES-HISTORY record contains an array named SALE-ARRAY. The array holds information for up to six sales. The NUM-SLOTS field tells how many occurrences of the array are actually used in any particular record.

Cobol Definition of SALES-HISTORY File

```
01    SALES-HISTORY-REC
         05 NAME                  PIC X(10).
         05 CITY                  PIC X(10).
         05 NUM-SLOTS             PIC 9.
         05 SALE-ARRAY  OCCURS 6 TIMES.
            10 SALE-DATE          PIC 9(6).
            10 SALE-AMT           PIC 9(5)V9(2).
         05 RECORD-STATUS         PIC X(1).
```

Physical SALES-HISTORY File (Sales Data is in 6 Locations)

			(sale info)	(sale info)	(sale info)	(sale info)	(sale info)	(sale info)	
BAKER	BOSTON	2	9212010042398	9301040091225	0000000000000	0000000000000	0000000000000	0000000000000	A
CHAVEZ	MIAMI	1	9301250188901	0000000000000	0000000000000	0000000000000	0000000000000	0000000000000	B
JEFFERSON	CHICAGO	2	9301200066755	9301230044234	0000000000000	0000000000000	0000000000000	0000000000000	B
JOHNSON	DALLAS	5	9212300100810	9301020055475	9301100075065	9301110029980	9301190030162	0000000000000	A
JONES	ATLANTA	6	9212290071105	9212300019256	9301080109023	9301100052475	9301130078912	9301160120030	A
MORRISON	NEW YORK	3	9301020052200	9301040091944	9301060140246	0000000000000	0000000000000	0000000000000	B
SHARP	PORTLAND	1	9301310060019	0000000000000	0000000000000	0000000000000	0000000000000	0000000000000	A
SMITH	ST LOUIS	4	9301190033423	9301210070810	9301240100056	9301280020072	9301310094199	0000000000000	B

Figure 78. SALES-HISTORY file containing an array

Figure 79 shows the same file after it has been normalized. As you can see, the normalized file has more records than the physical file. The normalized file contains from 1 to 6 logical records for each physical record in the original file.

The normalized file in **Figure 79** contains exactly the same sales information as the physical file in **Figure 78**. The difference is this: in the normalized file, the information for all sales appears in one location — the first occurrence of the array (where the SALE-DATE and SALE-AMT fields are defined). Also, the unused occurrences of the array are eliminated from the normalized file (that is, no logical records are created for those occurrences).

File Definition and INPUT Statements to Normalize the SALES-HISTORY File

```
FILE:  SALES-HISTORY DDNAME(SALEHIST) LRECL(100)
FIELD: NAME          LENGTH(10)
FIELD: CITY          LENGTH(10)
FIELD: NUM-SLOTS     LENGTH(1)     TYPE(NUM)
FIELD: SALE-ARRAY    LENGTH(13)
FIELD: SALE-DATE     COLUMN(*-13)  TYPE(YYMMDD)
FIELD: SALE-AMT      LENGTH(7)     TYPE(NUM)     DEC(2)
FIELD: RECORD-STATUS COLUMN(SALE-ARRAY + 78) LENGTH(1)
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS)
```

Normalized (Logical) SALES-HISTORY File (Sales Data is in 1 Location)

			sale info	unused fields	
BAKER	BOSTON	2	9212010042398	9301040091225000000000000000000000000000000000000000000000000000000000000000	A
BAKER	BOSTON	2	9301040091225	9301040091225000000000000000000000000000000000000000000000000000000000000000	A
CHAVEZ	MIAMI	1	9301250188901	0000000000000000000000000000000000000000000000000000000000000000000000000000	B
JEFFERSON	CHICAGO	2	9301200066755	9301230044234000000000000000000000000000000000000000000000000000000000000000	B
JEFFERSON	CHICAGO	2	9301230044234	9301230044234000000000000000000000000000000000000000000000000000000000000000	B
JOHNSON	DALLAS	5	9212300100810	9301020055475930110007506593011100299809301190030162000000000000000	A
JOHNSON	DALLAS	5	9301020055475	9301020055475930110007506593011100299809301190030162000000000000000	A
JOHNSON	DALLAS	5	9301100075065	9301020055475930110007506593011100299809301190030162000000000000000	A
JOHNSON	DALLAS	5	9301110029980	9301020055475930110007506593011100299809301190030162000000000000000	A
JOHNSON	DALLAS	5	9301190030162	9301020055475930110007506593011100299809301190030162000000000000000	A
JONES	ATLANTA	6	9212290071105	9212300019256930108010902393011000524759301130078912930116012003 0	A
JONES	ATLANTA	6	9212300019256	9212300019256930108010902393011000524759301130078912930116012003 0	A
JONES	ATLANTA	6	9301080109023	9212300019256930108010902393011000524759301130078912930116012003 0	A
JONES	ATLANTA	6	9301100052475	9212300019256930108010902393011000524759301130078912930116012003 0	A
JONES	ATLANTA	6	9301130078912	9212300019256930108010902393011000524759301130078912930116012003 0	A
JONES	ATLANTA	6	9301160120030	9212300019256930108010902393011000524759301130078912930116012003 0	A
MORRISON	NEW YORK	3	9301020052200	9301040091944930106014024600000000000000000000000000000000000000000000000000	B
MORRISON	NEW YORK	3	9301040091944	9301040091944930106014024600000000000000000000000000000000000000000000000000	B
MORRISON	NEW YORK	3	9301060140246	9301040091944930106014024600000000000000000000000000000000000000000000000000	B
SHARP	PORTLAND	1	9301310060019	0000000000000000000000000000000000000000000000000000000000000000000000000000	A
SMITH	ST LOUIS	4	9301190033423	9301210070810930124010005693012800200720000000000000000000000000000000	B
SMITH	ST LOUIS	4	9301210070810	9301210070810930124010005693012800200720000000000000000000000000000000	B
SMITH	ST LOUIS	4	9301240100056	9301210070810930124010005693012800200720000000000000000000000000000000	B
SMITH	ST LOUIS	4	9301310094199	9301210070810930124010005693012800200720000000000000000000000000000000	B

Figure 79. Normalizing the SALES-HISTORY file

Once you have normalized a file, it is quite easy to process it with Spectrum Writer. You simply ignore the array (beyond the first occurrence). In the normalized file, there is only one occurrence of relevant data in each record, always in the same location.

For example, assume that we want a report that simply lists all of the sales over $100 in the SALES-HISTORY file. If we used the physical file, we would have to examine up to six different amount fields in each record. We might also need to check the NUM-SLOTS field to see which amount fields were actually used in a given record (unless we knew that all unused fields contained zeros). That would require six COMPUTE statements. Then, we would need six COLUMNS statements to potentially print each of the six amount fields, plus an option to suppress any zero lines, and so on. (The details of this approach can be found in "How to Print a Variable Number of Lines Per Input Record" on page 249.)

On the other hand, if we let Spectrum Writer normalize the file, we can easily produce the report using just the single SALE-DATE and SALE-AMT fields, like this:

```
INCLUDEIF: SALE-AMT > 100
COLUMNS:   NAME  SALE-DATE  SALE-AMT
```

Remember that every sale amount in the array of the original file now exists in the SALE-AMT field of some logical record. We simply include those records where that amount is over $100.

Figure 80 shows a report that uses the above statements.

> **Note:** A normalized file exists only as a temporary, logical file. During the run, the records are built, processed by Spectrum Writer, and then discarded. They are not actually written to a physical file.

The next section explains exactly how to use the NORMALIZE parm.

The NORMALIZE Parm

To normalize an array in an input file, Spectrum Writer must know three things about that array:

- the **starting column** of the array
- the **total size** of each occurrence of the array
- **how many** occurrences there are

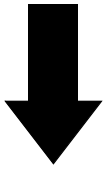
You supply this information by adding a NORMALIZE parm to the INPUT statement. The syntax of the NORMALIZE parm is:

```
NORMALIZE(normalize-field, occurs-expression [, ...] )
```

The **normalize-field** tells Spectrum Writer both the starting column of the array and the length of the array occurrences. Therefore, the normalize field must be a field that defines the *entire first occurrence* of the array that you want to normalize. In **Figure 79**, the normalize field is SALE-ARRAY. SALE-ARRAY is a 13-byte character field that includes both the SALE-DATE and SALE-AMT fields. Thus, it defines the entire first occurrence of the array.

These Control Statements:

```
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS)
TITLE: "SALES OVER $100"
INCLUDEIF: SALE-AMT > 100
COLUMNS: NAME  SALE-DATE  SALE-AMT
```



Produce this Report:

```
            SALES OVER $100

              SALE        SALE
   NAME       DATE         AMT
 BAKER      12/01/92          423.98
 BAKER      01/04/93          912.25
 CHAVEZ     01/25/93        1,889.01
 JEFFERSON  01/20/93          667.55
 JEFFERSON  01/23/93          442.34
 JOHNSON    12/30/92        1,008.10
 JOHNSON    01/02/93          554.75
 JOHNSON    01/10/93          750.65
 JOHNSON    01/11/93          299.80
 JOHNSON    01/19/93          301.62
 JONES      12/29/92          711.05
 JONES      12/30/92          192.56
 JONES      01/08/93        1,090.23
 JONES      01/10/93          524.75
 JONES      01/13/93          789.12
 JONES      01/16/93        1,200.30
 MORRISON   01/02/93          522.00
 MORRISON   01/04/93          919.44
 MORRISON   01/06/93        1,402.46
 SHARP      01/31/93          600.19
 SMITH      01/19/93          334.23
 SMITH      01/21/93          708.10
 SMITH      01/24/93        1,000.56
 SMITH      01/28/93          200.72

 *** GRAND TOTAL (    24 ITEMS)
                           17,445.76
```

Figure 80. A report that uses normalization to process an array

Note that in that example we could *not* use SALE-DATE as the normalize field. Doing so would yield incorrect results because it does not define the entire first occurrence of the array. It only defines the first six bytes of the first occurrence.

Why is the normalize field's length important? Because it tells Spectrum Writer how many bytes to move each time it builds a new logical record. It also tells Spectrum Writer where to start taking the new bytes from (namely, from the first byte following the normalize field).

The **occurs-expression** in the NORMALIZE parm tells Spectrum Writer how many occurrences of the array to process. It can be a constant numeric literal, the name of a numeric field, a numeric COMPUTE field, or any valid numerical expression.

Let's look at an example of the NORMALIZE parm in an INPUT statement:

```
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS)
```

The above statement tells Spectrum Writer to normalize the input records from the SALES-HISTORY file. The first occurrence of the array being normalized is defined by SALE-ARRAY. The number of occurrences to be used from the array is contained in the NUM-SLOTS field.

Figure 80 shows a report that uses this statement.

As records are read from the SALES-HISTORY file, here is what happens.

First, the unchanged physical record is processed by Spectrum Writer as usual. (That is, the INCLUDEIF tests are performed on it, and, if included, its contents are formatted into a line of the report.) This physical record is considered the first logical record.

Next, a second logical record is created by moving the 13 bytes immediately following the SALE-ARRAY field into the SALE-ARRAY area of the record. Now, SALE-DATE contains the date from the second occurrence of the array, and SALE-AMT contains the amount from the second occurrence of the array. This logical record is then processed just as if it had been read directly from the input file. (The INCLUDEIF tests are performed on it, and, if included, its contents are formatted into a line of the report.)

Then, a third logical record is created by moving the *next* 13 bytes into the SALE-ARRAY area. Now, SALE-DATE and SALE-AMT contain the date and the amount from the third occurrence of the array. This third logical record is then processed just as if it had been read directly from the input file, and so on

The number of logical records produced for each physical record is determined by the value of the NUM-SLOTS field (in this example). If NUM-SLOTS is 1 (or less), only the original, physical record is processed. If NUM-SLOTS contains 2, then two logical records are processed (the original physical record, plus one additional record). If NUM-SLOTS contains 3, then three logical records are processed, and so on.

> **Note:** You can also put the NORMALIZE parm directly in the FILE statement that defines a file. This is convenient when you know that you will *always* want to normalize a certain file. That way, you don't need to include the NORMALIZE parm in the INPUT statement every time you produce a report from that file. (In that case, you can still prevent normalization for individual runs by adding the NONORMALIZE parm to the INPUT statement.)

File Definition Tips for Records with Arrays

The FIELD statements that define an array will be different, depending on whether or not you will normalize that array. This section explains the differences.

Consider the SALES-HISTORY file, which contains an array with information for up to six sales. For runs where we normalize that file, we would use the file definition statements shown in **Figure 79** (page 239). But for runs where we do not normalize the file, we might prefer to use the file definition statements shown in **Figure 35** (page 250).

If you are *not* normalizing a record, and you want to access data from an array, you must define each occurrence of the array as a separate field. That is because the only way to access each occurrence of the array is to refer to it specifically by its unique field name. Thus, in **Figure 35** (page 250) each occurrence of the array in the SALES-HISTORY file is defined individually (SALE-DATE-1, SALE-AMT-1, SALE-DATE-2, SALE-AMT-2, etc.) This file definition is for use in reports where the SALES-HISTORY file is *not* normalized.

However, if you *are* using normalization to process an array, it is not necessary to define all of the occurrences of the array. It is sufficient to define just the fields in the first occurrence of the array. (Plus, you may need to define one additional field that includes the entire first occurrence of the array, for use in the NORMALIZATION parm itself.) You do not need to define the other occurrences of the array. This is because, after normalization, all array data will be located in the first occurrence of the array (of some logical record). Thus, if we know that the SALES-HISTORY file will always be normalized, we can define it as we did in **Figure 79** (page 239). It would not be necessary to define SALE-DATE-2, SALE-AMT-2, SALE-DATE-3 and so on.

Remember that if the array contains more than one field in each occurrence, you must also define a "high-level" field that defines the entire first occurrence of the array. This field will be needed in the NORMALIZATION parm. In the SALES-HISTORY file, we defined SALE-ARRAY as a high level field which includes both SALE-DATE and SALE-AMT. You should generally define the high-level field as a character field whose length is the length of the entire first occurrence of the array. That is, its length will be the sum of the lengths of each individual field. (Of course, if the array only consists of a single field, then no additional high-level field is needed. You can use the single field itself in the NORMALIZE parm.)

> **Caution:** Cobol sometimes adds what are called "slack bytes" at the end of each array occurrence. This is done in order to align the next occurrence on a halfword or fullword boundary. This may happen if a field in your Cobol record layout uses the SYNCHRONIZED parm. Any such slack bytes must be included in the length of the high-level field you define.

After defining the high-level field, you can use a COLUMN(*–nnn) parm on the next FIELD statement to "back up" again to the beginning of the array. You can then start "redefining" the lower level fields in that same portion of the record.

Of course, you may want to define your array so that it can be used either with or without normalization. In that case, define all of the occurrences of the array, as well as one high-level field for the entire first occurrence.

Normalizing Nested Arrays

Some records contain "arrays within arrays." Such arrays are also called "nested arrays." Spectrum Writer is able to normalize records containing any degree of nested arrays.

Specify one NORMALIZE parm for each array level. The first NORMALIZE parm defines the outermost array. The next NORMALIZE parm defines the next deeper array, and so on.

For example, consider this Cobol record layout:

```
01 RECORD.
     05 EMPL-NAME          PIC X(20).
     05 SALE-ARRAY OCCURS 10 TIMES.
        10 SALE-DATE       PIC 9(6).
        10 SALE-CUSTOMER   PIC X(10).
        10 SALE-PRODUCT-CODE OCCURS 5 TIMES PIC X(3).
     05 RECORD-STATUS      PIC X(1).
```

The above record contains a nested array. The outer array (SALE-ARRAY) contains an inner array (SALE-PRODUCT-CODE). We could define this record to Spectrum Writer in the following way:

```
FIELD: EMPL-NAME           LENGTH(20)
FIELD: SALE-ARRAY          LENGTH(31)
FIELD: SALE-DATE           COLUMN(* — 31)  TYPE(YYMMDD)
FIELD: SALE-CUSTOMER       LENGTH(10)
FIELD: SALE-PRODUCT-CODE   LENGTH(3)
*
FIELD: RECORD-STATUS       COLUMN(SALE-ARRAY + 310) LENGTH(1)
```

Since we will be normalizing this file, it is sufficient to define only the first occurrence of each array. (This is because, once normalized, the relevant data in each logical record will be in that first occurrence location.)

As you can see, we first defined a field that includes the whole, first 31-byte occurrence of the outer array (SALE-ARRAY). We then backed up 31 bytes in order to define the individual fields within that occurrence. First we defined the 6-byte SALE-DATE field, then the 10-byte SALE-CUSTOMER field. The last field within the 31-byte SALE-ARRAY field is the SALE-PRODUCT-CODE array (a total of 15 bytes). Again, it was only necessary to define the first occurrence of that array. And, since the SALE-PRODUCT-CODE array does not have multiple fields per occurrence, it was not necessary to define a higher-level field to define its entire first occurrence. The 3-byte SALE-PRODUCT-CODE field itself defines the entire first occurrence of that inner array.

After defining the first occurrence of each array, we defined the RECORD-STATUS field. We use the COLUMN(SALE-ARRAY + 310) parm to locate this field in the correct column. This COLUMN parm tells Spectrum Writer to locate the RECORD-STATUS field 310 bytes after the start of the SALE-ARRAY field. (Ten occurrences of the 31-byte SALE-ARRAY field is a total of 310 bytes.)

After defining our file as above, we can normalize it like this:

```
INPUT: OUR-FILE
       NORMALIZE(SALE-ARRAY, 10)
       NORMALIZE(SALE-PRODUCT-CODE, 5)
```

As Spectrum Writer normalizes the input records, it first "loops through" the most deeply nested array (the one specified in the last NORMALIZE parm). That is the SALE-PRODUCT-

CODE array in this example. Thus, the first logical record, as always, is the unchanged physical record. The second logical record will have the second product code in SALE-PRODUCT-CODE. The third logical record will have the third product code in SALE-PRODUCT-CODE, and so on for the fourth and fifth logical records.

Then, having fully normalized the inner array for the first occurrence of the outer array, Spectrum Writer begins normalizing the outer array (SALE-ARRAY). Thus, for the sixth logical record, Spectrum Writer moves the 31 bytes following the SALE-ARRAY field into the SALE-ARRAY location. That is, it moves the second occurrence of the outer array into the first occurrence's location. This 31-bytes includes the second occurrence of the SALE-DATE and SALE-CUSTOMER fields. It also includes the entire 15-byte SALE-PRODUCT-CODE array from the second occurrence of SALE-ARRAY. At this point, the SALE-PRODUCT-CODE field contains the first occurrence of that array (within the second occurrence of the outer array). Using Cobol notation, we could say it contains SALE-PRODUCT-CODE (2, 1). After processing this logical record, Spectrum Writer continues to fully normalize the inner array. Thus, for the next logical records, Spectrum Writer moves the 2nd, 3rd, 4th and 5th occurrences of the inner array to SALE-PRODUCT-CODE.

After that, Spectrum Writer moves the third occurrence of the outer array (SALE-ARRAY) into the SALE-ARRAY field, and so on.

Each time one inner level array has been fully normalized, the next higher array level is adjusted and all lower levels are normalized all over again.

Spectrum Writer continues normalization in this manner until the last occurrence of the inner array has been normalized for the last occurrence of the outer array. Thus, each physical record in this example results in 50 logical input records (the 5 occurrences of the inner array times the 10 occurrences of the outer array).

Normalizing Multiple, Non-Nested Arrays

Some records contain multiple arrays that are *not* nested. That is, there may be two or more independent arrays in a record. For example, consider this Cobol record layout:

```
01 RECORD.
     05 EMPL-NAME                      PIC X(20).
     05 SALE-DATE      OCCURS 10 TIMES  PIC 9(6).
     05 HIRE-DATE                      PIC 9(6).
     05 SALE-CUSTOMER  OCCURS 10 TIMES  PIC X(10).
     05 RECORD-STATUS                  PIC X(1).
```

In this record layout, both SALE-DATE and SALE-CUSTOMER are arrays. But they are not nested. We could define this record to Spectrum Writer in the following way:

```
FIELD: EMPL-NAME        LENGTH(20)
FIELD: SALE-DATE        TYPE(YYMMDD)
FIELD: HIRE-DATE        COLUMN(*+54)  TYPE(YYMMDD)
FIELD: SALE-CUSTOMER    LENGTH(10)
FIELD: RECORD-STATUS    COLUMN(*+90)  LENGTH(1)
```

Note again that when normalizing a file, we only need to define the first occurrence of each array. Thus we defined just the first SALE-DATE field at the beginning of that array. Since this is an array of just a single field, we did not need to define a separate "high level" field. The SALE-DATE field itself defines the entire first occurrence of the array.

We used the COLUMN(*+54) parm on the HIRE-DATE field to skip over the other 9 undefined occurrences of the 6-byte sales date field. (We could also have specified COLUMN(SALE-DATE + 60). Either method will properly locate the HIRE-DATE field.)

Similarly, we defined only the first occurrence of the SALE-CUSTOMER array.

We used the COLUMN(*+90) parm on the RECORD-STATUS field to skip over the other nine undefined occurrences of the 10-byte customer field.

Now that we have defined the file, how do we normalize it? There are two different ways to normalize records that contain non-nested arrays. The method you choose will depend on the logical relationship between the data in the two arrays.

Often, the data in the two arrays will have a one-to-one relationship. That is, the first date in the SALE-DATE array will correspond to the first customer in the SALE-CUSTOMER array. The second date in the SALE-DATE array will correspond to the second customer in the SALE-CUSTOMER array, and so on.

If this is the case, you want to normalize the two arrays in parallel. You can think of it as "stepping through" both arrays in sync. To normalize two or more arrays in parallel, specify all of the arrays in a *single* NORMALIZE parm:

```
INPUT: OUR-FILE
       NORMALIZE(SALE-DATE, 10, SALE-CUSTOMER, 10)
```

When performing this normalization, the first logical record, as always, will be the unchanged physical record. (The first date is in SALE-DATE and the first customer is in SALE-CUSTOMER.) The second logical record will have the second date in SALE-DATE and the second customer in SALE-CUSTOMER. The third logical record will have the third date in SALE-DATE and the third customer in SALE-CUSTOMER, and so on. If you normalize the above file in this manner, each physical record will result in 10 logical records.

On the other hand, you may have two separate arrays in a record whose data is *not* related in the manner described above. In other words, *all* occurrences of the first array may apply to *all* occurrences of the second array. In that case, you would normalize them *as if* they were nested (even though they are not physically nested). That will cause one logical record to be created for every possible combination of items from the two arrays. To normalize in this manner, use two separate NORMALIZE parms (just as for true nested arrays):

```
INPUT: OUR-FILE
       NORMALIZE(SALE-DATE, 10)
       NORMALIZE(SALE-CUSTOMER, 10)
```

When performing this normalization, the first logical record is, as always, the unchanged physical record. (The first date is in SALE-DATE and the first customer is in SALE-CUSTOMER.) The second logical record will retain the first date in SALE-DATE, but move the second customer to SALE-CUSTOMER. The third logical record again retains the first date in SALE-DATE and now has the third customer in SALE-CUSTOMER. Thus, the first ten logical records all have the first date in SALE-DATE, while the SALE-CUSTOMER array is normalized.

Next (for the eleventh logical record), the second date is moved to SALE-DATE and SALE-CUSTOMER is re-initialized to contain the first customer. The next logical record has the second date and the second customer. The next one has the second date and the third customer, and so on.

When normalized in this way, each physical record results in 100 logical records. (Ten occurrences of the first array times the ten occurrences of the second array.)

You can specify the two NORMALIZE parms in either order (since the arrays are not physically nested). The only difference will be the order in which the logical records are built. Spectrum Writer always normalizes the last NORMALIZE parm first.

Normalizing only Certain Records

Some files contain more than one type of record. For example, a file may contain a combination of header records and detail records. Perhaps the detail records contain an array that must be normalized, while the header records do not contain that array. Or, the header records might contain an entirely different array in a different location.

For such files, you need **conditional normalization**. Spectrum Writer provides the NORMWHEN parm to perform conditional normalization.

When a NORMALIZE parm is present in an INPUT or READ statement, the default is for Spectrum Writer to normalize *all* of the records from that file. You can, however, put a NORMWHEN parm ahead of the NORMALIZE parm(s). In that case, the normalization is done only on records where the condition specified in the NORMWHEN parm is true. For example:

```
INPUT: BATCH-FILE
       NORMWHEN(RECORD-TYPE = 'HDR')
         NORMALIZE(STATUS-ARRAY, 5)
       NORMWHEN(RECORD-TYPE = 'DET')
         NORMALIZE(CUSTOMER-ARRAY, 8)
```

The above statements tell Spectrum Writer to normalize the STATUS-ARRAY only for those physical records where the RECORD-TYPE field contains "HDR". And the CUSTOMER-ARRAY will be normalized only for those physical records where the RECORD-TYPE field contains "DET". Records with any other value in the RECORD-TYPE field will not be normalized at all. (That is, only the physical records themselves will be processed.)

Each NORMWHEN parm governs the NORMALIZE parm(s) that follow it (until the next NORMWHEN parm, if any). Spectrum Writer first tests the condition in the first NORMWHEN parm. If true, it performs the complete normalization specified in the following NORMALIZE parm(s). After that normalization, Spectrum Writer then tests the condition in the next NORMWHEN parm. If true, Spectrum Writer then performs the normalization specified by the following NORMALIZE parm(s), and so on.

If the conditions in multiple NORMWHEN parms are true, *each* of the corresponding normalizations will be performed on that record. When a record is normalized multiple times, the unchanged physical record is processed only one time (not one time per normalization).

Any valid conditional expression is allowed within the NORMWHEN parm.

If any NORMALIZE parms precede the first NORMWHEN parm, their normalization will be performed on every record.

Normalizing an Auxiliary Input File

If the records read from an auxiliary input file contain an array, you may want to normalize those records as well. Do that by adding the necessary NORMWHEN and/or NORMALIZE parms to your READ statement.

Remember that, by default, a READ statement only returns a single record (the first record whose key matches the READKEY value). Therefore to successfully normalize an auxiliary input file, you must also specify the MULTI parm in the READ statement. The MULTI parm tells Spectrum Writer to use *all* of the records from the file whose key matches the READKEY value. The MULTI parm allows all of the logical records created during the normalization process to be used in the report.

Normalization Errors

If Spectrum Writer detects erroneous normalization information in a record, it does not normalize the field in question for that record. (If normalization was requested for more than one field, Spectrum Writer will still attempt to perform the other normalization(s).) Examples of normalization errors are:

- an error occurs while trying to compute the "occurs" value. For example, a numeric field involved in the computation might contain a non-numeric value.
- the occurs expression results in a value that is negative or zero.
- the occurs expression results in a value that is too big. That is, the last occurrence of the array would be beyond the end of the record area.

When Spectrum Writer encounters any of these normalization errors, it prints a message in the control listing, along with a "dump" (hex listing) of the record in question. By default, only the first ten such normalization errors are printed. You can use the MAXNORMDUMP option (in an OPTIONS statement) to print more (or fewer) such messages. The syntax of the MAXNORMDUMP option is:

```
OPTIONS: MAXNORMDUMP(nnnnn/10)
```

By default, normalization error messages are treated as informational messages only. When a normalization error occurs, Spectrum Writer processes the physical record, and then skips the normalization in question for that record. If you want normalization errors to be treated as more serious errors, use the ONNORMERROR option (in an INPUT, READ or OPTIONS statement). The syntax of the ONNORMERROR option is:

```
ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)
```

For example, to stop the entire run (with a "user ABEND") if a normalization error occurs, specify:

```
 OPTION: ONNORMERROR(STOP)
```

How to Print a Variable Number of Lines Per Input Record

If you choose not to use normalization to process an array, there is another approach that you may find useful. It involves the **SKIPZERODET option**. The SKIPZERODET option tells Spectrum Writer to skip (that is, to not write out) any detail lines that contain only zero values. Let's look at how this option can be used.

Consider the sample SALES–HISTORY file shown in **Figure 35**. This file contains 3 fields in fixed positions (the name, the city, and a numeric field that tells how many sales "slots" are used in the following array). After these three fields there is an array containing sales information for up to six sales. Each of these occurrences of the array (or "slots") contains the date and the amount of one sale. But not all six slots are actually filled in for each record. As you can see, some records have only 1 slot filled in. Others have two or three. One record has all six filled in. The unused slots within a record contain zeros.

Our goal is to produce a report that shows all the sales made by each employee. But we do *not* want to see all the unused (or "zero") sales. We'll consider two different strategies to accomplish this objective.

> **Note:** An entirely different approach to this task was discussed in "Using Normalization to Process Arrays" (page 237).

Variable Number of Lines — Strategy 1

Let's start by seeing what our report would look like if we did nothing to remove the "zero" sales fields. We'll use one COLUMNS statement for the constant information in each record (the name and city). Then we will use one additional COLUMNS statement for each of the 6 sales slots, showing the date and amount of a sale. If we do nothing else, Spectrum Writer will always print 7 lines for each input record (one line per COLUMNS statement). The resulting report is shown in **Figure 36** (page 251). It isn't very attractive. It also wastes a lot of paper showing sales data for non–existent sales.

The first strategy to remove the "zero" sales data from the report is this: *simply specify the SKIPZERODET option*. This causes Spectrum Writer to skip (suppress) all detail report lines (or PC file records) that contains only zeros. In our sample report, this means that the lines for unused sales slots (lines with only a zero date and a zero amount) will be suppressed. The report now contains only the lines that actually have real sales data in them. The report in **Figure 37** (page 252) illustrates this strategy. (Note that we also specified the DOUBLE option to double–space the report, making it easier to read.)

Once again, the SKIPZERODET option simply means that a detail line will not be output if it contains only "zero" items. The following are considered "zero" items for this purpose:

- blanks (for character fields)
- zero numeric values (0, 0.00, etc.)
- 00/00/00 (zero dates)
- 00:00:00 (zero times)

File Definition Statements for SALES-HISTORY File:

```
FILE:  SALES-HISTORY DDNAME(SALEHIST) LRECL(100)
FIELD: NAME          LEN(10)
FIELD: CITY          LEN(10)
FIELD: NUM-SLOTS     LEN(1) TYPE(NUM)
FIELD: SALE-DATE-1           TYPE(YYMMDD)
FIELD: SALE-AMT-1    LEN(7) TYPE(NUM)     DEC(2)
FIELD: SALE-DATE-2           TYPE(YYMMDD)
FIELD: SALE-AMT-2    LEN(7) TYPE(NUM)     DEC(2)
FIELD: SALE-DATE-3           TYPE(YYMMDD)
FIELD: SALE-AMT-3    LEN(7) TYPE(NUM)     DEC(2)
FIELD: SALE-DATE-4           TYPE(YYMMDD)
FIELD: SALE-AMT-4    LEN(7) TYPE(NUM)     DEC(2)
FIELD: SALE-DATE-5           TYPE(YYMMDD)
FIELD: SALE-AMT-5    LEN(7) TYPE(NUM)     DEC(2)
FIELD: SALE-DATE-6           TYPE(YYMMDD)
FIELD: SALE-AMT-6    LEN(7) TYPE(NUM)     DEC(2)
FIELD: RECORD-STATUS LEN(1)
```

Contents of SALES-HISTORY File:

```
BAKER     BOSTON    292120100423989301040091225000000000000000000000000000000000000000000000000000000000000A
CHAVEZ    MIAMI     193012501889010000000000000000000000000000000000000000000000000000000000000000000000000B
JEFFERSON CHICAGO   293012000667559301230044234000000000000000000000000000000000000000000000000000000000000B
JOHNSON   DALLAS    592123001008109301020055475930110007506593011100299809301190030162000000000000A
JONES     ATLANTA   692122900711059212300019256930108010902393011000524759301130078912930116012003 0A
MORRISON  NEW YORK  393010200522009301040091944930106014024600000000000000000000000000000000000000B
SHARP     PORTLAND  193013100600190000000000000000000000000000000000000000000000000000000000000000000A
SMITH     ST LOUIS  493011900334239301210070810930124010005693012800200720000000000000000000000000B
```

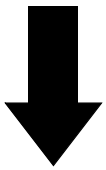
Remarks:

- this "Sales History" file contains 100–byte records
- each record contains: the salesperson's name, city, and information about up to 6 sales
- each "sales slot" in the record consists of a sales date and a sales amount
- a one–byte field after the city tells how many slots are in use
- unused slots contain all zeros

Figure 35. A sample file containing sales data for up to 6 sales per record

These Control Statements:

```
INPUT:    SALES-HISTORY
COLUMNS:  NAME  CITY
COLUMNS:  SALE-DATE1  SALE-AMT-1
COLUMNS:  SALE-DATE2  SALE-AMT-2
COLUMNS:  SALE-DATE3  SALE-AMT-3
COLUMNS:  SALE-DATE4  SALE-AMT-4
COLUMNS:  SALE-DATE5  SALE-AMT-5
COLUMNS:  SALE-DATE6  SALE-AMT-6
```



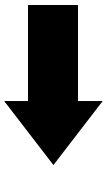
Produce this Report:

```
MON  01/23/95    1:53 PM   DATA FROM SALES-HISTORY   PAGE     1

 BAKER      BOSTON
 12/01/92          423.98
 01/04/93          912.25
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 CHAVEZ     MIAMI
 01/25/93        1,889.01
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 JEFFERSON  CHICAGO
 01/20/93          667.55
 01/23/93          442.34
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 00/00/00            0.00
 JOHNSON    DALLAS
 12/30/92        1,008.10
 01/02/93          554.75
 01/10/93          750.65
 01/11/93          299.80
 01/19/93          301.62
 00/00/00            0.00

                 (other report lines not shown)
```

Figure 36. A report with “no strategy” to deal with unused array items

These Control Statements:

```
OPTIONS:  SKIPZERODET  DOUBLE
INPUT:    SALES-HISTORY
COLUMNS:  NAME  CITY
COLUMNS:  SALE-DATE1  SALE-AMT-1
COLUMNS:  SALE-DATE2  SALE-AMT-2
COLUMNS:  SALE-DATE3  SALE-AMT-3
COLUMNS:  SALE-DATE4  SALE-AMT-4
COLUMNS:  SALE-DATE5  SALE-AMT-5
COLUMNS:  SALE-DATE6  SALE-AMT-6
```



Produce this Report:

```
MON  01/23/95    2:31 PM   DATA FROM SALES-HISTORY   PAGE     1

 BAKER      BOSTON
 12/01/92          423.98
 01/04/93          912.25

 CHAVEZ     MIAMI
 01/25/93        1,889.01

 JEFFERSON  CHICAGO
 01/20/93          667.55
 01/23/93          442.34

 JOHNSON    DALLAS
 12/30/92        1,008.10
 01/02/93          554.75
 01/10/93          750.65
 01/11/93          299.80
 01/19/93          301.62

 JONES      ATLANTA
 12/29/92          711.05
 12/30/92          192.56
 01/08/93        1,090.23
 01/10/93          524.75
 01/13/93          789.12
 01/16/93        1,200.30

 MORRISON   NEW YORK
 01/02/93          522.00
 01/04/93          919.44
 01/06/93        1,402.46

              (other report lines not shown)
```

Figure 37. Strategy 1 — just add the SKIPZERODET option

Note: For the purposes of this option, "detail lines" means: the lines printed for each individual input record (COLUMNS statement lines); the total lines printed at control breaks (if any); and the Grand Totals lines (if any). Title lines, column heading lines and break heading lines are not affected by this option.

Note: Only the first 256 bytes of each line are examined when checking for zero detail lines. This is generally not a problem, since detail lines are usually not this long.

Note: A related option named SKIPBLANKDET is also available. It suppresses lines only when they are completely blank. It is for occasions when you want to suppress blank detail lines, but still print lines that have zeros in them.

Of course, there are many variations that you can use with this technique. For example, you might want to include the data from the first sale in the first COLUMNS statement (along with the constant information). Then you would just have 5 additional COLUMNS statements for the remaining 5 sales slots.

```
COLUMNS: NAME CITY  SALE-DATE-1  SALE-AMT-1
COLUMNS: 22         SALE-DATE-2  SAME-AMT-2
COLUMNS: 22         SALE-DATE-3  SAME-AMT-3
COLUMNS: 22         SALE-DATE-4  SAME-AMT-4
COLUMNS: 22         SALE-DATE-5  SAME-AMT-5
COLUMNS: 22         SALE-DATE-6  SAME-AMT-6
```

Or, you might want to combine two or more sales on each COLUMNS statement. For example:

```
COLUMNS: NAME  CITY
COLUMNS: SALE-DATE-1  SAME-AMT-1  SALE-DATE-2  SAME-AMT-2
COLUMNS: SALE-DATE-3  SAME-AMT-3  SALE-DATE-4  SAME-AMT-4
COLUMNS: SALE-DATE-5  SAME-AMT-5  SALE-DATE-6  SAME-AMT-6
```

This will take up less space in your report. And again, any line with only "zero" information in it will be suppressed. Of course, you could still end up with a line that has good sales information for one sale, and zero data for the other sale on that line. See "Putting a Variable Number of Items on a Single Line" on page 257 for a solution to that problem.

There is another option that may also be useful in reports such as these. It is the SPLITDETAIL option. It allows Spectrum Writer to split the detail lines for a single input record across pages in the report. If you do not specify this option, Spectrum Writer will skip to a new page if the current page does not have enough room to show *all of* the detail lines for an input record. For example, if a record from the SALES–HISTORY file had all six sales filled in, it would require seven report lines in the example on page 252. When printing that record's data, Spectrum Writer would skip to the next page if there were not seven lines left in the current page.

Normally you will probably not use SPLITDETAIL, since it is easier to view related data when it is all on a single page. But that does use extra paper. And, it may be impractical if you are listing 30 or 40 items from each input record, since virtually every record would end up requiring a new page. In these cases, you may specify SPLITDETAIL to allow Spectrum Writer to fill up each page before going on to the next page of the report.

Note: Remember that any time multiple COLUMNS statements are specified, Spectrum Writer does not produce column headings (by default). Use the

MULTICOLHDG option if you want the column headings for the first COLUMNS statement to appear in the report.

Note: This technique (unlike the next one discussed) did not require use of the NUM–SLOTS field at all. As long as your unused data contains only zeros or blanks, you can use Strategy 1 even when there is no field that explicitly tells you how many occurrences in an array are being used.

Variable Number of Lines — Strategy 2

The technique discussed above (Strategy 1) is the easiest way to suppress unwanted lines from your report or PC file. But it only works as long as your unused "slots" always contain valid zero values (for numeric, date and time fields) and blanks (for character fields). In some cases, your unused slots may contain "low–values" or some other kind of invalid data.

Note: If you *know* that the unused fields in your input record will contain invalid data, you can just use the ZEROINVDATA option. That option causes fields with invalid data to be treated as if they contained zeros. That will enable the SKIPZERODET option to work for you as described under Strategy 1 above.

There may be cases when it is not safe to treat *all* invalid values as zeros. Or, the unused fields in your record may contain something other than invalid values (such as all 9's). In such cases, you can use Strategy 2.

Strategy 2 also uses the SKIPZERODET option. But in this case, we don't use the fields from the actual input record in the COLUMNS statements (since those fields might contain invalid data). Instead, we create a set of corresponding COMPUTE fields, which we use in the COLUMNS statements. Each COMPUTE field will be assigned one of two values:

1. The value from its corresponding record field (when that field contains "good data"), or
2. A zero value (if the corresponding record field does not contain "good data").

We use conditional COMPUTE statements to selectively move data from just the filled–in sales "slots" to this set of corresponding COMPUTE fields. The COMPUTE statement will contain a WHEN condition so that the record value is only assigned to the compute field when the record value contains good data. Otherwise, no WHEN condition will be true and the COMPUTE field will be assigned a default value of zeros.

We create one COMPUTE statement for each field which might potentially not be used. In our present example, we create a COMPUTE field for each of the six date and amount fields:

```
COMPUTE: S-DATE-1 = WHEN(NUM-SLOTS >= 1)  ASSIGN(SALE-DATE-1)
COMPUTE: S-AMT-1  = WHEN(NUM-SLOTS >= 1)  ASSIGN(SALE-AMT-1)

COMPUTE: S-DATE-2 = WHEN(NUM-SLOTS >= 2)  ASSIGN(SALE-DATE-2)
COMPUTE: S-AMT-2  = WHEN(NUM-SLOTS >= 2)  ASSIGN(SALE-AMT-2)

...
COMPUTE: S-DATE-6 = WHEN(NUM-SLOTS >= 6)  ASSIGN(SALE-DATE-6)
COMPUTE: S-AMT-6  = WHEN(NUM-SLOTS >= 6)  ASSIGN(SALE-AMT-6)
```

In the above statements, we used the NUM–SLOTS field to determine whether a particular sales slot has good data or not. (In the SALES–HISTORY file, NUM–SLOTS is used like an OCCURS DEPENDING ON variable in Cobol that tells how many slots in the sales array are in use.)

The first COMPUTE statement above will assign the SALE–DATE–1 value to the COMPUTE field named S–DATE–1 *only if* the first slot is actually used. (That is, only if NUM–SLOTS is at least 1.) If NUM–SLOTS is zero, then S–DATE–1 will be assigned a zero date value. (That is the default value assigned when no WHEN conditions are met.) The next statement does the same thing for the amount value in the first slot. It assigns the record's value to S–AMT–1 *only if* the first slot was actually used. Otherwise, S–AMT–1 will be assigned a value of zero.

We do the same thing for the second sales slot. If NUM–SLOTS is at least 2, we assign the sales date and amount from the second slot to S–DATE–2 and S–AMT–2. Otherwise, S-DATE–2 and S–AMT–2 remain zero. And so on with slots 3 through 6.

In our COLUMNS statement, we now use these COMPUTE fields rather than the actual fields from the input record. That is because we know for sure that our COMPUTE fields contain either valid sales information or zeros. Thus, the SKIPZERODET option will work just fine.

```
COLUMNS: NAME  CITY
COLUMNS: S-DATE-1  S-AMT-1
COLUMNS: S-DATE-2  S-AMT-2
COLUMNS: S-DATE-3  S-AMT-3
COLUMNS: S-DATE-4  S-AMT-4
COLUMNS: S-DATE-5  S-AMT-5
COLUMNS: S-DATE-6  S-AMT-6
```

You can also use a similar technique to assign constant "line identifier" values to each line of your report or PC file. For example, let's assume that you want the words "SALE 1:" to appear beside the values for the first sale. You can't just put that literal on the COLUMNS statement, because then that report line would *never* be all blanks and zeros, and therefore would never be suppressed. (It would always say "SALE 1:", which is not blanks or zeros.) Instead, conditionally assign your literal text to a COMPUTE field the same way you do the other data. Assign the literal value to the compute field *only* when the related sales data is present:

```
COMPUTE: SALES-ID-1 = WHEN(NUM-SLOTS >= 1)  ASSIGN('SALE 1:')
COMPUTE: SALES-ID-2 = WHEN(NUM-SLOTS >= 2)  ASSIGN('SALE 2:')
COMPUTE: SALES-ID-3 = WHEN(NUM-SLOTS >= 3)  ASSIGN('SALE 3:')
COMPUTE: SALES-ID-4 = WHEN(NUM-SLOTS >= 4)  ASSIGN('SALE 4:')
COMPUTE: SALES-ID-5 = WHEN(NUM-SLOTS >= 5)  ASSIGN('SALE 5:')
COMPUTE: SALES-ID-6 = WHEN(NUM-SLOTS >= 6)  ASSIGN('SALE 6:')
```

The "SALE–ID" fields computed above will be blank when the associated sales fields are not used. Use these COMPUTE fields in your COLUMNS statement.

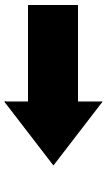
```
COLUMNS: SALE-ID-1  SALE-DATE-1  SALE-AMT-1
COLUMNS: SALE-ID-2  SALE-DATE-2  SALE-AMT-2
...
```

Your report line will still result in only blanks and zeros for sales slots that are not used. Such lines will not print in the report. But for slots containing a sales value, the SALE–ID field will contain the desired literal value and will appear before the sales amount in the report. The report in **Figure 38** illustrates this.

What if your record does not contain a numeric field that tells you how many slots are used? More than likely you can still use this technique. You will just need to find another

These Control Statements:

```
OPTIONS:  SKIPZERODET  DOUBLE
INPUT:    SALES-HISTORY

COMPUTE:  SALES-ID-1 = WHEN(NUM-SLOTS >= 1) ASSIGN('SALE 1:')
COMPUTE:  SALES-ID-2 = WHEN(NUM-SLOTS >= 2) ASSIGN('SALE 2:')
COMPUTE:  SALES-ID-3 = WHEN(NUM-SLOTS >= 3) ASSIGN('SALE 3:')
COMPUTE:  SALES-ID-4 = WHEN(NUM-SLOTS >= 4) ASSIGN('SALE 4:')
COMPUTE:  SALES-ID-5 = WHEN(NUM-SLOTS >= 5) ASSIGN('SALE 5:')
COMPUTE:  SALES-ID-6 = WHEN(NUM-SLOTS >= 6) ASSIGN('SALE 6:')

COLUMNS:  NAME  CITY
COLUMNS:  SALE-ID-1  SALE-DATE1  SALE-AMT-1
COLUMNS:  SALE-ID-2  SALE-DATE2  SALE-AMT-2
COLUMNS:  SALE-ID-3  SALE-DATE3  SALE-AMT-3
COLUMNS:  SALE-ID-4  SALE-DATE4  SALE-AMT-4
COLUMNS:  SALE-ID-5  SALE-DATE5  SALE-AMT-5
COLUMNS:  SALE-ID-6  SALE-DATE6  SALE-AMT-6
```



Produce this Report:

```
MON  01/23/95    2:01 PM   DATA FROM SALES-HISTORY   PAGE     1

BAKER      BOSTON
SALE 1: 12/01/92          423.98
SALE 2: 01/04/93          912.25

CHAVEZ     MIAMI
SALE 1: 01/25/93        1,889.01

JEFFERSON  CHICAGO
SALE 1: 01/20/93          667.55
SALE 2: 01/23/93          442.34

JOHNSON    DALLAS
SALE 1: 12/30/92        1,008.10
SALE 2: 01/02/93          554.75
SALE 3: 01/10/93          750.65
SALE 4: 01/11/93          299.80
SALE 5: 01/19/93          301.62

JONES      ATLANTA
SALE 1: 12/29/92          711.05
SALE 2: 12/30/92          192.56
SALE 3: 01/08/93        1,090.23
SALE 4: 01/10/93          524.75
SALE 5: 01/13/93          789.12
SALE 6: 01/16/93        1,200.30

                    (other report lines not shown)
```

Figure 38. Adding literal identifiers to variable lines

way of determining whether a slot is filled in or not. For example, if there is a character field within each slot, you might be able to compare it to blanks to see if the whole slot is in use or not. If our file had a Customer Name field within each sales slot, we could test that field like this:

```
COMPUTE: S-DATE-1 = WHEN(SALE-CUSTOMER-NAME-1 ¬= ' ')  ASSIGN(SALE-DATE-1)
COMPUTE: S-AMT-1  = WHEN(SALE-CUSTOMER-NAME-1 ¬= ' ')  ASSIGN(SALE-AMT-1)
COMPUTE: S-DATE-2 = WHEN(SALE-CUSTOMER-NAME-2 ¬= ' ')  ASSIGN(SALE-DATE-2)
COMPUTE: S-AMT-2  = WHEN(SALE-CUSTOMER-NAME-2 ¬= ' ')  ASSIGN(SALE-AMT-2)
...
COLUMNS: SALE-CUSTOMER-NAME-1  S-DATE-1  S-AMT-1
COLUMNS: SALE-CUSTOMER-NAME-2  S-DATE-2  S-AMT-2
...
```

If there is no character field for you to test, you may be able to test the date or amount field. For example, if your unused slots are filled with hex zeros (which is "invalid data" for the numeric amount fields), you could use these COMPUTE statements:

```
COMPUTE: SLOT-1-USED = #ISNUM(SALE-AMT-1)
COMPUTE: S-DATE-1 = WHEN(SLOT-1-USED)  ASSIGN(SALE-DATE-1)
COMPUTE: S-AMT-1  = WHEN(SLOT-1-USED)  ASSIGN(SALE-AMT-1)
```

The first COMPUTE statement above sets SLOT-1-USED to "true" only when there is a valid numeric value in SALE-AMT-1. The WHEN parms in the next two statements then test this boolean result to know whether to copy the SALE-DATE-1 and SALE-AMT-1 data to the S-DATE-1 and S-AMT-1 fields.

Putting a Variable Number of Items on a Single Line

The methods just discussed work by suppressing output lines that contain only zero or blank data. To use these methods, you generally must put each element of your array on a separate line. But what if you want to put multiple array elements on a single report line (or PC file record) and *not* see a lot of zeros for the unused slots? Here is a technique for doing that.

This technique is similar to strategy 2 above in that we use a COMPUTE statement for each record field which may or may not be filled in.

```
COMPUTE: S-DATE-1 = WHEN(SALE-CUSTOMER-NAME-1 ¬= ' ')  ASSIGN(SALE-DATE-1)
COMPUTE: S-AMT-1  = WHEN(SALE-CUSTOMER-NAME-1 ¬= ' ')  ASSIGN(SALE-AMT-1)
COMPUTE: S-DATE-2 = WHEN(SALE-CUSTOMER-NAME-2 ¬= ' ')  ASSIGN(SALE-DATE-2)
COMPUTE: S-AMT-2  = WHEN(SALE-CUSTOMER-NAME-2 ¬= ' ')  ASSIGN(SALE-AMT-2)
...
```

You can then list as many of these COMPUTE fields as you want in a single COLUMNS statement. By using the BIZ ("blank if zero") parm, we ensure that all unused fields appear as blanks in the output line:

```
COLUMNS: NAME CITY S-DATE-1(BIZ) S-AMT-1(BIZ) S-DATE-2(BIZ) S-AMT-2(BIZ)
COLUMNS: 22        S-DATE-3(BIZ) S-AMT-3(BIZ) S-DATE-4(BIZ) S-AMT-4(BIZ)
COLUMNS: 22        S-DATE-5(BIZ) S-AMT-5(BIZ) S-DATE-6(BIZ) S-AMT-6(BIZ)
```

Now your report will show the date and amount of each sales slot that was filled in in the input record. Blanks will appear for unused slots. And, as long as you use the SKIPZERODET (or SKIPBLANKDET) option, any line that contains *only* blanks will still be suppressed altogether.

Creating PC Files from Non-Spectrum Writer Reports

This section shows how to:

- turn **existing mainframe reports** (not created by Spectrum Writer) into PC files for your favorite PC program
- how to use the **RETAIN parm** in the COMPUTE statement

Normally Spectrum Writer creates PC files from the data in mainframe *files*. Sometimes, however, the data you want to download may not be in a file, but in a *report* already produced on your mainframe. Perhaps someone in your shop must manually key data from such a report into a PC spreadsheet. Spectrum Writer can let you automate that process, increasing accuracy and saving hours of manual work.

The first step is to write your existing report to a *file* (rather than to a printer). Then simply define this "report file" to Spectrum Writer as if it were any other input file. Consider the sample mainframe report in **Figure 39**. This is an accounts payable report. It lists each cost center's outstanding invoices, including such information as the invoice number, the customer number, the date the invoice is due and the amount due. When defining this report as a file to Spectrum Writer we can say that an INVOICE–NUM field begins in column 2 and is 6 bytes long. Then, the CUST–NUM field starts in column 11 and is 4 bytes long. And we can define the CUSTOMER, DUE–DATE, and AMOUNT fields similarly. **Figure 40** shows Spectrum Writer definition statements for this sample report. (We'll explain shortly the other fields defined in that Figure.)

Now let's consider some unique situations that arise when we use reports as input files:

- The first column in each report line usually contains a "carriage control" character. This character is normally hidden from you when you view reports online or have them printed on paper. However, this character must be taken into account when specifying a field's beginning column. So when defining a report's fields, remember that what you normally think of as the first column in a report is actually column 2. In the report in **Figure 39** we have shown the carriage control characters. They are the characters "1", "0" and " " that you see in the first column of each report line.
- Report files usually contain some lines which you'll want to completely ignore. These lines do not contain any data that you want to download to the PC. For example, in the report in **Figure 39** we would want to completely ignore:
  - the first title line on each page ("ABC COMPANY...")
  - the column heading lines
  - the cost center total lines (we can always use Spectrum Writer to compute the totals if we want them in our PC file)
  - and all blank lines (such as those between the title line and the column headings)

  We'll see shortly how to use the INCLUDEIF statement to have Spectrum Writer ignore certain lines in your report.

```
1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75...80
1ABC COMPANY -- ACCOUNTS PAYABLE LISTING                    RUN DATE:  01/31/95
 ITEMS FOR COST CENTER: 501 - ACCOUNTING                              PAGE:   1

0INVOICE  CUST.
 NUMBER   NUMBER    CUSTOMER         DATE DUE  AMOUNT DUE

 18003A   2987      PIP PRINTING     02/15/95     $245.78
 209812   1098      FEDEX            02/08/95       90.12
 N/A      1167      A1 ACCOUNTING    02/28/95    1,030.75
                           COST CENTER TOTAL  $1,366.65



1ABC COMPANY -- ACCOUNTS PAYABLE LISTING                    RUN DATE:  01/31/95
 ITEMS FOR COST CENTER: 502 - OPERATIONS                              PAGE:   2

0INVOICE  CUST.
 NUMBER   NUMBER    CUSTOMER         DATE DUE  AMOUNT DUE

 66761    2013      ACME CATERER     03/05/95     $200.00
 AB0291   0889      AT&T             02/01/95      676.99
                           COST CENTER TOTAL    $876.99



1ABC COMPANY -- ACCOUNTS PAYABLE LISTING                    RUN DATE:  01/31/95
 ITEMS FOR COST CENTER: 504 - PERSONNEL                               PAGE:   3

0INVOICE  CUST.
 NUMBER   NUMBER    CUSTOMER         DATE DUE  AMOUNT DUE

 787611   1292      GAS COMPANY      02/20/95     $192.10
 898-1    0987      FAST TRAVEL      02/03/95      972.00
 K00921   1200      CITIBANK         02/27/95     2987.11
 18021A   2987      PIP PRINTING     02/19/95       21.78
                           COST CENTER TOTAL  $4,172.99
```

Figure 39. A typical mainframe report that has been written to a disk file

```
  FILE: AP-REPORT DDNAME(REPORTIN)
  *
  *** FOLLOWING TEST FIELDS ARE USED TO DETERMINE TYPE OF RECORD
  FLD:  COL40             COL(40)  LEN(1)
  FLD:  COLS2-THRU-6      COL(2)   LEN(5)
  *
  *** FOLLOWING FIELDS ARE ONLY IN THE 2ND TITLE LINE OF REPORT
  FLD:  TITLE-COST-CENTER COL(25)  LEN(3)
  FLD:  TITLE-CC-NAME     COL(31)  LEN(10)
  *
  *** FOLLOWING FIELDS ARE ONLY IN THE DETAIL LINES OF REPORT
  FLD:  INVOICE-NUM       COL(2)   LEN(6)
  FLD:  CUST-NUM          COL(11)  LEN(4)
  FLD:  CUSTOMER          COL(21)  LEN(13)
  FLD:  DUE-DATE          COL(38)           TYPE(MM-DD-YY)
  FLD:  AMOUNT            COL(48)  LEN(10)  TYPE(NUM)  DEC(2)
```

Figure 40. Spectrum Writer statements to define the “report file” shown above

- Other report lines may contain data which applies to all of the other report lines on the same page. An example of such data in our sample report in **Figure 39** is the cost center number and the cost center name which appear in the second title line of each page. (For example, "ITEMS FOR COST CENTER: 501 – ACCOUNTING".) This cost center information is printed only once per page. It does not appear in each detail report line. This kind of data from title lines must be "retained" so that it is available along with the detail line's data when Spectrum Writer writes each record to the PC file. We'll see how to use COMPUTE statements to retain data from title lines.

Now let's look at how to handle each of these special situations when creating PC files from reports.

How to Ignore Certain Report Lines

The INCLUDEIF statement tells Spectrum Writer which records from the input file to include in the PC file. When using report files for input, we use the INCLUDEIF statement to identify just those report lines that actually contain the data we need in our PC file— that is, the detail report lines. By examining the different lines in your report (the title lines, the column heading lines, blank lines, total lines and detail lines) you should be able to come up with a conditional expression that selects only the detail lines. For the sample report in **Figure 39**, an easy way to do that is with the following statement:

```
INCLUDEIF: COL40 = '/'
```

The above statement tells Spectrum Writer to include report records in the PC file only if the field named COL40 contains a slash. (Note in the file definition statements in **Figure 40** that we defined COL40 as a 1–byte field at column 40.) In looking at the report, you'll notice that only the detail lines contain a slash in column 40 (as part of the Date Due value). The titles, column headings, blank lines, etc. will all be excluded from the PC file since none of those lines contains a slash in column 40.

Your report may not have such a unique identifying character in its detail lines. In that case you will need to use more than one test in your INCLUDEIF statement. For example, if the report in **Figure 39** had not had a date field with a slash in it, we might have used the following statement instead:

```
INCLUDEIF: COL55 = '.'  AND  COL-2-THRU-6 ¬= '     '
```

The above statement selects the detail records by examining what they have in 2 places. Report lines must have a decimal point in column 55 (where the Amount field appears). That test alone, however, would also include the total lines since they have a decimal point in column 55 too. We do not want to include total lines in our PC file because they do not contain the other fields we need (such as invoice number, customer number, etc.) The second test requires that columns 2 through 6 not contain blanks. The detail lines will pass this test (since they have Invoice Numbers in those columns), while the total lines (which have blanks in those columns) will not pass the test. So, the only records which do contain a decimal point in column 55 and do not contain blanks in columns 2 through 6 are our report detail records.

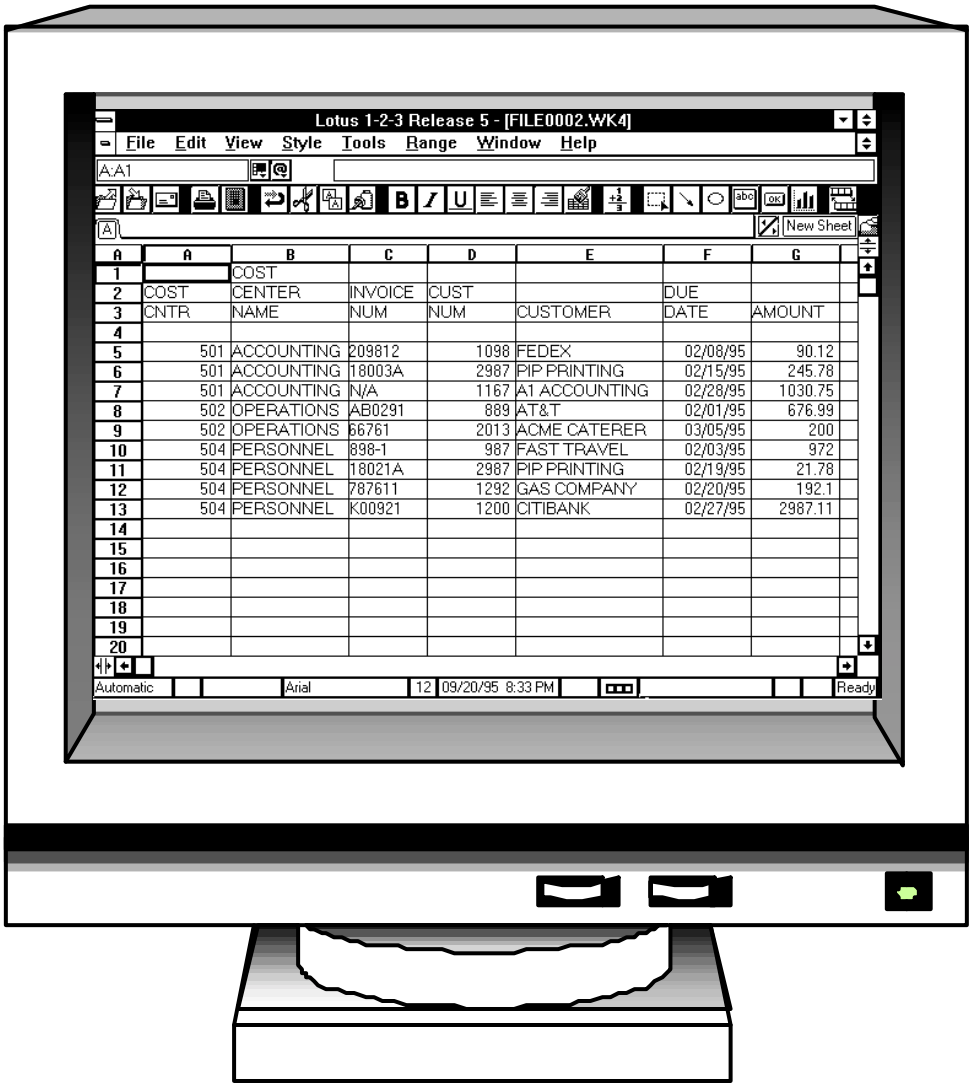
How to Retain Data from Report Titles

We saw in the preceding section how to eliminate the title and other unwanted lines from our PC file and include only the detail lines. But what if the report titles contain some data that we want to download to the PC along with the data in the detail lines? To do this we need for Spectrum Writer to capture data from the title lines as they are processed and

These Control Statements:

```
OPTION:      LOTUS
INPUT:       AP-REPORT
COMPUTE:     COST-CNTR =      WHEN(COLS2-THRU-6 = 'ITEMS') ASSIGN(TITLE-COST-CENTER)
                              ELSE RETAIN
COMPUTE:     COST-CNTR-NAME = WHEN(COLS2-THRU-6 = 'ITEMS') ASSIGN(TITLE-CC-NAME)
                              ELSE RETAIN
INCLUDEIF:   COL40 = '/'
COLUMNS:     COST-CNTR  COST-CNTR-NAME  INVOICE-NUM  CUST-NUM
             CUSTOMER  DUE-DATE  AMOUNT
SORT:        COST-CNTR  DUE-DATE
```

Result in this Lotus 1-2-3 Spreadsheet



	A	B	C	D	E	F	G
1		COST					
2	COST	CENTER	INVOICE	CUST		DUE	
3	CNTR	NAME	NUM	NUM	CUSTOMER	DATE	AMOUNT
4							
5	501	ACCOUNTING	209812	1098	FEDEX	02/08/95	90.12
6	501	ACCOUNTING	18003A	2987	PIP PRINTING	02/15/95	245.78
7	501	ACCOUNTING	N/A	1167	A1 ACCOUNTING	02/28/95	1030.75
8	502	OPERATIONS	AB0291	889	AT&T	02/01/95	676.99
9	502	OPERATIONS	66761	2013	ACME CATERER	03/05/95	200
10	504	PERSONNEL	898-1	987	FAST TRAVEL	02/03/95	972
11	504	PERSONNEL	18021A	2987	PIP PRINTING	02/19/95	21.78
12	504	PERSONNEL	787611	1292	GAS COMPANY	02/20/95	192.1
13	504	PERSONNEL	K00921	1200	CITIBANK	02/27/95	2987.11
14							
15							
16							
17							
18							
19							
20							

Figure 41. Creating a Lotus 1-2-3 spreadsheet from a mainframe report

"retain" that data until it comes to the detail lines. We use a COMPUTE statement with the RETAIN option to accomplish this. For example, to retain the cost center from the second title line in our report we could use this statement:

```
COMPUTE: COST-CNTR = WHEN(COL-2-THRU-6 = 'ITEMS')  ASSIGN(TITLE-COST-CENTER)
                     ELSE RETAIN
```

The statement above is a "conditional" COMPUTE statement. That is, the value assigned to COST–CNTR depends on a logical condition. In this case, when columns 2 through 6 of the report line contain "ITEMS" (that is, when the input record being processed is the second title line of a page), we assign the value of TITLE–COST–CENTER (in columns 25 though 27 of the report) to our new field. When processing any input record other than the second title line, this new field will simply retain its current value. That is, it will retain the value of the Cost Center from the most recent title line processed. We also use a similar COMPUTE statement to retain the cost center *name* from the same title line:

```
COMPUTE: COST-CNTR-NAME = WHEN(COL-2-THRU-6 = 'ITEMS')  ASSIGN(TITLE-CC-NAME)
                          ELSE RETAIN
```

Now we can use these two retained fields in our COLUMNS statement to create columns in our PC file containing the cost center and the cost center name. For example:

```
COLUMNS: COST-CNTR  COST-CNTR-NAME  INVOICE-NUM  CUST-NUM  CUSTOMER  ...
```

Why couldn't we simply put TITLE–COST–CNTR and TITLE–CC–NAME directly in our COLUMNS statement? Remember that our INCLUDEIF statement is written to include only the detail report records in our PC file. And the cost center is not present in the detail records. The columns where the cost center appears in the title lines contain other data in the detail lines. If we specified TITLE–COST–CNTR in our COLUMNS statement, we would just get "garbage" in our PC file.

You may wonder why we couldn't "include" *both* title lines and detail lines in the PC file to solve this problem. The answer is that the title lines don't contain the *other* information needed in the PC file (such as invoice number, customer number, etc.) If we included title records, the TITLE–COST–CNTR data would look just fine in our PC file, but the INVOICE–NUM and other fields would then contain "garbage."

The correct way to use data from both titles and detail lines is to "include" only the detail records, and use COMPUTE statements to save data from the title lines as they are read. Then we use that saved title data along with the data in the detail lines to write our PC file records. By using the techniques discussed in this section, you can apply all of Spectrum Writer's extracting and PC–formatting power to the existing reports in your shop.

Figure 41 shows an actual example of creating a Lotus 1–2–3 spreadsheet from the report shown in **Figure 39** (page 259). Notice that we had Spectrum Writer re-sort the PC file into cost center and due date order.

Working with SMF Records

You can use Spectrum Writer to produce many useful reports from your shop's SMF files. In addition, Spectrum Writer can also turn your SMF data into PC files, letting you work with extracted SMF data in your favorite PC spreadsheet program. This section provides some tips on using Spectrum Writer with SMF files.

The SMF files are among the most complicated files in any shop. But Spectrum Writer makes it easy to produce reports from them. Here are some specific points to keep in mind when dealing with SMF files. Some of these points are illustrated in the SMF file definition statements shown in **Figure 42**.

- SMF records can be *big*. So to be safe, specify Spectrum Writer's largest LRECL value (32,767) when defining the file. Do this in either the FILE statement or the INPUT statement. For example:

  ```
  FILE: SMF  DDNAME(SMF)  LRECL(32767)
  ```

  This will ensure that Spectrum Writer allocates an I/O area big enough to handle the largest SMF records.

- You should not need to specify DCB information in your DD statement. Spectrum Writer gets this information from the file's label. If you do give explicit DCB information, be sure your LRECL and BLKSIZE values are correct for the input file.

- Spectrum Writer normally ignores the 4–byte RDW (record descriptor word) at the beginning of variable–length records (such as SMF records). That is, Spectrum Writer considers "column 1" of the SMF record to be the first byte after the RDW. If you prefer to include the RDW as part of the input record, specify the KEEPRDW option. Do this in either the FILE statement, the INPUT statement, or an OPTIONS statement. For example:

  ```
  FILE: SMF  DDNAME(SMF)  LRECL(32767)  KEEPRDW
  ```

  Note: When KEEPRDW is specified, "column 1" of the SMF record becomes the first byte of the RDW. One reason you may want to specify KEEPRDW is to use the field offsets listed in the SMF manual as a guide when writing your FIELD statements. The SMF manual gives field offsets relative to the beginning of the RDW.

- When defining fields to Spectrum Writer, you can use either the COLUMN parm or the DISP (DISPLACEMENT) parm to specify where a field begins in a record. Since the SMF manual indicates field locations as offsets (displacements), it's generally more convenient to use the DISP parm in your FIELD statements.

  ```
  FIELD: REC-TYPE  DISP(5)  LENGTH(1)  TYPE(BIN)  NOACC
  ```

- Spectrum Writer has a number of date and time "data types" that are especially intended for use with SMF data. Use these in the TYPE parm of your FIELD statements to define SMF dates and times. Some common data types for SMF records are:

  P–CYYDDD This is a packed Julian date which includes a single–digit century indicator. Most SMF dates are stored in this format (written

```
FILE: SMF DDNAME(SMF) LRECL(32767) KEEPRDW
**
** SMF HEADER FIELDS FOLLOW
FLD: REC-LEN                TYPE(HALFWORD)
FLD: REC-TYPE     DISP(5)   TYPE(BIN)        LEN(1) NOACC
FLD: SMF-TIME               TYPE(B-SECS)     DEC(2) LEN(4)
FLD: SMF-DATE               TYPE(P-CYYDDD)
FLD: SUB-TYPE     DISP(22)  TYPE(HALFWORD)
**
** OFFSET AND LENGTH INFO FOR SELECTED SECTIONS IN TYPE 30 REC
FLD: ID-OFFSET    DISP(32)  TYPE(FULLWORD)
FLD: ID-LEN                 TYPE(HALFWORD)
FLD: ID-NUM                 TYPE(HALFWORD)
FLD: IO-OFFSET              TYPE(FULLWORD)
FLD: IO-LEN                 TYPE(HALFWORD)
FLD: IO-NUM                 TYPE(HALFWORD)
FLD: COMP-OFFSET            TYPE(FULLWORD)
FLD: COMP-LEN               TYPE(HALFWORD)
FLD: COMP-NUM               TYPE(HALFWORD)
FLD: PROC-OFFSET            TYPE(FULLWORD)
FLD: PROC-LEN               TYPE(HALFWORD)
FLD: PROC-NUM               TYPE(HALFWORD)
**
** SELECTED FIELDS FROM THE ID SECTION
FLD: JOBNAME                LEN(8)   OFFSET(ID-OFFSET)
FLD: PGMNAME                LEN(8)
FLD: STEPNAME               LEN(8)
FLD: USERID                 LEN(8)
FLD: JES-JOBID              LEN(8)
FLD: STEP-NUM               TYPE(HALFWORD)      NOACC
FLD: JOB-CLASS              LEN(1)
FLD: DEV-ALLOC-TIME         TYPE(B-SECS) DEC(2) LEN(4) DISP(*+5)
FLD: PGM-START-TIME         TYPE(B-SECS) DEC(2) LEN(4)
FLD: STEP-START-TIME        TYPE(B-SECS) DEC(2) LEN(4)
FLD: STEP-START-DATE        TYPE(P-CYYDDD)
FLD: RDR-START-TIME         TYPE(B-SECS) DEC(2) LEN(4)
FLD: JOB-START-DATE         TYPE(P-CYYDDD)
FLD: RDR-END-TIME           TYPE(B-SECS) DEC(2) LEN(4)
FLD: RDR-END-DATE           TYPE(P-CYYDDD)
FLD: PGMR-NAME              LEN(20)
**
** SELECTED FIELDS FROM THE I/O ACTIVITY SECTION
FLD: NUM-CARDS              TYPE(FULLWORD)  OFFSET(IO-OFFSET)
FLD: NUM-TPUTS              TYPE(FULLWORD)  DISP(*+4)
FLD: NUM-TGETS              TYPE(FULLWORD)
**
** SELECTED FIELDS FROM THE COMPLETION SECTION
FLD: COMP-CODE              LEN(2) FORMAT(HEX) OFFSET(COMP-OFFSET)
FLD: ABEND                  BIT(7) ONTEXT('ABEND') OFFTEXT(' ')
FLD: FLUSH                  BIT(8)
**
** SELECTED FIELDS FROM THE PROCESSOR ACCOUNTING SECTION
FLD: DPRTY                  TYPE(HALFWORD) NOACC  OFFSET(PROC-OFFSET)
FLD: STEP-TCB-SECS          TYPE(FULLWORD) DEC(2) DISP(*+2)
FLD: STEP-SRB-SECS          TYPE(FULLWORD) DEC(2)
FLD: INIT-TCB-SECS          TYPE(FULLWORD) DEC(2)
FLD: INIT-SRB-SECS          TYPE(FULLWORD) DEC(2)
```

Figure 42. File definition of selected fields in SMF type 30 records

0cyydddF in the SMF manual). Here is an example of defining a date field and then using it to select the SMF records to include in a report:

```
FIELD:     SMF-DATE  DISP(10)  TYPE(P-CYYDDD)
INCLUDEIF: REC-TYPE = 5  AND  SMF-DATE = 6/15/1994
```

B–SECS This is a "binary seconds" time field. Most time–of–day and elapsed time fields in SMF records are of this type. You should specify LENGTH(4) for most SMF time fields. Also use the DEC(2) parm to indicate that the binary seconds value contains hundredths of seconds. Here is an example of defining a time field and using it to select SMF records for a report:

```
FIELD:     SMF-TIME  DISP(6)  TYPE(B-SECS)  LENGTH(4)  DEC(2)
INCLUDEIF: REC-TYPE = 5  AND
           (SMF-TIME > 12:59:00 AND < 13:02:30)
```

BIT Some SMF data is contained in bits. For example, there is a bit in type 5 records that indicates whether a job has ABENDed or not. This bit is in the byte at offset 66, and is bit number 6 under IBM's bit numbering convention. Remember that Spectrum Writer numbers bits from 1 to 8 (rather than 0 to 7) from left to right. Thus the ABEND field in the type 5 record can be defined like this:

```
FIELD: ABEND  DISP(66)  BIT(7)
```

To test a bit field, just name the field in your conditional expression. For example, to include all type 5 records which completed with an ABEND, use this statement:

```
INCLUDEIF: REC-TYPE = 5  AND  ABEND
```

You can list bit fields in your COLUMNS statement as well.

```
COLUMNS: SMF-DATE  SMF-TIME  JOBNAME  ABEND
```

By default the word "ABEND" will print in the report if the bit is on, and the words "NOT ABEND" will print if the bit is off. Use the ONTEXT and OFFTEXT parms in the FIELD statement if you want to print different texts. (See an example of this on page 264.)

When defining bit fields, keep one other thing in mind. You should explicitly specify a DISP or COLUMN parm for the first field *following* the bit fields. Spectrum Writer does not automatically increment the current location counter after FIELD statements for bit fields. (This is to allow you to define additional bits within the same byte.) An easy way to specify the DISP of the field following a bit field is to use DISP(*+1):

```
FIELD: BIT-FIELD-A BIT(3)
FIELD: BIT-FIELD-B BIT(7)
FIELD: NEXT-FIELD DISP(*+1) LENGTH(5) ...
```

- In general you should work with only one type of SMF record at a time. Use the INCLUDEIF statement to include only the appropriate type of records in your

report. You can use additional tests to further narrow down which records are included.

```
INCLUDEIF: REC-TYPE = 30  AND  SMF-DATE >= 6/1/1994
```

- Production SMF reports often report on "yesterday's" data. Rather than having to change the date literal in your INCLUDEIF statement for each run, you can COMPUTE yesterday's date, like this:

  ```
  COMPUTE:   YESTERDAY = #INCDATE(-1, DAY)
  INCLUDEIF: REC-TYPE = 30  AND  SMF-DATE = YESTERDAY
  ```

  See "Computing Dates Like "Yesterday," "Last Week", etc." on page 272 for more examples of automatically selecting date ranges bases on the system date.

- Some SMF records are variably formatted. That is, a field may be located at one offset in one record, and at a different offset in another record of the same type. This usually occurs when the record contains segments that are repeated a variable number of times (such as one segment per DD statement in a step). Use Spectrum Writer's OFFSET parm to define variably located fields. This parm is used in the FIELD statement to specify an *additional* offset value to use when determining where a field is located within a record. (This value is added to the COLUMN or DISP parm value.) The advantage of the OFFSET parm is that, unlike the COLUMN and DISP parms, it need not contain a constant numeric value. The OFFSET parm can be any type of numeric expression. For example, it might be something as simple as the name of a previously defined numeric field:

  ```
  FIELD: IO-OFFSET DISP(32) TYPE(FULLWORD)  /* OFFSET TO ID SECTION   */
  ...
  FIELD: JOBNAME DISP(0) OFFSET(IO-OFFSET) LEN(8) /*1ST ITEM IN ID SECTION*/
  ```

  Or, the OFFSET value might be a complex calculation, such as would be needed to compute the location of a field that follows a variable–length array (such as an OCCURS DEPENDING ON array) in a record. For example:

  ```
  FIELD: LAST-FIELD
         OFFSET(100 + (NUM-ITEMS-IN-ARRAY * ITEM-SIZE)) DISP(0)  LENGTH(10)
  ```

  When using the OFFSET parm, remember that the OFFSET parm remains in effect for all subsequent FIELD statements (until a new OFFSET parm is encountered). Thus, you only need to specify the OFFSET parm for the first field in any variably–located segment. Specify OFFSET(0) if you wish to resume defining FIELDs that do not require any OFFSET value.

The following pages show some sample SMF reports produced with Spectrum Writer.

These Control Statements:

```
INPUT: SMF
TITLE: 'BATCH JOB STEPS THAT ABENDED ON' STEP-START-DATE
TITLE: '(522 AND 622 ABENDS NOT INCLUDED)'
INCLUDEIF: REC-TYPE = 30 & SUB-TYPE = 3 & ABEND & NUM-TGETS = 0
           & COMP-CODE ¬= X'0522' & ¬= X'0622'
COLUMNS: JES-JOBID STEP-NUM(4) JOBNAME STEPNAME PGMNAME
         COMP-CODE JOB-CLASS DPRTY(5) PGMR-NAME
         STEP-START-DATE STEP-START-TIME SMF-TIME('STEP|END|TIME')
SORT:    STEP-START-DATE  STEP-START-TIME
```

Produce this Report:

BATCH JOB STEPS THAT ABENDED ON 04/15/94
(522 AND 622 ABENDS NOT INCLUDED)

JES JOBID	STEP NUM	JOBNAME	STEPNAME	PGMNAME	COMP CODE	JOB CLASS	DPRTY	PGMR NAME	STEP START DATE	STEP START TIME	STEP END TIME
STC01453	1	CICS01X	JILLHRS	DFHSIP	0A03		245		04/15/94	06:45:03.39	19:00:36.80
STC01460	1	CICS02	CICS02	DFHSIP	0A03		255		04/15/94	06:45:14.74	19:02:01.47
JOB01596	6	US1PCTN1	UMUD50	XAMUD01	00C7	1	105	PROD.CONTROL	04/15/94	07:10:16.42	07:10:51.27
JOB01609	2	US1PCTDT	USLW47	IDCAMS	0913	T	105	*O"HARRIS	04/15/94	07:18:33.11	07:18:33.76
JOB01609	10	US1PCTDT	UDPX80	XADPX80L	87CF	T	105	*O"HARRIS	04/15/94	07:18:34.44	07:18:57.16
JOB01611	9	US1PCTDT	USLW70	XASLW70	0222	T	105	*O"HARRIS	04/15/94	07:26:57.78	07:50:06.04
JOB01703	1	DPTSSTG0	COMPRS	PG01R00A	840B	M	105	*2ND FLR WEST	04/15/94	07:39:10.53	07:39:31.10
JOB01937	1	DPTSSTG0	COMPRS	PG01R00A	840B	M	105	*2ND FLR WEST	04/15/94	08:24:59.96	08:25:22.89
JOB02028	1	DPTSSTG0	COMPRS	PG01R00A	840B	M	105	*2ND FLR WEST	04/15/94	08:41:21.68	08:41:44.29
STC02055	1	SUBJOB	S1	ACFPRODS	0013		105		04/15/94	08:49:18.15	08:49:18.84
JOB02123	1	DPTSSTG0	COMPRS	PG01R00A	83FC	M	105	*2ND FLR WEST	04/15/94	09:00:39.33	09:01:08.57
STC02196	1	CICS01	CICS	DFHSIP	0222		245		04/15/94	09:13:39.20	10:20:56.80
JOB02214	1	DPTSSTG0	COMPRS	PG01R00A	83FC	M	105	*2ND FLR WEST	04/15/94	09:17:19.92	09:17:48.64
JOB02312	1	DPTSSTG0	COMPRS	PG01R00A	83FC	M	105	*2ND FLR WEST	04/15/94	09:37:36.47	09:38:05.41
JOB02366	1	US1EWT7L	UWCX10	X09CX10	840B	T	105	IMS OPER 281 283	04/15/94	09:48:14.54	09:48:31.08
JOB02385	1	DPTSSTG0	COMPRS	PG01R00A	83FC	M	105	*2ND FLR WEST	04/15/94	09:52:50.16	09:53:21.03
JOB02388	1	DPTSSTG0	COMPRS	PG01R00A	840B	M	105	*2ND FLR WEST	04/15/94	09:53:22.38	09:53:50.41
STC02473	1	IMSMRGN1	MINIRGN1	DFSRRC00	82B0		202		04/15/94	10:07:39.69	10:07:49.04
JOB02500	1	X99M01AC	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	10:11:52.68	10:12:09.66
STC02556	1	CICS01	CICS	DFHSIP	0222		245		04/15/94	10:21:24.92	10:58:19.00
JOB02583	1	X09V01AP	STEP1	IEBGENER	8063	1	105	AP JONES	04/15/94	10:26:03.95	10:26:04.92
JOB02700	2	US1FMTGZ	NPAW012	PNPAW01C	00C7	U	105	WASHINGTON.T	04/15/94	10:49:07.14	10:49:27.19
JOB02741	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	10:56:57.11	10:57:20.74
JOB02747	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	10:58:03.86	10:58:24.58
JOB02748	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	10:58:54.76	10:59:09.61
JOB02752	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	11:00:02.82	11:00:19.91
JOB02768	3	X07W01AX	LKED	IEWL	0D37	T	105	COBOL2	04/15/94	11:03:26.05	11:03:32.71
JOB02773	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	11:04:55.49	11:05:10.95
JOB02774	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	11:05:34.16	11:05:52.24
JOB02815	1	X99M01AT	UPDX86	PRDAX86A	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	11:12:24.80	11:12:39.00
STC03407	1	CICS01	CICS	DFHSIP	0A03		245		04/15/94	13:24:09.32	20:02:03.76
JOB03453	1	DPTSSTCS		XASSR59	0222	M	105	THOMAS	04/15/94	13:30:22.83	13:30:38.52
JOB03466	6	US1PCTN1	UMUD50	XAMUD01	83EA	1	105	SOUTH PROD -MUD50	04/15/94	13:33:09.68	13:33:44.80
JOB03554	3	X06C01A0	AMILL	M2XDNR	00C7	T	105	JONES.LARRY	04/15/94	13:51:48.91	13:53:06.19
JOB03629	1	DPTSSTCP	S1	IEBGENER	0913	M	105	PRINT-OUTPUT	04/15/94	14:02:20.06	14:02:21.44
JOB03715	1	X99M01AT	UPDX86	PRDAX86B	8BB9	T	105	ACCOUNTING- RM 201	04/15/94	14:24:28.94	14:24:41.74
JOB03778	1	US1PCTN8	SRCIN	IEBGENER	8063	T	105	SMITH	04/15/94	14:37:55.27	14:37:56.88
JOB03789	1	US1PCTN8	SRCIN	IEBGENER	8063	T	105	SMITH	04/15/94	14:39:11.97	14:39:13.34
JOB03808	1	US1PCTN2	STEP1	TSMUD02K	0806	1	105	KAREN SMITH	04/15/94	14:42:03.00	14:42:03.91

*** GRAND TOTAL (39 ITEMS)

Figure 43. SMF "Daily ABEND" report

These Control Statements:

```
INPUT: SMF
TITLE: 'TSO SESSIONS ON' STEP-START-DATE
INCLUDEIF: REC-TYPE = 30 & SUB-TYPE = 3 & NUM-TGETS > 0 &
           COMP-CODE ¬= X'0522' & ¬= X'0622'
COMPUTE: SESSION-MINUTES =
         (#MAKENUM(SMF-TIME) - #MAKENUM(STEP-START-TIME)) / 60
COMPUTE: SESSION-COST = SESSION-MINUTES * .0625
COLUMNS: JOBNAME PGMR-NAME STEP-START-DATE('START|DATE')
         STEP-START-TIME('START|TIME') SMF-TIME('END|TIME')
         SESSION-MINUTES(PIC'ZZZ,ZZ9.9') SESSION-COST(PIC'$$$$9.99')
         NUM-TPUTS(7) NUM-TGETS(7)
         STEP-TCB-SECS(8) STEP-SRB-SECS(8)
SORT: PGMR-NAME(2)  STEP-START-DATE  STEP-START-TIME
```

Produce this Report:

TSO SESSIONS ON 05/04/94

JOBNAME	PGMR NAME	START DATE	START TIME	END TIME	SESSION MINUTES	SESSION COST	NUM TPUTS	NUM TGETS	STEP TCB SECS	STEP SRB SECS
D01CDT3	JOE CATRINA	05/04/94	07:47:09.20	11:41:05.33	233.9	$14.62	98	76	10.42	0.45
D01CDTC	JOE CATRINA	05/04/94	11:38:49.99	11:55:37.85	16.8	$1.05	2	3	0.16	0.01
D01CDT3	JOE CATRINA	05/04/94	11:42:04.81	11:55:30.98	13.4	$0.84	75	67	5.71	0.44
D01CDT3	JOE CATRINA	05/04/94	14:07:52.49	16:23:51.19	136.0	$8.50	6	7	0.32	0.03
D01CDTC	JOE CATRINA	05/04/94	14:07:56.93	16:23:41.04	135.7	$8.48	2	3	0.17	0.01
D01CDT3	JOE CATRINA	05/04/94	16:25:07.56	16:37:25.29	12.3	$0.77	22	14	2.62	0.14
*** TOTAL FOR JOE CATRINA	(6 ITEMS)				548.2	$34.26	205	170	19.40	1.08
A20D01A	JOHN A DENNEY	05/04/94	15:58:44.89	16:47:08.89	48.4	$3.03	4	4	0.44	0.02
*** TOTAL FOR JOHN A DENNEY	(1 ITEM)				48.4	$3.03	4	4	0.44	0.02
B55DZT3	JOHN ALWORTH	05/04/94	07:33:55.23	09:04:23.78	90.5	$5.65	4	5	0.17	0.02
B55DZT3	JOHN ALWORTH	05/04/94	10:01:35.15	11:30:33.96	89.0	$5.56	3	4	0.21	0.02
B55DZT3	JOHN ALWORTH	05/04/94	14:07:10.55	16:00:01.04	112.8	$7.05	6	7	0.25	0.04
*** TOTAL FOR JOHN ALWORTH	(3 ITEMS)				292.3	$18.27	13	16	0.63	0.08
Z99TPT6	JOHN TEMPLE	05/04/94	15:11:29.05	16:56:14.00	104.7	$6.55	1	2	0.18	0.01
*** TOTAL FOR JOHN TEMPLE	(1 ITEM)				104.7	$6.55	1	2	0.18	0.01
B02C00A	JOHN X CARLISLE	05/04/94	09:53:24.86	10:07:48.95	14.4	$0.90	27	6	0.50	0.03
B02C00A	JOHN X CARLISLE	05/04/94	11:19:39.67	11:19:58.16	0.3	$0.02	14	1	0.31	0.01
B02C00A	JOHN X CARLISLE	05/04/94	11:24:16.85	11:25:37.29	1.3	$0.08	14	1	0.31	0.01
B02C00A	JOHN X CARLISLE	05/04/94	11:26:04.50	11:27:10.40	1.1	$0.07	11	4	1.23	0.05
B02C00A	JOHN X CARLISLE	05/04/94	11:31:29.49	11:32:41.34	1.2	$0.07	10	7	1.11	0.04
B02C00A	JOHN X CARLISLE	05/04/94	14:23:09.11	14:56:54.33	33.8	$2.11	12	11	0.31	0.02
B02C00A	JOHN X CARLISLE	05/04/94	16:07:53.30	16:15:33.37	7.7	$0.48	54	50	3.02	0.25
B02C00A	JOHN X CARLISLE	05/04/94	16:16:15.29	20:21:33.41	245.3	$15.33	84	84	0.87	0.17
*** TOTAL FOR JOHN X CARLISLE	(8 ITEMS)				305.1	$19.07	226	164	7.66	0.58
F22PDTJ	JOSEPH BROWN	05/04/94	12:11:57.53	12:23:39.63	11.7	$0.73	15	3	1.94	0.09
F22PDTJ	JOSEPH BROWN	05/04/94	15:19:28.85	15:26:54.94	7.4	$0.46	29	16	2.10	0.11
F22PDTJ	JOSEPH BROWN	05/04/94	16:14:43.19	16:57:04.29	42.4	$2.65	0	1	0.16	0.00
*** TOTAL FOR JOSEPH BROWN	(3 ITEMS)				61.5	$3.84	44	20	4.20	0.20
A90CR09	JOY KRAMES	05/04/94	08:08:20.39	08:49:11.43	40.9	$2.55	33	18	0.76	0.07
A90CR09	JOY KRAMES	05/04/94	08:50:34.08	10:26:24.16	95.8	$5.99	15	15	1.96	0.22
A90CR09	JOY KRAMES	05/04/94	10:29:00.02	10:56:38.77	27.6	$1.73	16	17	1.51	0.12
A90CR09	JOY KRAMES	05/04/94	10:58:08.00	11:05:34.36	7.4	$0.46	40	22	2.99	0.18
A90CR09	JOY KRAMES	05/04/94	13:42:25.60	16:35:40.64	173.3	$10.83	39	26	5.42	0.19
*** TOTAL FOR JOY KRAMES	(5 ITEMS)				345.0	$21.56	143	98	12.64	0.78

(other report lines not shown)

Figure 44. SMF "TSO Sessions" report

Working with Date Fields

This section contains tips for working with date fields. In addition to this section, information about dates is found in the following parts of this manual:

- "How to Define a Date Field" (page 340) explains how to define the date fields in your input files.
- "Handling Date and Time Fields in Record Layouts" (page 375) has some tips for working with date fields in Cobol record layouts.

Spectrum Writer supports over 30 different types of date fields commonly found in mainframe files. (For a list of these "date data types," see Appendix A, "Data Types" on page 609.) However, once a date field has been properly defined with the correct data type (in a FIELD statement), you no longer need to be concerned with how it was stored in the input file.

Internally, Spectrum Writer converts *all* date fields from their input file format into its own standard format. Thus, the conversion required to make various kinds of date fields and date literals compatible is done for you automatically.

This means that, regardless of whether a date was stored as a Gregorian date, a Julian date or something else, you will always use **date literals** in the standard MM/DD/YYYY (or MM/DD/YY) format when testing their values. For example:

```
INCLUDEIF: JULIAN-START-DATE > 1/1/2001 AND GREGORIAN-END-DATE < 12/31/2001
```

> **Note:** you can also write your date literals in DD/MM/YYYY and DD/MM/YY format, if you prefer. Just use the **DDMMYYLIT** option (in an OPTIONS statement).

Any date field can be compared with **any other date field,** regardless of how the two dates were stored in the input file. Spectrum Writer handles all necessary conversions:

```
INCLUDEIF: MY-JULIAN-DATE = MY-GREGORIAN-DATE
```

Century Windowing

Spectrum Writer stores all date fields internally with 4-digit years. If your input file contains date fields that do not have an explicit century (for example, YYMMDD or YYDDD dates), Spectrum Writer must decide what century to assign the date to. That is, it must decide whether YY means 19YY or 20YY.

The CENTURY option (in an OPTIONS statement) is used to tell Spectrum Writer how to make this decision. Use it to specify a **century cutoff year** (from 0 to 99). For example:

```
OPTION: CENTURY(80)
```

The above statement tells Spectrum Writer that YY dates less than 80 are 20YY and all other dates are 19YY. Note that the CENTURY Option applies to all YY dates from **all input files** used in a run.

The CENTURY option also applies to the **date literals** in your Spectrum Writer control statements. Date literals may be written in either MM/DD/YYYY or MM/DD/YY format. However, all date literals are stored internally with 4-digit years. When you write a date

literal in the MM/DD/YY format, Spectrum Writer assigns a century for you in the manner just described — based on the century cutoff year from the CENTURY option.

If you do not specify a CENTURY option, Spectrum Writer uses a default century cutoff year of 50. That means that, by default, all YY dates in a run fall in the range from 1950 to 2049.

Of course, the century windowing logic applies *only* to YY date fields and literals. Date fields and literals that have a 4-digit year (YYYY) are not affected. Spectrum Writer uses the century contained in the YYYY value.

It is possible that different files in your shop will use different cutoff years. Since the CENTURY option applies to *all* YY dates in a run, it alone could not handle that situation. In such a case, use the CENTURY option for the most common cutoff year. Then use COMPUTE statements to perform custom century windowing logic on the non-standard cases. For example, assume that one file in your shop uses a cutoff year of 40, while the other files have cutoff years of 60. Specify 60 in your CENTURY parm to handle the most common cases. Then handle the exceptional case this way:

```
FIELD: YYMMDD-DATE  COLUMN(1)  LEN(6) TYPE(CHAR)
FIELD: YY-PART      COLUMN(1)  LEN(2) TYPE(CHAR)

COMPUTE: MY-DATE =  WHEN(YY-PART < '40') ASSIGN(#MAKEDATE('20' + YYMMDD-DATE))
                    ELSE                 ASSIGN(#MAKEDATE('19' + YYMMDD-DATE))
```

You can store the COMPUTE statement right along with the FIELD statements in your file definition library.

How Dates Are Formatted in Your Reports

By default, all date fields, regardless of their century and regardless of how they are stored in the input file, are formatted in your reports like this:

```
MM/DD/YY
```

Over 40 different date display formats are available if you want to format some or all of your date fields differently. The date display formats are listed in Appendix B, "Display Formats" (page 617). For example, you can specify the MM-DD-YYYY display format if you want a to display a date with a four-digit year.

```
COLUMNS: SALES-DATE(MM-DD-YYYY)
```

You can also change the *default* date display format for all dates in a report by using the FORMAT option (in an OPTIONS statement.)

```
OPTIONS: FORMAT(YYYY-MM-DD)
```

Date Delimiters

Date display formats that contain a "dash" (–) result in dates formatted with a delimiter. (The delimiter appears where the dashes appear in the display format name.) By default, this delimiter is a "slash" (/). Thus, the MM-DD-YYYY display format results in dates like this:

```
12/31/2004
```

If you want a different delimiter for your displayed dates, use the DATEDELIM option (in an OPTIONS statement). For example:

```
OPTIONS: DATEDELIM('.')
```

The above statement would result in dates being formatted in your report like this (depending on the display format you choose):

```
12.31.04
31.12.04
04.12.31
12.31.2004, etc.
```

How Dates Are Formatted in Your PC Files

By default, dates in most PC files created by Spectrum Writer are in MM/DD/YY format. If you want MM/DD/YYYY dates in a PC File, use the FORMAT option (*after* the PC option) to specify a different default display format. For example:

```
OPTIONS: PC FORMAT(MM-DD-YYYY)
```

The FORMAT option changes the default display format for date fields. In the above example, dates will now be formatted as MM/DD/YYYY. This unquoted format works in most recent versions of the popular spreadsheet programs. If your PC program still requires quotation marks around dates, use this statement instead:

```
OPTIONS: PC FORMAT(Q-MM-DD-YYYY)
```

Note: Be sure that the FORMAT option *follows* the PC option. Otherwise, the PC option will reset the default date display format.

Working with Julian Dates

You may wonder if Julian date fields require different handling from other kinds of date fields. The answer is no. Once you have used the appropriate Julian DATATYPE in its FIELD statement, you (and other users of the file) can simply forget that the date was originally stored in Julian format. You will work with that date field in exactly the same way as you work with any other date field.

That means that even for fields stored in Julian format, you will still use **date literals** in the standard MM/DD/YYYY (or MM/DD/YY) format when making comparisons to them. (Spectrum Writer does *not* have a "YYDDD" format date literal, so do not try to use such a format.) Here is an example of comparing a Julian date with two date literals:

```
INCLUDEIF: MY-JULIAN-DATE > 1/1/2001 AND < 12/31/2001
```

You can also compare a Julian date field with **another date field** stored in a different format without any special effort on your part:

```
INCLUDEIF: MY-JULIAN-DATE = MY-GREGORIAN-DATE OR MY-SMF-DATE OR MY-STCK-DATE
```

The only exception to this is if your Julian date fields contain ***non-date values*** with special significance (perhaps all zeros, all nines, high-values, etc.) Since such values are not valid Julian dates, Spectrum Writer simply considers these values to be "invalid" data. (You would see ***I*** in your report for such cases.) It is possible to test for these special cases. But to do so, you *will* need to be aware of how the field is stored in the input record. You

should compare the field to an explicit hexadecimal literal of the correct length. For example:

```
INCLUDEIF: MY-JULIAN-DATE <> X'F9F9F9F9F9'     /*COMPARE CHAR JULIAN DATE TO NINES */
INCLUDEIF: MY-JULIAN-PACKED-DATE <> X'000000'  /*COMPARE PACKED DATE TO LOW-VALUES */
```

As far as **report output** goes, by default Spectrum Writer formats Julian date field like all other date fields — in the standard MM/DD/YY format. So again, you don't need to do anything special to have a Julian date field re-formatted into Gregorian in your report. Of course, you can also use an override display format to format a Julian date in any of the over 40 date formats available. (For a list, see "Date Display Formats" on page 620.)

Computing Dates Like "Yesterday," "Last Week", etc.

You can use Spectrum Writer's powerful date-manipulation functions to compute dates or date ranges based on the system date. For example, to select all of the sales for **"yesterday"** from the SALES-FILE, we could use these statements:

```
COMPUTE:   YESTERDAY  = #INCDATE(-1, DAY)
INCLUDEIF: SALES-DATE = YESTERDAY
```

Similarly, to report on all sales made **"last week,"** you could use these statements:

```
COMPUTE:   START-DATE = #BEGWEEK(#INCDATE(-1, WEEK))
COMPUTE:   END-DATE   = #ENDWEEK(#INCDATE(-1, WEEK))
INCLUDEIF: SALES-DATE >= START-DATE AND <= END-DATE
```

You can also increment/decrement date and time pairs by units of time. For example, you could compute an **"expiration date and time"** that is 36 hours after the date and time of a sale this way:

```
COMPUTE: EXPIRE-DATE = #INCDATETIME(SALES-DATE, SALES-TIME, 36, HOURS)
COMPUTE: EXPIRE-TIME = #INCTIME(SALES-TIME, 36, HOURS)
```

You can find the complete syntax for all of these built-in functions, along with other date manipulation functions, in Appendix D, "Built-In Functions" on page 628.

Working with Time Fields

This section offers some tips that you may find helpful when working with time fields.

Spectrum Writer supports two dozen different types of time fields commonly found in mainframe files. These are listed in "Time Data Types" on page 613. For information on defining the time fields in your input files, see "How to Define a Time Field" on page 344.

Time fields, regardless of how they are stored in the input file, are normally formatted in your reports like this:

```
HH:MM:SS
```

However, time fields defined as containing only hours and minutes (the HHMM data type, for example) will be formatted without seconds, like this:

```
HH:MM
```

A number of other time display formats are available if you want to format your time fields differently. These are listed in "Time Display Formats" on page 622. For example, you can specify the HH–MM display format if you want a time field (that has seconds) to be displayed without showing the seconds. Spectrum Writer will round the time to the nearest minute.

You may also specify a "time picture" to change the formatting of time fields in your report. A time picture is similar to a regular numeric picture, except that it begins with TPIC or TP (rather than PIC or P). For example, to format a time field so that leading zeros in the hours are suppressed, you could use a time picture like this:

```
COLUMNS: START-TIME(TPIC'Z9:99:99')
```

Time pictures can also specify decimal digits if needed for the time field:

```
COLUMNS: JOB-END(TP'ZZ:ZZ:Z9.99999')
```

By default, time fields are not totalled in reports. If you want to total a time field, you may specify the ACCUM parm in either the FIELD, COMPUTE or COLUMNS statement (just as with numeric fields). If you do print totals for a time field, you may also need to specify additional display digits for the hour portion of the total (in case the total is more than 99 hours):

```
COLUMNS: DURATION(ACCUM,TP'ZZZ9:99:99')
```

You may also choose to format time fields in your report as hours and decimal portions of an hour. That is, the time 04:15:00 would be displayed as 4.25 (4 and one–fourth hours). The HOURS display format does this. There are also MINS and SECS formats to display time fields as a number of minutes or a number of seconds. The number of decimal digits printed with such display formats is the number of decimal digits specified in the FIELD or COMPUTE statement used to define the field (usually zero). To force a certain number of decimal digits to print with these display formats, use a COMPUTE statement to change a field's decimal precision. For example, to print START–TIME in hours, with three decimal digits, do this:

```
FIELD:   START-TIME COL(10) TYPE(HHMMSS)
COMPUTE: X-START-TIME(3) = START-TIME
COLUMNS: X-START-TIME(HOURS)
```

You may use time fields in conditional expressions. They can be compared with other time fields or with time literals. Time literals must be expressed either as HH:MM or HH:MM:SS (with optional decimal parts of seconds also allowed: HH:MM:SS.NNN). Here are some examples of using time fields and time literals in INCLUDEIF statements:

```
INCLUDEIF: START-TIME > END-TIME
INCLUDEIF: START-TIME > 12:00
INCLUDEIF: LOG-TIME > 13:01:00.0 AND < 13:01:00.5
```

You may also use time fields in computational expressions. For example:

```
COMPUTE: DURATION = END-TIME - START-TIME
```

The above statement computes a time field called DURATION, whose value is the difference between END–TIME and START–TIME. For example, if END–TIME had a value of 17:30:45 and START–TIME was 17:25:35, then DURATION would have a value of 00:05:10.

If the start and end times might occur on different days, you should also convert the start and end *dates* into seconds and use those in the computation as well:

```
COMPUTE: DURATION = ((#MAKENUM(END-DATE) * 86400)   + END-TIME)
                  - ((#MAKENUM(START-DATE) * 86400) + START-TIME)
```

Note: There are 86,400 seconds in one day.

When computing time fields, you are allowed to mix time fields and numeric fields in the computational expression. Any numeric fields (or numeric literals) in the expression are considered to represent a number of seconds. For example:

```
COMPUTE: NEXT-MINUTE = START-TIME + 60
```

The above statement creates a new time field call NEXT–MINUTE whose value is equal to START–TIME plus 60 seconds.

Two built–in functions are provided to allow you to convert time fields to numeric fields and vice verse. Use the #MAKENUM function to convert a time field into a numeric field. For example:

```
COMPUTE: START-SECONDS = #MAKENUM(START-TIME)
```

The above statement creates a new numeric field named START–SECONDS. If START–TIME contained 02:30:05, START–SECONDS' value would be 9005. (Two hours is 7200 seconds, 30 minutes is another 1800 seconds, plus the 5 seconds.)

To convert numeric fields (which are considered a number of seconds) into a time field, use the #MAKETIME function:

```
COMPUTE: END-TIME = #MAKETIME(END-SECONDS)
```

If END–SECONDS contained 3600, then END–TIME would be 01:00:00 (since 3600 seconds is one hour).

You can also use the #MAKETIME function to convert a character value (in HHMMSS format) into a time field. For example:

```
COMPUTE: END-TIME = #MAKETIME(CHAR-TOD)
```

If CHAR–TOD was a 6–byte character field containing 191059, then END–TIME would be a time field with a value of 19:10:59.

Spectrum Writer has a built–in field named #HHMMSS which contains the system time that Spectrum Writer began running. You can use this field like any other time field in creating reports or PC files.

Spectrum Writer supports time fields based on the "Store Clock" (STCK) machine instruction. (Use the STCKTIME data type — in the FIELD statement — to define such a field.) STCK values contain the date and time in GMT. Spectrum Writer automatically converts STCKTIME times from GMT to local time. The hours added or subtracted to the GMT time are determined by your installation's system parm. To change this default, use the STCKADJ option to specify the number of hours that should be added to the STCKTIME time. For example, to suppress conversion and leave STCKTIME times in GMT, you could specify the following:

```
OPTIONS: STCKADJ(0)
```

Computing Times Like "30 Minutes Ago," "Last Hour", etc.

You can use Spectrum Writer's powerful time-manipulation functions to compute times or time ranges based on the system time. For example, to select all of the sales made "today, in the **last 30 minutes"** we could use these statements:

```
COMPUTE:   THIRTY-MINUTES-AGO  = #INCTIME(-30, MINUTES)
INCLUDEIF: SALES-DATE = #TODAY AND SALES-TIME >= THIRTY-MINUTES-AGO
```

Similarly, to report on all sales made **"last hour,"** you could use these statements:

```
COMPUTE:   START-TIME = #BEGHOUR(#INCTIME(-1, HOUR))
COMPUTE:   END-TIME   = #ENDHOUR(#INCTIME(-1, HOUR))
INCLUDEIF: SALES-DATE = #TODAY AND SALES-TIME >= START-TIME AND <= END-TIME
```

However, if the "last hour" could have occured in a different day (as would be the case if you ran the job shortly after midnight), then you need an additional COMPUTE statement. It determines what the *date* was 30 minutes ago:

```
COMPUTE:   START-TIME = #BEGHOUR(#INCTIME(-1, HOUR))
COMPUTE:   END-TIME   = #ENDHOUR(#INCTIME(-1, HOUR))
COMPUTE:   START-DATE = #INCDATETIME(-1, HOUR)
INCLUDEIF: SALES-DATE = START-DATE AND SALES-TIME >= START-TIME AND <= END-TIME
```

You can find the complete syntax for all of these built-in functions, along with other time manipulation functions, in Appendix D, "Built-In Functions" on page 628.

Producing Files for Non-Standard PC Programs

Chapter 3, "How to Request a PC File" explained how to produce PC files that work in virtually all PC spreadsheet, database and similar programs. This section describes several methods you can use to get mainframe data into a PC program that does not accept Spectrum Writer's standard comma-delimited files. These methods are:

- specify your own combination of formatting options to create a PC file that the program will accept
- create a "fixed format ASCII" file (rather than a "delimited ASCII" file), if your PC program will import such files. Fixed format ASCII files generally are not as easy to import, since you must describe the file's exact record layout to your PC program.
- use a two–step process. For example, if your PC program *can* import Excel spreadsheets, do the following. First, use Spectrum Writer to create a PC file and import that into Excel. Then, have Excel save it as a spreadsheet file, which your other PC program can now open.

Terminology: In this section, we talk about creating "delimited ASCII" and "fixed format ASCII" files. These are the terms you are likely to see in your PC program's menus and help screens. To be precise, the files Spectrum Writer creates on your mainframe are not yet "ASCII." They are EBCDIC (which allows you to browse them conveniently while they are still on your mainframe). The files will be translated

from EBCDIC to ASCII during the download process to your PC. Once on your PC, you will truly have a delimited or fixed format ASCII file.

For information on creating true ASCII files directly on the mainframe, read "How to Format Data as ASCII" on page 143.

Standard Delimited PC File

Most popular PC programs will import data that is formatted as a **delimited ASCII file**. The first method, then, is to create an output file in this standard format and try to import it into your PC program. Use the following statement to create a standard delimited ASCII file:

```
OPTIONS: PC
```

For instructions on importing delimited ASCII files into your PC program, check the program's online help (or printed manual) under "importing" or "ASCII". You might also check under various other names that are commonly used for this kind of file, such as: "delimited files," "comma separated values," "CSV," "DOS files," "ASCII files," or "text files".

Custom PC File

First, let's look in detail at the standard PC file that Spectrum Writer creates when you specify the PC option. **Figure 45** shows such a file. The PC file has the following features:

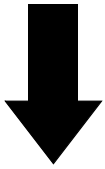
- fields are separated from each other with commas
- character data is enclosed within quotation marks
- numbers are formatted without imbedded commas
- dates are formatted in MM/DD/YY format and are enclosed in quotation marks
- times are formatted in HH:MM:SS format and are enclosed in quotation marks
- no titles or Grand total lines are included
- a "carriage control" character is *not* inserted in the first byte of each output record

Each of these characteristics of the PC file can be specified separately, in any combintation. Many of them can also be customized. This is done by speciying individual formatting options in the OPTIONS statement, instead of the PC option.

If your PC program does not import the comma-delimited files properly, it may have its own special requirements for import files. Study the special OPTIONS statement parms below to find the ones that will enable you to format your output file according to the PC program's requirements. By specifying these options in various combinations you can

These Control Statements:

```
OPTIONS:   PC
INPUT:     EMPL-FILE
COLUMNS:   LAST-NAME  HIRE-DATE   SALES-QTR1  SALES-QTR2
                                  SALES-QTR3  SALES-QTR4
```



Produce this Report:

```
"LAST","HIRE","SALES","SALES","SALES","SALES"
"NAME","DATE","QTR1","QTR2","QTR3","QTR4"
" "," "," "," "," "," "
"JONES            ","01/31/80",    9956.01,   10511.56,    8698.07,   13334.25
"JOHNSON          ","06/21/75",   21560.15,   21350.21,   19970.10,   24118.78
"JOHNSON          ","11/25/79",   14590.34,   17220.10,   20100.08,   23113.12
"MACDONALD        ","07/04/82",     548.50,     687.13,     599.25,     726.10
"SIMPSON          ","12/01/82",    1287.58,    5109.03,     998.12,    1329.15
"MORRISON         ","11/30/79"'   25014.19,   26112.21,   28010.09,   18918.50
"CHRISTOPHERSON   ","08/15/81",   13807.22,   16549.01,    8050.07,    9259.01
"BAKER            ","06/04/82",   21336.10,   24999.02,   24001.33,   21789.44
"THOMAS           ","06/04/82",   14889.07,   18045.05,   14250.12,   13009.25
```

Remarks:

- specifying PC causes the "report" to be formatted as a "delimited ASCII" file
- all character data is enclosed in quotation marks
- all numbers are formatted without commas
- all dates are formatted as MM/DD/YY, and enclosed in quotation marks
- each column is separated from the next column with a comma
- all titles and column headings are suppressed
- the Grand Total line is suppressed
- this file can be downloaded to a PC and imported directly into many PC programs

Figure 45. A standard comma-delimited PC File

create an output file in just about any format. (Each of these options is discussed in more detail in Chapter 10, "Control Statement Syntax.")

USEFUL OPTIONS FOR CREATING CUSTOM PC FILES

OPTION	DESCRIPTION
COLHDGONCE	This option suppresses all title lines and causes the column headings to print just once (at the very beginning of the PC file).

COLSEP	This option lets you specify a "column separator" character. When producing PC files, you usually want to separate ("delimit") the data columns with commas. The following statement does that:
	`OPTIONS: COLSEP(',')`
	If your PC program requires that fields be separated with a tab character (as older versions of Excel did), use this statement:
	`OPTIONS: COLSEP(X'05')`

FORMAT	This option allows you to specify any display format you want as the *default* display format for a run. This is useful when you want to change the way *all fields* in your output are formatted. For example, when creating PC files you might specify:
	`OPTIONS: FORMAT(QCHAR, NOCOMMA, Q-MM-DD-YYYY, Q-HH-MM-SS)`
	The above statement makes QCHAR, NOCOMMA, Q–MM–DD–YYYY and Q–HH–MM–SS the default display formats for character, numeric, date and time fields, respectively. Therefore, by default: all character fields will be enclosed within quotation marks; all numeric fields will be formatted without imbedded commas; all dates will be formatted in MM/DD/YYYY format and be enclosed within quotation marks; and all times will be formatted in HH:MM:SS format and be enclosed within quotation marks.
	Let's say that your PC program requires dates to be imported in YYYYMMDD format, within quotation marks. You could use this option:
	`OPTIONS: FORMAT(QCHAR, NOCOMMA, Q-YYYYMMDD, Q-HH-MM-SS)`
	A list of all available display formats can be found in Appendix B, "Display Formats," on page 617.

FORMAT *(continued)*	Use this option to specify the display formats that are appropriate for your PC program. (A complete list of display formats is found in Appendix B, "Display Formats" on page 617).

USEFUL OPTIONS FOR CREATING CUSTOM PC FILES (CONTINUED)	
OPTION	**DESCRIPTION**
HGCOLHDG	This option specifies that "Harvard Graphics" style column headings are wanted. This option causes the column headings to appear in a single line in the output file (rather than being split onto multiple lines). The "blank" line that normally separates the column headings from the actual data is also suppressed. This option is useful when the PC program which will be importing your output file expects the first line of input to contain a legend for the data in the subsequent lines.
NOCC	This option suppresses the "carriage control" character in the report records. The carriage control character is needed when sending a report to a *printer*, but is not normally desired when writing output records to a PC file.
NOCOLHDGS	This option suppresses all column headings (but not report titles) from the output.
NOGRANDSPACES	This option suppresses the blank spacing lines before the Grand Total record. It is useful in those cases where you do want a Grand Total record, but don't want extra blank records in your output file.
NOGRANDTOTAL	This option suppresses the Grand Totals (including spacing lines) from the output.
NOTITLES	This option suppresses all titles, column headings, footnotes and page breaks from the report.
OUTPUT	You may specify the OUTPUT option parm, like this: `OPTIONS: OUTPUT` The OUTPUT option tells Spectrum Writer that you are creating some form of output file rather than a report. It produces the following results, which are normally desired for output files: • it suppresses all titles and column headings • it suppresses the Grand Totals line • it suppresses the "carriage control" character • it suppresses the maximum pages/lines message (which is normally printed when the MAXPAGE or MAXPRINT option is used).

Fixed Format ASCII Files

Some PC programs import fixed format (or "fixed width") ASCII files. To create a fixed format ASCII file, use the following combination of options:

```
OPTIONS: OUTPUT FORMAT(CHAR, NOCOMMA, MM-DD-YY, HH-MM-SS)
```

The above statement results in an output file with the following features:

- there is one blank space between each field in the output record
- quotation marks are *not* put around any of the data
- character data is written "as is"
- numbers are formatted without imbedded commas
- dates are formatted in MM/DD/YY format
- times are formatted in HH:MM:SS format
- no titles, column headings, or Grand Total lines are included
- a "carriage control" character is not inserted in the first byte of each output record

As mentioned earlier, when importing a fixed format ASCII file into a PC program, you must define the PC file records to that PC program. Check your PC program's "Help" for instructions on how to import fixed format files.

Producing Files for Mainframe Programs

Output files that will be used in mainframe programs will be considerably different from output files intended for PC programs. The exact requirements for a mainframe output file will depend, of course, on the particular program that will process the file. This section discusses various options that you'll find helpful when creating mainframe output files.

Simply specifying MAINFRAME is one way to produce a "generic" mainframe output file:

```
OPTIONS: MAINFRAME
```

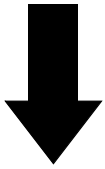
Figure 46 shows a sample output file created using the above statement. Files in this format are compatible with COBOL, PL/1 and Assembler language programs. The output file has the following features:

- there are no blank spaces (nor commas) between the fields in the output record
- character data is written "as is"
- numbers are formatted in the DISPLAY format (no imbedded commas, no leading zero suppression, the last digit includes the sign)
- dates are formatted in YYMMDD format
- times are formatted in HHMMSS format
- no titles, column headings, or Grand Total lines are included
- a "carriage control" character is not inserted in the first byte of each output record

If the standard "mainframe" formatted output file described above is not what you need, you can specify various other individual options to customize your output file. The following paragraphs discuss some of these options.

These Control Statements:

```
OPTIONS: MAINFRAME
INPUT:   EMPL-FILE
COLUMNS: LAST-NAME  HIRE-DATE  SALES-QTR1  SALES-QTR2
                               SALES-QTR3  SALES-QTR4
```



Produce this Output File

```
JONES          800131000009956.01000010511.56000008698.07000013334.25
JOHNSON        791125000014590.34000017220.10000020100.08000023113.12
JOHNSON        750621000021560.15000021350.21000019970.10000024118.78
MACDONALD      820704000000548.50000000687.13000000599.25000000726.10
SIMPSON        821201000001287.58000005109.03000000998.12000001329.15
MORRISON       791130000025014.19000026112.21000028010.09000018918.50
CHRISTOPHERSON 810815000013807.22000016549.01000008050.07000009259.01
BAKER          820604000021336.10000024999.02000024001.33000021789.44
THOMAS         820604000014889.07000018045.05000014250.12000013009.25
```

Remarks:

- specifying MAINFRAME causes the "report" to be formatted as a mainframe file
- all character data is written "as is"
- all numbers are formatted in the DISPLAY display–format
- all dates are formatted as YYMMDD
- there are no blank spaces or delimiters between fields
- all titles and column headings are suppressed
- the Grand Total line is suppressed

Figure 46. An output file created with the MAINFRAME option

When creating mainframe output files, you probably will *not* want blank spaces between fields in the output records. This will save disk space in the output file. You can accomplish this by specifying zero in the "column spacing" option:

```
OPTIONS: COLSPACE(0)
```

In mainframe files, you may want some numeric fields to be "packed" in order to take up less room in the file. ("Packed" is the same as COMP–3 in COBOL, and FIXED DECIMAL in PL/1.) To do this, just use the PACKED display format for those numeric fields. You can specify PACKED directly in the COLUMNS statement for individual fields, like this:

```
COLUMNS: EMPL-NAME  SALES-QTR1(PACKED,6)  SALES-QTR2(PACKED,6)
```

The above statement causes SALES–QTR1 and SALES–QTR2 to be formatted as 6–byte packed fields in the output file. You can also make PACKED the *default* numeric display format by using the FORMAT option, like this:

```
OPTIONS: MAINFRAME FORMAT(PACKED)
COLUMNS: EMPL-NAME  SALES-QTR1(6)  SALES-QTR2(6)
```

The above statements also cause the two sales fields to be output as 6–byte packed fields. Be sure to put the FORMAT option *after* the MAINFRAME option. Otherwise, the MAINFRAME option will override the FORMAT option.

If you want your output file to contain *binary* data (COMP in COBOL, FIXED BINARY in PL/1), use the BINARY display format in a similar way:

```
COLUMNS: EMPL-NAME  DEPT-NUM(BINARY,1)  TOTAL-SALES(PACKED,8)
```

The above statement formats DEPT–NUM as a 1–byte binary field, and TOTAL–SALES as an 8–byte packed field. Note that the output format you specify for a field can be different than the way the field is formatted in the input file. For example, TOTAL–SALES is defined as a 7–byte "display" numeric field in our sample EMPL–FILE. Yet we chose to output it as an 8–byte packed number in the example above.

You can also use the HALFWORD and FULLWORD display formats as a shorthand way to output 2–byte and 4–byte binary fields, respectively:

```
COLUMNS: EMPL-NAME  DEPT-NUM(HALFWORD)  TOTAL-SALES(FULLWORD)
```

Also use display formats to specify how you want *date* fields to be output. For example:

```
COLUMNS: EMPL-NAME  HIRE-DATE(P-YYDDD)
```

The above statement formats HIRE–DATE as a 3-byte packed, Julian date. (This is equivalent to PICTURE S9(5) COMP–3 in COBOL.)

Again, you can use the FORMAT option to change the default way that date fields are formatted in your mainframe file:

```
OPTIONS: MAINFRAME FORMAT(YYYYMMDD)
COLUMNS: HIRE-DATE
```

The above statements cause the HIRE-DATE field (and all other date fields) to be formatted in YYYYMMDD format.

A complete list of display formats available for formatting numeric, date and time fields in your output records is found in Appendix B, "Display Formats" (page 617).

When creating files for use on ASCII-based machines, you may want to format some fields in ASCII (rather than in EBCDIC). To do this, specify the ASCII parm after the field name in your COLUMNS statement:

```
COLUMNS:  EMPL-NAME(ASCII) HIRE-DATE(ASCII) DEPT-NUM(BINARY,1) TOTAL-SALES(PACKED,8)
```

For more information on creating ASCII output files, see page 143.

When creating mainframe files you probably will not want titles, columns headings or Grand Total lines. You will also not want a carriage control character in the first byte of the output records. The MAINFRAME option automatically suppresses all of these for you. Or, you can use the following options to selectively suppress one or more of those items:

```
OPTIONS: NOTITLES  NOCOLHDGS  NOGRANDTOTAL  NOCC
```

In some cases you may want to include a Grand Total record in your output file. In such cases, you may want to specify the NOGRANDSPACES option to suppress the blank lines normally written just before the Grand Total line.

```
OPTIONS: NOGRANDSPACES
```

When creating mainframe output files, you may want your records to be larger (or smaller) than the standard 133–byte output record. Chapter 8, "Operating System Considerations" explains how to specify any record length you want for your output file. See page 417 (OS/390) or page 431 (VSE).

How to "Subset" Mainframe Files

One common reason for creating mainframe files is to select certain *whole records* from the input file and write them to a "subset" file. For example, we might want to create an output file consisting of complete EMPL–FILE records, but *only* for those employees in department 2. It would take a lot of effort to write a COLUMNS statement containing each individual field name from the EMPL–FILE along with its desired output format. A much simpler way is to define a single character field which corresponds to the *entire input record*, and just write that one field to your output file:

```
OPTIONS:   MAINFRAME
INPUT:     EMPL-FILE
FIELD:     RECORD  COLUMN(1)  LENGTH(150)
INCLUDEIF: DEPT-NUM = 2
COLUMNS:   RECORD
```

The above statements create an output file which contains the EMPL–FILE records for employees in department 2.

How to Sort Mainframe Files

Similarly, you can use Spectrum Writer to sort mainframe files. One advantage of using Spectrum Writer is that you can simply name the fields that you want to sort on (rather than having to specify the exact columns, lengths and data types of the sort fields). Here is an example of sorting a mainframe file.

```
OPTIONS: MAINFRAME
INPUT:   EMPL-FILE
FIELD:   RECORD  COLUMN(1)  LENGTH(150)
```

```
SORT:    DEPT-NUM  LAST-NAME  FIRST-NAME
COLUMNS: RECORD
```

The above statements create an output file which contains all of the EMPL–FILE records, sorted into DEPT–NUM, LAST–NAME and FIRST–NAME order.

Computing Percent of Totals

While Spectrum Writer does not have an automatic "percent of total" function, it is possible to create reports with such data. The trick is to use multiple Spectrum Writer steps.

The problem with performing such calculations in a single step is this: Spectrum Writer adds up totals as a report is being printed. Therefore it doesn't know what the total value will be (for a region, for example) until all of the records in that region have printed. Thus the total value (which is required to calculate percent of total) is not available when the detail report lines are being printed. However, using multiple steps lets you get around this problem.

The report in **Figure 49** (page 288) shows a report with two "percent of total" columns. Those columns show the percent of the regional total represented by the AMOUNT and TAX fields in each sales record.

We used three short Spectrum Writer steps to produce this report. Here's an overview of those steps:

Step 1 computed the regional totals and wrote those totals out to a file, along with all of the regular records from the SALES-FILE.

Step 2 sorted this file so that the regional totals were now located *before* each region's sales records.

Step 3 used this sorted file to print a report with percent of totals. The necessary regional totals were now available while the detail records were being processed.

Now let's look at each of these steps in more detail.

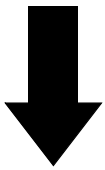
Step 1.

The first step writes out a temporary dataset that contains all of the input records, plus total records (containing the regional totals). A special sort key is also added to each record in this output file. (See **Figure 47**.)

This step uses the standard SALES-FILE as input. Since this step produces a mainframe output file (instead of a report) we specified the MAINFRAME option. We want the output file to contain the complete SALES-FILE records, plus one new "regional total" record for each region. The new total record will contain the region's total value for AMOUNT and TAX. We defined a new field called RECORD. It is simply a character field that includes the whole 80-byte record from the SALES-FILE. The COLUMNS statement tells Spectrum Writer to write out this unchanged 80-byte record. After the 80-byte record, we write the value of the REGION field (again) and the letter "B". This forms a 6-byte sort key that we will use in the next step.

These Control Statements:

```
OPTIONS: MAINFRAME
INPUT: SALES-FILE
FIELD: RECORD COLUMN(1) LENGTH(80)
COLUMNS: RECORD REGION 'B'
SORT: REGION
BREAK: REGION NOTOTALS SPACE(0)
       FOOTING(80 REGION 0 'A' 0 AMOUNT(TOTAL DISPLAY 8)
                0 TAX(TOTAL DISPLAY 8) )
```



Produce this Output File:

```
SIMPSON   041042EAST 00238701430360950430153021J & S LUMBER   40855523212451916 EAST B
SIMPSON   041039EAST 00149900900360950401081757EUROPEAN DELI  40855565430150916 EAST B
MORRISON  042045EAST 00296501780360950330190541A1 PHOTOGRAPHY 40855577860600919 EAST B
MORRISON  042036EAST 00443502660360950329153022STAR MARKET    40855576540599907 EAST B
                                                                                 EAST A00112.8600006.77
JOHNSON   039044NORTH000998006003709504051433l0MARYS ANTIQUES 41555512560000997 NORTHB
JOHNSON   039036NORTH023445140703709504011702471VILLA HOTEL    41555576300929926 NORTHB
JONES     036039NORTH001025006203709504151352411TOY TOWN       41555515000523977 NORTHB
JONES     036039NORTH012176073103709504150801591TOY TOWN       41555515001200907 NORTHB
JONES     036042NORTH001025006203709504150758321EZ GROCERY     41555548720810977 NORTHB
                                                                                 NORTHA00386.6900023.22
JOHNSON   037041SOUTH010138060903509503121025001ACE ELECTRICAL 21355598710079952 SOUTHB
JOHNSON   037042SOUTH050000300003509504161148331ACME BUILDING  21355521211025976 SOUTHB
                                                                                 SOUTHA00601.3800036.09
BAKER     044045WEST 013700082203609503261209091JACKS CAFE     21455511240102978 WEST B
THOMAS    045037WEST 000998006003609504141541381YOGURT CITY    21455517895421997 WEST B
BAKER     044037WEST 013575081503609504121431121JACKS CAFE     21455511240231916 WEST B
                                                                                 WEST A00282.7300016.97
```

Remarks:

- the output file contains all of the SALES-FILE records, plus one regional total record for each region
- all records now contain a new 6-byte sort key, consisting of the region name followed by either "A" or "B"
- the regional total records contain the total values for AMOUNT and TAX in new fields at the end of the record

Figure 47. This step adds "region total records" to the file, and also creates a special sort key

In order to compute regional totals, we sort and break on the REGION field. At the control break, we suppress the default totals line (with the NOTOTALS parm in the BREAK statement). Instead, we specify our own custom output record using the FOOTING parm. This total record will consist of 80 bytes of blanks, followed again by a special sort key. The sort key is the 5-byte region followed by the letter "A". (The letter "A" allows this total record to sort *ahead* of the detail records in the next step.) Following this sort key, is the region's total value for AMOUNT and TAX. Each of these totals is written as an 8-byte "display numeric" field.

We also specified SPACE(0) in the BREAK statement. That prevents two blank records (the default break spacing) from being written at the control breaks.

Figure 47 (page 285) shows the output file created by this first step.

> **Note:** In the execution JCL we directed SWOUTPUT's output to a temporary dataset, rather than to a printer. We specified an LRECL of 102 for this file (the length of the 80 byte sales record, plus a new 6-byte sort key, plus the two 8-byte regional totals).

Step 2.

This step simply sorts the file created in Step 1 on the new 6-byte sort key. (This key is the region value plus either the letter "A" or "B".) After the sort, the record containing a region's totals will appear *ahead* of the detail sales records for that region. (See **Figure 48**.)

You can use a standard Sort step if you prefer. Or you can use a Spectrum Writer step to perform the sort. **Figure 48** shows the control statements used to sort a file with Spectrum Writer. It also shows the resulting temporary dataset, now sorted and ready to use in the final step.

In the JCL, we used the temporary dataset created in Step 1 as the input. And we wrote the output to another 102-byte temporary dataset.

Step 3.

Now we're ready to produce the final report. We read in the sorted temporary dataset. The nice thing about this file is that it contains each region's totals (for AMOUNT and TAX) *before* the first detail record for that region. As we read this file, we can save these two regional totals in special compute fields. Then, as we read the following detail sales records for that region, we can perform the percent of region computation.

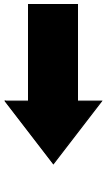
This step is shown in **Figure 49** (page 288).

We named the SALES-FILE in the INPUT statement, even though the JCL actually points to our temporary 102-byte dataset. That's because the first 80 bytes of these records match the SALE-FILE layout All of the FIELD statements for SALES-FILE will also work with this file (for the detail records). We specified an override LRECL parm in the INPUT statement, since our input file is larger than the standard SALES-FILE (102 bytes versus 80).

We then added three FIELD statements to define the new fields that aren't a part of the SALES-FILE.

These Control Statements:

```
OPTIONS: MAINFRAME
FILE: BIGSALE  DDNAME(BIGSALE) LRECL(102)
FIELD: BIGREC  COLUMN(1)  LENGTH(102)
FIELD: SORTKEY COLUMN(81) LENGTH(6)
INPUT: BIGSALE
COLUMNS: BIGREC
SORT: SORTKEY
```



Produce this Output File:

```
                                                                                EAST A00112.8600006.77
MORRISON  042036EAST 00443502660360950329153022STAR MARKET    40855576540599907 EAST B
MORRISON  042045EAST 00296501780360950330190541A1 PHOTOGRAPHY 40855577860600919 EAST B
SIMPSON   041039EAST 00149900900360950401081757EUROPEAN DELI  40855565430150916 EAST B
SIMPSON   041042EAST 00238701430360950430153021J & S LUMBER   40855523212451916 EAST B
                                                                                NORTHA00386.6900023.22
JONES     036042NORTH00102500620370950415075832EZ GROCERY     41555548720810977 NORTHB
JONES     036039NORTH01217607310370950415080159TOY TOWN       41555515001200907 NORTHB
JONES     036039NORTH00102500620370950415135241TOY TOWN       41555515000523977 NORTHB
JOHNSON   039036NORTH02344514070370950401170247VILLA HOTEL    41555576300929926 NORTHB
JOHNSON   039044NORTH00099800600370950405143310MARYS ANTIQUES 41555512560000997 NORTHB
                                                                                SOUTHA00601.3800036.09
JOHNSON   037042SOUTH05000030000350950416114833ACME BUILDING  21355521211025976 SOUTHB
JOHNSON   037041SOUTH01013806090350950312102500ACE ELECTRICAL 21355598710079952 SOUTHB
                                                                                WEST A00282.7300016.97
BAKER     044037WEST 01357508150360950412143112JACKS CAFE     21455511240231916 WEST B
THOMAS    045037WEST 00099800600360950414154138YOGURT CITY    21455517895421997 WEST B
BAKER     044045WEST 01370008220360950326120909JACKS CAFE     21455511240102978 WEST B
```

Remarks:

- the regional total records now appear *ahead* of the detail records for that region

Figure 48. Sorting the temporary dataset so that the regional totals come before the detail data

These Control Statements:

```
INPUT: SALES-FILE LRECL(102)
FIELD: RECTYPE    LENGTH(1)  COLUMN(86)
FIELD: REGION-AMT LENGTH(8)  TYPE(NUM-SLD) DEC(2)
FIELD: REGION-TAX LENGTH(8)  TYPE(NUM-SLD) DEC(2)
COMPUTE: SAVE-REGION-AMT = WHEN(RECTYPE='A') ASSIGN(REGION-AMT)
                           ELSE              RETAIN
COMPUTE: SAVE-REGION-TAX = WHEN(RECTYPE='A') ASSIGN(REGION-TAX)
                           ELSE              RETAIN
COMPUTE: PERCENT-REGION-AMOUNT = AMOUNT / SAVE-REGION-AMT * 100
COMPUTE: PERCENT-REGION-TAX    = TAX    / SAVE-REGION-TAX * 100
INCLUDEIF: RECTYPE='B'
COLUMNS: REGION EMPL-NAME SALES-DATE CUSTOMER
         AMOUNT PERCENT-REGION-AMOUNT(PIC'ZZ9.9%' NOACCUM)
         TAX PERCENT-REGION-TAX(PIC'ZZ9.9%' NOACCUM)
SORT:  REGION SALES-DATE
BREAK: REGION
```

Produce this Report:

```
MON  07/19/04    2:51 PM        DATA FROM SALES-FILE                    PAGE     1

                                                          PERCENT             PERCENT
          EMPL     SALES                                  REGION              REGION
REGION    NAME     DATE     CUSTOMER          AMOUNT      AMOUNT     TAX      TAX

EAST   MORRISON   03/29/95 STAR MARKET            44.35   39.3%        2.66   39.3%
EAST   MORRISON   03/30/95 A1 PHOTOGRAPHY         29.65   26.3%        1.78   26.3%
EAST   SIMPSON    04/01/95 EUROPEAN DELI          14.99   13.3%        0.90   13.3%
EAST   SIMPSON    04/30/95 J & S LUMBER           23.87   21.2%        1.43   21.1%
*** TOTAL FOR EAST  (    4 ITEMS)                112.86                6.77

NORTH  JOHNSON    04/01/95 VILLA HOTEL           234.45   60.6%       14.07   60.6%
NORTH  JOHNSON    04/05/95 MARYS ANTIQUES          9.98    2.6%        0.60    2.6%
NORTH  JONES      04/15/95 TOY TOWN               10.25    2.7%        0.62    2.7%
NORTH  JONES      04/15/95 TOY TOWN              121.76   31.5%        7.31   31.5%
NORTH  JONES      04/15/95 EZ GROCERY             10.25    2.7%        0.62    2.7%
*** TOTAL FOR NORTH (    5 ITEMS)                386.69               23.22

SOUTH  JOHNSON    03/12/95 ACE ELECTRICAL         101.38   16.9%        6.09   16.9%
SOUTH  JOHNSON    04/16/95 ACME BUILDING          500.00   83.1%       30.00   83.1%
*** TOTAL FOR SOUTH (    2 ITEMS)                601.38               36.09

WEST   BAKER      03/26/95 JACKS CAFE             137.00   48.5%        8.22   48.4%
WEST   BAKER      04/12/95 JACKS CAFE             135.75   48.0%        8.15   48.0%
WEST   THOMAS     04/14/95 YOGURT CITY              9.98    3.5%        0.60    3.5%
*** TOTAL FOR WEST  (    3 ITEMS)                282.73               16.97


****** GRAND TOTAL (    14 ITEMS)              1,383.66               83.05
```

Figure 49. A report with "percent of total" columns

We used a special kind of COMPUTE statement to "save" a region's total AMOUNT and TAX values for use with all of the detail records that follow. The COMPUTE statement makes use of the RETAIN parm:

```
COMPUTE: SAVE-REGION-AMT = WHEN(RECTYPE='A') ASSIGN(REGION-AMT)
                           ELSE              RETAIN
```

When a type "A" record is read, the WHEN condition in this statement is true and the value from the REGION-AMT field is assigned to SAVE-REGION-AMT.

Then, as the "B" records (containing the normal SALES-FILE details records) are read, the WHEN parm is not true and no new value is assigned to SAVE-REGION-AMT. It simply retains the value it already has. That means we can now compute the percent of total. We simply divide the AMOUNT field from the detail record by the region total (which is retained in SAVE-REGION-AMT):

```
COMPUTE: PERCENT-REGION-AMOUNT = AMOUNT / SAVE-REGION-AMT * 100
```

Notice that this step has an INCLUDEIF statement that selects only type "B" records. That is because we did not want to print a line in the report for the type "A" header records. This INCLUDEIF statement does not interfere with saving the regional total values from the type "A" records. COMPUTE statements that use the RETAIN parm are always computed for *each* record in the input file (not just for those records that are included in the report.)

The rest of this step just consists of the standard COLUMNS, SORT and BREAK statements. Note in the COLUMNS statement that we used the NOACCUM parm for the percent of total fields. That tells Spectrum Writer not to "accumulate" the value of this field for the totals lines in the report.

Creating Multiple Reports in a Single Run

This section explains:

- how to create **more than one report** (or output file) during a single run of Spectrum Writer
- how to use the **NEWOUT statement**

One very powerful feature of Spectrum Writer is its ability to produce multiple reports during a single pass of the input file. This can greatly reduce the total amount of I/O processing, compared to producing the same reports in separate Spectrum Writer runs.

In multiple report runs, you have complete control over each individual report. For example, each report can have its own:

- criteria for including records from the input file (INCLUDEIF statement)
- sort order (SORT statement)
- control breaks (BREAK statements)
- data columns, column headings, format, etc. (COLUMNS statements)
- titles and/or footnotes (TITLE/FOOTNOTE statements)

- most OPTIONS statement options that affect a report's appearance

The only things that all reports in a single run will always have in common are:

- the primary input file (INPUT statement)
- a few OPTIONS statement options that relate to the processing of the input file or that apply to the run as a whole.

The NEWOUT Control Statement

The NEWOUT (or NEWOUTPUT) statement is used to request an additional report.

Put the NEWOUT statement after all of the control statements used to request the first report. The NEWOUT statement tells Spectrum Writer that you have finished one report request and are now starting to define a new output (that is, a new report or output file). The NEWOUT statement has no parms:

```
NEWOUT:
```

After the NEWOUT statement, you are now working on a fresh report. Nothing that you specified for any earlier report(s) is still in effect, except that:

- the primary input file and all of it's fields are available
- any auxiliary input files and all of their fields are available
- all computed fields are available

After the NEWOUT statement, just put the COLUMNS, SORT, TITLE, INCLUDEIF, etc. statements that you want for the new report. The control statements following a NEWOUT statement affect only the new report. They do not affect any earlier report(s).

If needed, you can also add any additional READ statements or COMPUTE statements required for the new report. Or, if you prefer, you can place those READ and COMPUTE statements back with the control statements for the first report. Their location will not affect the results of the run.

Here is an example of producing three reports from a single pass of the SALES-FILE.

```
INPUT: SALES-FILE
TITLE: 'SALES BY REGION'
COLUMNS: REGION CUSTOMER AMOUNT TAX
SORT: REGION
BREAK: REGION

NEWOUT:
TITLE: 'SALES BY CUSTOMER'
COLUMNS: CUSTOMER REGION AMOUNT TAX  EMPL-NAME
SORT: CUSTOMER
BREAK: CUSTOMER

NEWOUT:
TITLE: 'SALES OVER $100 SORTED BY DESCENDING AMOUNT'
INCLUDEIF: AMOUNT > 100
COLUMNS: AMOUNT CUSTOMER
SORT: AMOUNT(DESC)
```

The above statements request three separate reports from the SALES-FILE. (The blank lines before the NEWOUT statements are not required. We added them only to make the request more readable.) As you can see, each report has it's own title, sort order and columns. The

first report is sorted on REGION, with REGION control totals. The second report is sorted by CUSTOMER, with CUSTOMER control totals. The last report is sorted by descending AMOUNT, and has no control breaks. This last report includes only those records whose AMOUNT field contains a value greater than 100.

> **Note:** When producing multiple outputs, you will need to make some changes to your JCL. These changes are discussed in Chapter 8, "Operating System Considerations". See page 419 (OS/390) or page 435 (VSE).

How Many Reports Can You Request?

Each NWOUT statement results in one additional report. Spectrum Writer does not impose a limit on the number of NEWOUT statements that you can specify. You can produce as many reports in a single run as your system resources will allow.

Keep in mind that each report in a run does require additional system resources (in particular, storage). When a job needs more resources than are available, an ABEND can occur. For suggestions on maximizing storage resources, see page 420 (OS/390) or page 436 (VSE).

Chapter 5. How to Make a Web Report

Chapter Table of Contents

Chapter 5. How to Make a Web Report 293

How to Create a Web Report 294
Writing Your Own HTML Tags 296
Experimenting with HTML Tags 297
Customizing the Web Report's Titles 298
Customizing the Web Report's Data Columns 301
Customizing Control Breaks and Grand Totals 303
Putting Graphics in Your Web Report 305
Putting Graphics in Your Report Title 305
Putting Graphics in the Body of Your Report 306
Putting Graphics at Control Breaks 308
Putting Hot Links in your Web Report 308
Using HTML Tables in your Web Report 312
Using Dynamic HTML Tags 315
Using the PRESCRIPT and POSTSCRIPT Options 318
Summary of Options for Web Reports 320
Common HTML Tags 321

Chapter 5. How to Make a Web Report

In earlier chapters we've seen how to create custom reports with Spectrum Writer. In this chapter you will learn how Spectrum Writer can format your reports especially for viewing on Web browsers (such as Microsoft's Internet Explorer and Netscape's Navigator). You can put such reports on your company's Intranet or Internet site for easy company-wide (or public) viewing. Or, send the report to your colleagues as an e-mail attachment that they can automatically view from their e-mail reader. Spectrum Writer is a powerful tool in the move toward paperless "enterprise reporting."

As you will see, you can simply add one statement to any existing Spectrum Writer report to create a Web-viewable version of that report. But that's just the starting point in making attractive Web reports with Spectrum Writer. Advanced users — those familiar with the HTML language — can insert their own HTML commands into a Spectrum Writer report. These commands can take advantage of such Web features as:

- Custom fonts, font sizes, and colors, as well as bold, italic and underlined text.
- Special effects like animation, blinking text or text that scrolls marquee-like across the screen.
- Logos, graphics, charts, and photographs. For example, you could include employee photographs in a personnel directory. (See **Figure 43** on page 312 for an example of this.)
- "Hot links" that help viewers navigate within your report — or which let them jump to external Web pages.
- Playing audio or video clips. For example, a viewer could click on a product number in an inventory report and automatically see a video clip demonstrating the product in use!

Note: Before reading the rest of this chapter, you should already be familiar with creating regular Spectrum Writer reports. Chapter 2, "How to Request a Report" explains this.

How to Create a Web Report

Web pages can be formatted using a language called HTML. In order to properly view your Spectrum Writer report on the Web, certain HTML commands (also called "tags") must be added to the report. Without these HTML tags, the Web browser would "wrap" all of your report lines together into one big, jumbled "paragraph."

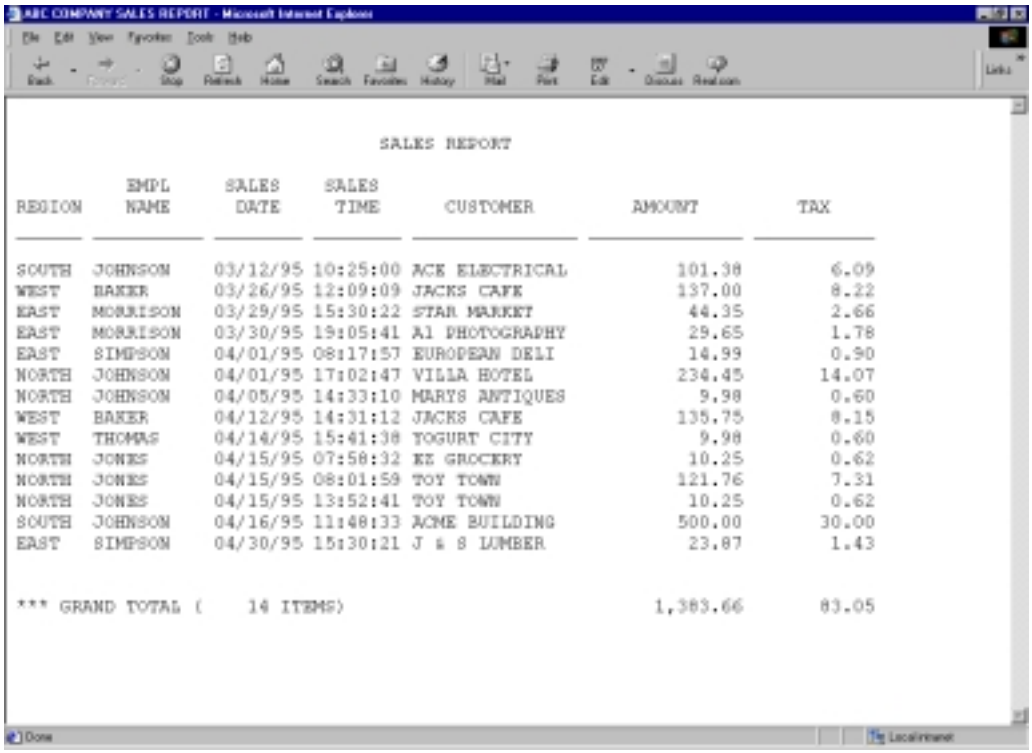
The HTML option tells Spectrum Writer to automatically add basic HTML information to your report:

```
OPTION: HTML
```

These Control Statements:

```
OPTION: HTML('ABC COMPANY SALES REPORT')
TITLE: 'SALES REPORT'
INPUT:  SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
```

Result in this Web Report:



SALES REPORT

REGION	EMPL NAME	SALES DATE	SALES TIME	CUSTOMER	AMOUNT	TAX
SOUTH	JOHNSON	03/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
WEST	BAKER	03/26/95	12:09:09	JACKS CAFE	137.00	8.22
EAST	MORRISON	03/29/95	15:30:22	STAR MARKET	44.35	2.66
EAST	MORRISON	03/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
EAST	SIMPSON	04/01/95	08:17:57	EUROPEAN DELI	14.99	0.90
NORTH	JOHNSON	04/01/95	17:02:47	VILLA HOTEL	234.45	14.07
NORTH	JOHNSON	04/05/95	14:33:10	MARYS ANTIQUES	9.98	0.60
WEST	BAKER	04/12/95	14:31:12	JACKS CAFE	135.75	8.15
WEST	THOMAS	04/14/95	15:41:38	YOGURT CITY	9.98	0.60
NORTH	JONES	04/15/95	07:58:32	EZ GROCERY	10.25	0.62
NORTH	JONES	04/15/95	08:01:59	TOY TOWN	121.76	7.31
NORTH	JONES	04/15/95	13:52:41	TOY TOWN	10.25	0.62
SOUTH	JOHNSON	04/16/95	11:48:33	ACME BUILDING	500.00	30.00
EAST	SIMPSON	04/30/95	15:30:21	J & S LUMBER	23.87	1.43
*** GRAND TOTAL (	14 ITEMS)				1,383.66	83.05

Figure 35. A basic Web report (viewed on Microsoft's Internet Explorer)

The above statement tells Spectrum Writer to add HTML tags to your report so that it will display properly on a Web browser. After that, all you do is route Spectrum Writer's output to a file which you can then upload to your Web site or other destination.

> **Tip:** Name your uploaded file using an extension of ".htm" or ".html". That tells Web browsers that your file is an HTML file.

The HTML option also lets you specify a title for your Web page, if you like:

```
OPTION: HTML('ABC COMPANY SALES REPORT')
```

The HTML title is different from the regular report titles. This special HTML title appears at the top of the Web browser's window as the name of your Web page.

Figure 35 shows a Web page created using the above statement.

Writing Your Own HTML Tags

For users who know the HTML language, Spectrum Writer lets you specify your own HTML tags directly within your Web report. By specifying your own HTML tags, you can create very impressive Web reports like the examples shown in the remainder of this chapter.

> **Note:** The rest of this chapter is intended for readers who are familiar with the HTML language.

"HTML tags" are formatting instructions that tell Web browsers how to format a portion of a Web page's text. HTML tags always begin with a "less than" sign and end with a "greater than" sign, like this <...>.

Many HTML tags are used in pairs. Their formatting information applies to all of the text between the opening HTML tag (for example, <B>) and the closing HTML tag (for example, </B>). Closing HTML tags consist of a slash and the first word (or abbreviation) of the opening tag. For example:

```
ABC COMPANY <B>SALES REPORT</B>
```

When the above text is displayed by a Web browser, it will look like this:

ABC COMPANY **SALES REPORT**

The words "**SALES REPORT**" are in bold letters. That is because those words are within the HTML "bold" tags. Notice that the HTML tags themselves are not displayed by Web browsers.

Other HTML tags are used by themselves. For example,
 specifies that a line "break" is wanted. It does not require a closing tag.

A list of common HTML tags that you might want to use begins on page 321.

Spectrum Writer's HTML option causes your report to be surrounded with just enough HTML information to make it viewable on the Web. To further customize your report for the Web,

Spectrum Writer lets you insert your own HTML tags (as character literals) directly within your report. Here are some of the places you might want to put your own HTML tags:

- in TITLE statements, to customize the report titles. For example:

```
TITLE: 'ABC COMPANY <B>SALES REPORT</B>'
```

- in COLUMNS statements, to customize one or more columns of data. For example:

```
COLUMNS: REGION EMPL-NAME '<B>' SALES-DATE '</B>' SALES-TIME CUSTOMER AMOUNT TAX
```

- in BREAK statements, to customize the lines printed at control breaks (or the Grand Total lines). For example:

```
BREAK: REGION NOTOTALS
       FOOTING('<B>TOTALS FOR' REGION 36 AMOUNT(TOTAL) TAX(TOTAL) '</B>')
```

- in COMPUTE statements, to create fields that contain dynamically built HTML tags. These fields can then be used in other control statements. For example:

```
COMPUTE: COLOR-TAG = WHEN(AMOUNT > 100) ASSIGN('<FONT COLOR=GREEN>')
                     ELSE               ASSIGN('<FONT COLOR=RED  >')
```

- in PRESCRIPT and POSTSCRIPT options, for formatting information that applies to the entire report. For example:

```
OPTION: PRESCRIPT('<FONT COLOR=BLUE>')
OPTION: POSTSCRIPT('</FONT>')
```

Experimenting with HTML Tags

Before we continue, a word of caution is in order. Formatting reports with HTML tags is more of an art than a science. For one thing, there are a number of different versions of HTML and it is constantly evolving as new versions of Web browsers are released. In addition, different browsers sometimes process the same HTML tag in slightly different ways. Furthermore, different preference settings in viewers' browsers can cause the same Web report to look different to different viewers. Finally, remember that the actual report that you see on the Web is the result of a two-step process. First Spectrum Writer formats a text-only report that contains your HTML tags mixed in with your report data. Then the Web browser strips the HTML tags out of the formatted report (sometimes throwing off Spectrum Writer's own carefully calculated alignment) and then reformats the report data that remains. All of this is to say that sometimes you will get quite unexpected results the first time you try out an HTML tag! But with enough experimentation, you can usually achieve the desired result.

> **Tip:** While refining a new Web report over and over again, it is easy for HTML syntax errors to creep into the results. There are resources on the Web that perform free syntax-checking of HTML files. This can be very helpful if you are not getting the results you expect from your HTML file. At the time of this writing, one such "HTML validator" service is available at www.w3.org.

The following sections show how to successfully use HTML tags in several specific areas of your reports.

Customizing the Web Report's Titles

You can customize a report title by including HTML tags within the title text in your TITLE statement. For example:

```
TITLE: '<H1>ABC COMPANY</H1>'
TITLE: '<H2>RECENT SALES</H2>'
```

Spectrum Writer will simply write out your HTML tags along with the rest of your title text. The Web browser will then recognize the HTML tags and format the text within the tags accordingly.

The <H1> and </H1> HTML tags in the first TITLE statement above tell the Web browser to format the title text within those tags as a "level 1 header." Thus, the person viewing your report on the Web will see "ABC COMPANY" in big, bold text at the top of each report page. They will not see the HTML tags themselves because the Web browser strips them away. Similarly, the second title will be formatted as a "level 2 header" (somewhat smaller than a level 1 header).

Here is another example. In this case, we make the first title red and the second title blue:

```
TITLE: '<H1><FONT COLOR=RED>ABC COMPANY</FONT></H1>'
TITLE: '<H2><FONT COLOR=BLUE>RECENT SALES</FONT></H2>'
```

The <Hn> tags cause the titles to be left-justified in some browsers. If you want the titles centered, just enclose them in <CENTER> tags:

```
TITLE: '<CENTER><H1><FONT COLOR=RED>ABC COMPANY</FONT></H1></CENTER>'
TITLE: '<CENTER><H2><FONT COLOR=BLUE>RECENT SALES</FONT></H2></CENTER>'
```

Figure 36 shows a report that uses the above TITLE statements.

Notice that the <CENTER> tags cause the titles to be centered within the Web browser window (not necessarily centered just over the report columns). You can always force your titles into any desired position by including padding blanks in the title text:

```
TITLE: '<H1><FONT COLOR=RED>          ABC COMPANY</FONT></H1>'
TITLE: '<H2><FONT COLOR=BLUE>     RECENT SALES BY REGION</FONT></H2>'
```

You can use HTML tags in the TITLE statement to change a title's font, size and color and to specify bold, underlined and italicized text. There are also HTML tags to make titles blink or scroll marquee-like. The appropriate HTML tags are listed in the table that begins on page 321.

Following are some additional suggestions for certain situations that may come up when customizing your Web report's titles.

To Avoid Report Titles in the Middle of Screens

Since a PC screen does not usually display as many report lines as a sheet of paper, it may take several PC screens to show a single "page" of a Spectrum Writer report. This means that the report titles and column headings may seem to appear randomly as viewers "page" through the report online.

These Control Statements:

```
OPTION: HTML('ABC COMPANY -- RECENT SALES')
TITLE:  '<CENTER><H1><FONT COLOR=RED>ABC COMPANY</FONT></H1></CENTER>'
TITLE:  '<CENTER><H2><FONT COLOR=BLUE>RECENT SALES</FONT></H2></CENTER>'
INPUT:  SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
```

Result in this Web Report:

Figure 36. A Web report with customized titles

Use the TITLEONCE option to just print the report titles and column headings once at the beginning of your report. This solves the problem of random titles and saves valuable screen space for actual data. For example:

```
OPTION: TITLEONCE
```

Syncing Report Titles with the Web Browser Screen

You may also be able to sync the Web browser window with the report pages by specifying "short" report pages. That would let viewers use their PC's Page Up and Page Down keys to page through your report, one screen per report page. Use Spectrum Writer's PAGELEN option to specify a page length of about 20 lines (experiment to get the exact number for your PC). Also add a blank FOOTNOTE statement to your report. (That forces trailing blank lines at the end of each page to ensure that all pages are exactly the same length.) For example:

```
OPTION:  PAGELEN(20)
FOOTNOTE:
```

Then adjust the size of your Web browser window as necessary to exactly match the length of your report pages. You can also adjust the font size in your Browser's "preferences" to fit more or fewer lines of the report on each screen.

Note: Be aware that even when this method puts the report pages in sync with *your* PC's Web browser window, it may still scroll differently on other PCs.

To Remove the Underscore Line from the Column Headings

Specify the NOUNDERSCORES option to eliminate the underscore line that appears after the column headings. This often looks better on PC screens and also saves valuable screen space for actual data. For example:

```
OPTION: NOUNDERSCORES
```

To Solve Alignment Problems

Remember, you can always use explicit spacing factors to force items into the place you desire. Also, specifying a single slash (/) at the end of the TITLE statement will prevent Spectrum Writer from trying to center- or right-justify your title. For example:

```
TITLE: '<H2>' 0 #TODAY 5 'SALES REPORT' 4 'PAGE' #PAGENUM '</H2>' /
```

The numbers in the above statement are spacing factors that tell Spectrum Writer how many spaces to put between each item in the title. The trailing slash (/) tells Spectrum Writer to leave the title left-justified.

Customizing the Web Report's Data Columns

You can put HTML tags (as literals) in the COLUMNS statement to customize individual columns of the report. For example, to make the AMOUNT and TAX columns bold, we could use this statement:

```
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER '<B>' AMOUNT TAX '</B>'
```

The above statement causes the literal text "<B>" to appear in each report line before the AMOUNT column. And the literal text "</B>" will appear as the last item on each line. Of course the Web browser will recognize these special HTML tags and process them as formatting instructions, rather than including them as part of the report shown to the viewer. The viewer won't see these literal texts in the online report, but the AMOUNT and TAX values will now appear in bold letters on the Web page.

The Web report produced by the above COLUMNS statement would have two blank spaces (instead of just one) between the CUSTOMER column and the AMOUNT column. That is because Spectrum Writer defaults to putting one blank space between all report columns. Thus, there would be one blank space after the CUSTOMER column and one blank space after the "<B>" column. The Web browser removes that HTML literal, but not the blank space after it. To solve this problem (which uses up valuable screen space), specify an override spacing factor of 0 as needed in COLUMNS statements that contain HTML literals:

```
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER '<B>' 0 AMOUNT TAX 0 '</B>'
```

Figure 37 uses the above statement to make the AMOUNT and TAX columns bold. Notice in that report that the *column headings* and *total values* for the AMOUNT and TAX fields are also bold. When the HTML (or HTMLAID) option is specified, Spectrum Writer automatically copies your COLUMNS statement HTML tags into the column heading lines and into the total lines for you. Thus, the formatting information that you specify for a data column is also applied to the column headings and total value (if any) for that column.

You can use HTML tags in the COLUMNS statement to change a column's font, size and color and to specify bold, underlined and italicized text. There are also HTML tags to make columns blink or scroll. The appropriate HTML tags are shown in the table that begins on page 321.

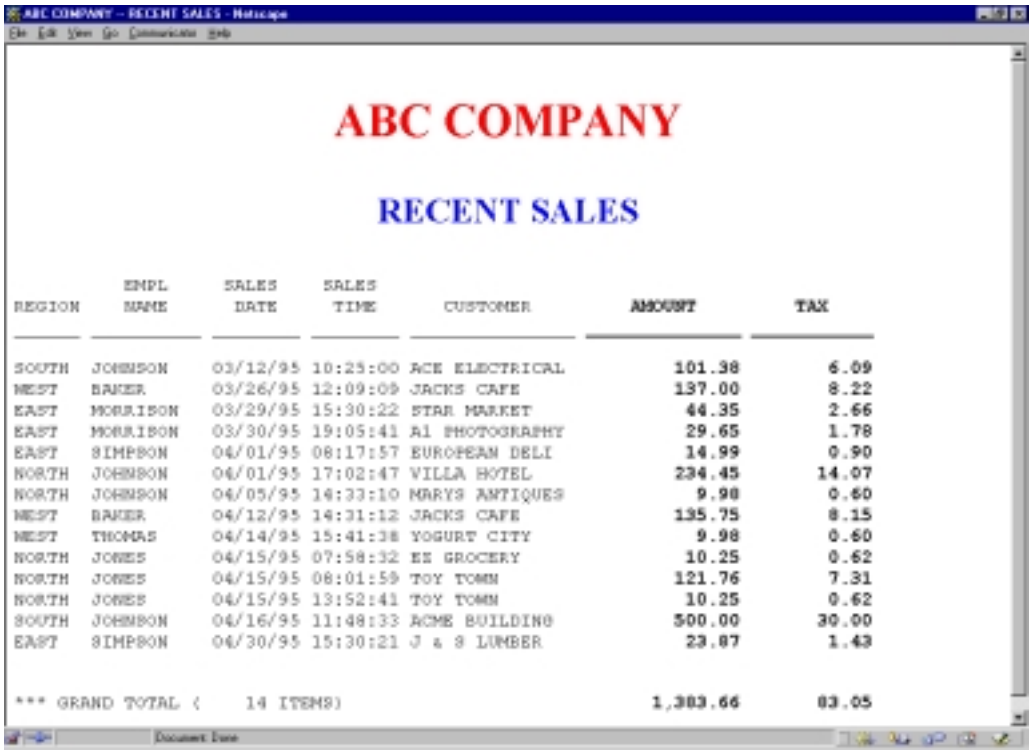
Using Other Fonts in Your Report

The HTML option causes Spectrum Writer to embed your entire report within <PRE> ("preformatted") tags. This tells the Web browser to preserve the report's line breaks and alignment. To accomplish this, the Web browser chooses a non-proportional font for your report. (A non-proportional font is one where all letters have exactly the same width.) If you override this by specifying the name of a proportional font (in a <FONT> tag in your COLUMNS statement), your report columns will probably not line up correctly. Therefore, it is usually best not to change the font of the body of the report. Or, if you do, be sure to change it to another non-proportional font. (On the other hand, it is generally safe to specify any kind of font for the report titles, since they are not in columnar format.)

These Control Statements:

```
OPTION: HTML('ABC COMPANY -- RECENT SALES')
TITLE: '<CENTER><H1><FONT COLOR=RED>ABC COMPANY</FONT></H1></CENTER>'
TITLE: '<CENTER><H2><FONT COLOR=BLUE>RECENT SALES</FONT></H2></CENTER>'
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME
         CUSTOMER '<B>' 0 AMOUNT TAX 0 '</B>'
```

Result in this Web Report:



ABC COMPANY

RECENT SALES

REGION	EMPL NAME	SALES DATE	SALES TIME	CUSTOMER	AMOUNT	TAX
SOUTH	JOHNSON	03/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
WEST	BAKER	03/26/95	12:09:09	JACKS CAFE	137.00	8.22
EAST	MORRISON	03/29/95	15:30:22	STAR MARKET	44.35	2.66
EAST	MORRISON	03/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
EAST	SIMPSON	04/01/95	08:17:57	EUROPEAN DELI	14.99	0.90
NORTH	JOHNSON	04/01/95	17:02:47	VILLA HOTEL	234.45	14.07
NORTH	JOHNSON	04/05/95	14:33:10	MARYS ANTIQUES	9.98	0.60
WEST	BAKER	04/12/95	14:31:12	JACKS CAFE	135.75	8.15
WEST	THOMAS	04/14/95	15:41:38	YOGURT CITY	9.98	0.60
NORTH	JONES	04/15/95	07:58:32	EZ GROCERY	10.25	0.62
NORTH	JONES	04/15/95	08:01:59	TOY TOWN	121.76	7.31
NORTH	JONES	04/15/95	13:52:41	TOY TOWN	10.25	0.62
SOUTH	JOHNSON	04/16/95	11:48:33	ACME BUILDING	500.00	30.00
EAST	SIMPSON	04/30/95	15:30:21	J & S LUMBER	23.87	1.43
*** GRAND TOTAL (	14 ITEMS)				1,383.66	83.05

Figure 37. A Web report with two bold columns

You can also put your own HTML tags in the lines printed at control breaks (including the Grand Total "control break"). For example, to make the whole total line at a control break appear in bold, underlined, italicized text, you could specify:

```
BREAK: REGION  NOTOTALS
       FOOTING('<B><U><I>TOTALS FOR' REGION 36 AMOUNT(TOTAL) TAX(TOTAL) '</I></U></B>')
```

The report in **Figure 38** uses the above BREAK statement.

Notice that we did not use the TOTAL parm to customize the total line text (as you would for a normal report). Instead, we specified NOTOTALS (to suppress the default total line) and then added a FOOTING parm to create our own total line. There are two reasons for this.

First, while the TOTAL parm would allow us to specify our *opening* HTML tags, it offers no way to specify the *closing* tags, since these must come after the totalled numeric columns.

Second, putting HTML tags in the TOTAL parm throws off the alignment of the totalled columns. The Web browser strips the HTML tags from the first part of the total line, which causes the totalled numeric columns to be shifted left.

By using a FOOTING parm (as in the statement above), you can specify both opening and closing HTML tags. You can also use an explicit spacing factor (the "36") to force the totalled columns into their proper location. Experimentation is the best way to determine the correct spacing factor for a particular report.

> **Tip:** When experimenting, use the MAXINCLUDE option to limit the number of records included in your report. This can greatly speed up your trial runs, especially if you're working with large input files.

HTML tags can also be used in the BREAK statement's HEADING parm. See **Figure 40** (page 309) for an example of this.

You can use HTML tags in the BREAK statement to change a total line's font, size and color and to specify bold, underlined and italicized text. There are also HTML tags to make text blink or scroll. The appropriate HTML tags are shown in the table that begins on page 321.

Following are some additional suggestions for certain situations that may come up when customizing your Web report's control breaks.

Putting a Divider Line at Control Breaks

You may want to use the <HR> tag to put a "horizontal rule" (line) in your report. For example, to add lines to a report to separate the regions, you could specify:

```
BREAK: REGION FOOTING('<HR>')
```

The report in **Figure 41** (page 310) has such a line at the control breaks.

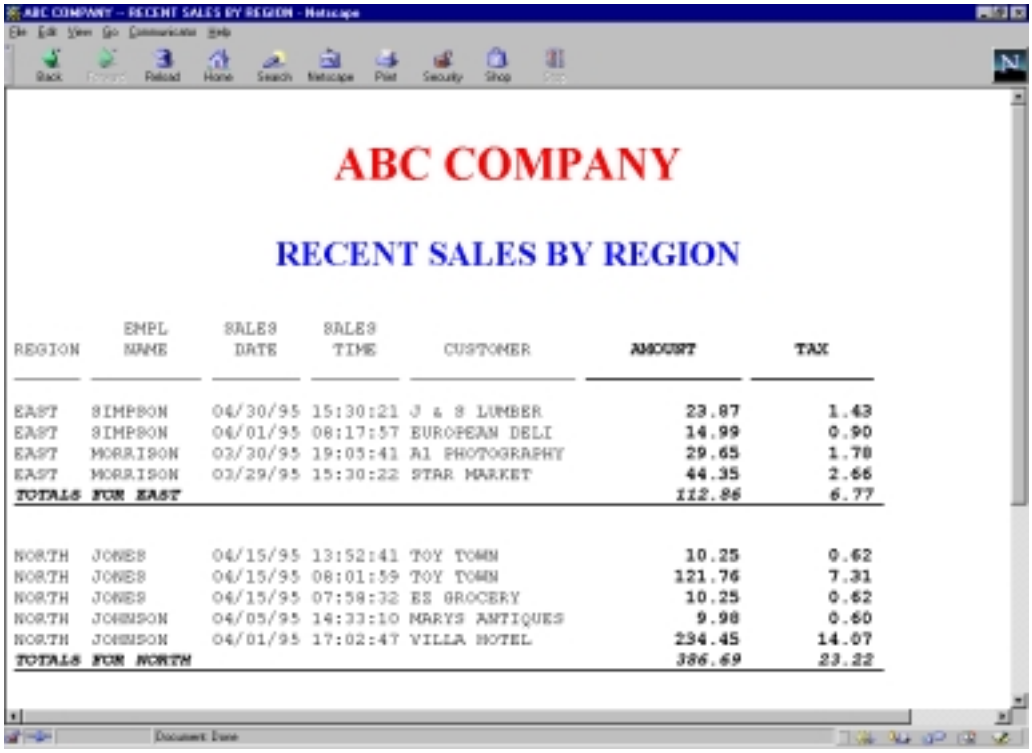
Why is the Total Line Split into Two Lines?

Including HTML tags in the COLUMNS statement sometimes causes the default total line to be split into two lines. This is because all HTML literals from the COLUMNS statement are also copied into the total lines. (This insures that if a report column is bold, for example,

These Control Statements:

```
OPTION: HTML('ABC COMPANY -- RECENT SALES BY REGION')
TITLE: '<CENTER><H1><FONT COLOR=RED>ABC COMPANY</FONT></H1></CENTER>'
TITLE: '<CENTER><H2><FONT COLOR=BLUE>RECENT SALES BY REGION' 0
       '</FONT></H2></CENTER>'
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME
         CUSTOMER '<B>' 0 AMOUNT TAX 0 '</B>'
SORT:  REGION
BREAK: REGION  NOTOTALS
       FOOTING('<B><U><I>TOTALS FOR' REGION 36 AMOUNT(TOTAL) TAX(TOTAL)
              '</I></U></B>')
```

Result in this Web Report:



ABC COMPANY

RECENT SALES BY REGION

REGION	EMPL NAME	SALES DATE	SALES TIME	CUSTOMER	AMOUNT	TAX
EAST	SIMPSON	04/30/95	15:30:21	J & S LUMBER	23.87	1.43
EAST	SIMPSON	04/01/95	08:17:57	EUROPEAN DELI	14.99	0.90
EAST	MORRISON	03/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
EAST	MORRISON	03/29/95	15:30:22	STAR MARKET	44.35	2.66
TOTALS FOR EAST					*112.86*	*6.77*
NORTH	JONES	04/15/95	13:52:41	TOY TOWN	10.25	0.62
NORTH	JONES	04/15/95	08:01:59	TOY TOWN	121.76	7.31
NORTH	JONES	04/15/95	07:58:32	ES GROCERY	10.25	0.62
NORTH	JOHNSON	04/05/95	14:33:10	MARYS ANTIQUES	9.98	0.60
NORTH	JOHNSON	04/01/95	17:02:47	VILLA HOTEL	234.45	14.07
TOTALS FOR NORTH					*386.69*	*23.32*

Figure 38. A Web report with customized total lines

the totals for that column will also be bold.) The problem occurs when there is not enough room to put the entire total line text before the first HTML tag. Spectrum Writer then puts the total line text on a separate line so that the HTML tags can appear in the correct location on the total line. If you want to force the totals onto a single line, specify the NOTOTALS option on your BREAK statement. Then use a FOOTING parm to specify exactly how your total line should look. The report in **Figure 38** demonstrates this.

Putting Graphics in Your Web Report

Use the <IMG> ("image") tag to display a graphic image in your Web report. The graphic might be a logo, a drawing, a photograph, a chart, a graph, etc. When you include an <IMG> tag in your report output, the Web browser will insert the image into the Web page right at that point.

The format of the <IMG> tag is:

```
<IMG SRC="url" [WIDTH=nnn] [ALIGN=TOP/MIDDLE/BOTTOM]>
```

The "url" identifies the source file that contains your graphic image. The graphic image file will usually be in GIF or JPEG format and will be named with an extension of ".gif" or ".jpg". The optional WIDTH parm specifies how wide (in pixels) the image should be. The optional ALIGN parm tells the Web browser how to vertically align any text that follows the image. The <IMG> tag also has other optional parms not shown above.

Putting Graphics in Your Report Title

Put an <IMG> tag in your TITLE statement to include a graphic among your titles. For example, to include a corporate logo along with a title at the top of each report page, we could use these statements:

```
TITLE: '<CENTER><IMG SRC="LOGO.GIF" WIDTH=150></CENTER>'
TITLE: '<H2><CENTER>SALES REPORT</CENTER></H2>'
```

Of course, for this to work your Web site must have a file named LOGO.GIF that contains your company's logo.

Figure 39 (page 307) uses TITLE statements similar to the ones shown above. (So do the reports on page 312 and page 317.)

You can also put an <IMG> tag in the same TITLE statement as your title text. For example:

```
TITLE: '<IMG SRC="LOGO.GIF" ALIGN=MIDDLE WIDTH=50>SALES REPORT'
```

Notice the ALIGN=MIDDLE parm in the above <IMG> tag. It tells the Web browser to align the subsequent text with the middle (height wise) of the graphic.

A more powerful (but more complicated) way of combining text and graphics in the title is to use an HTML table. That technique is discussed under "Using HTML Tables in your Web Report" (page 312).

You can also put <IMG> tags in your COLUMNS statement to include a graphic in each detail line of the report. In other words, your report can have one or more columns of graphics, instead of text. In this case, of course, you do not want to show the *same* graphic in every detail line. You want a graphic that is related to the data in the rest of that particular report line. Use a COMPUTE statement to dynamically build an appropriate <IMG> tag for each report line.

For example, assume that we want to produce a personnel directory for the Web. We want the directory to include each employee's photograph. Further assume that our Web site has photographs of the employee's stored in JPG files. The photograph files are named EMPnnn.JPG, where nnn is the employee's employee number. We would use the following COMPUTE statement to build the correct <IMG> tag for the report detail lines:

```
COMPUTE: PICTURE-TAG = '<IMG SRC="EMP' + EMPL-NUM + '.JPG" WIDTH=60>'
```

The COMPUTE statement above builds a unique <IMG> tag for each employee, based on their employee number. For example, the PICTURE-TAG field for employee number 044 will contain this text: <IMG SRC="EMP044.JPG" WIDTH=60>.

We can then use this PICTURE-TAG field in our COLUMNS statement:

```
COLUMNS: PICTURE-TAG LAST-NAME FIRST-NAME EMPL-NUM HIRE-DATE
```

The first column of the resulting Web report is a column of employee pictures. **Figure 39** uses the above statements.

Notice how we handled the column headings in **Figure 39**. We specified NOCOLHDGS to suppress the default column headings. Then we used extra TITLE statements to build our own column headings. (We also used an empty TITLE statement to put a blank line between the report titles and the column headings.)

Why didn't we just use Spectrum Writer's default column headings? Because adding a "picture" column to a report tends to throw off the alignment of all of the following column headings (and of the total values, as well). Remember that when Spectrum Writer formats its report, the "picture" column just contains text (the <IMG> tag) like any other column. Later, the Web browser strips away these <IMG> tags in the detail lines and replaces them with actual images. Those images (photographs, in this example) take up a different amount of space than the <IMG> tag took up. Since no similar substitution takes place in the column heading lines (or in the total lines) their spacing no longer matches the detail line spacing.

When including graphics in the body of your report, you will probably want to use this technique to control exactly how your column headings look. Similarly, use the BREAK statement FOOTING parm to precisely control how your total lines look (as we also did in **Figure 38** on page 304).

These Control Statements:

```
OPTION: HTML('ABC COMPANY PERSONNEL DIRECTORY')
OPTION: NOCOLHDGS
INPUT: EMPL-FILE
TITLE: '<CENTER><IMG SRC="ABCLOGO2.GIF"></CENTER>'
TITLE: '<CENTER><FONT FACE="LUCIDA HANDWRITING" SIZE=5>' 0
       'ONLINE PERSONNEL DIRECTORY</FONT></CENTER>'
TITLE:
TITLE: '                                          EMPL   HIRE' /
TITLE: 'PHOTO    LAST NAME       FIRST NAME     NUM.   DATE'
       '  CITY            ST  ZIP' /
COMPUTE: PICTURE-TAG='<IMG SRC="EMP' + EMPL-NUM + '.JPG" WIDTH=60>'
COLUMNS: PICTURE-TAG LAST-NAME FIRST-NAME EMPL-NUM HIRE-DATE
         CITY STATE ZIP
SORT:    LAST-NAME
```

Result in this Lotus 1-2-3 Spreadsheet:

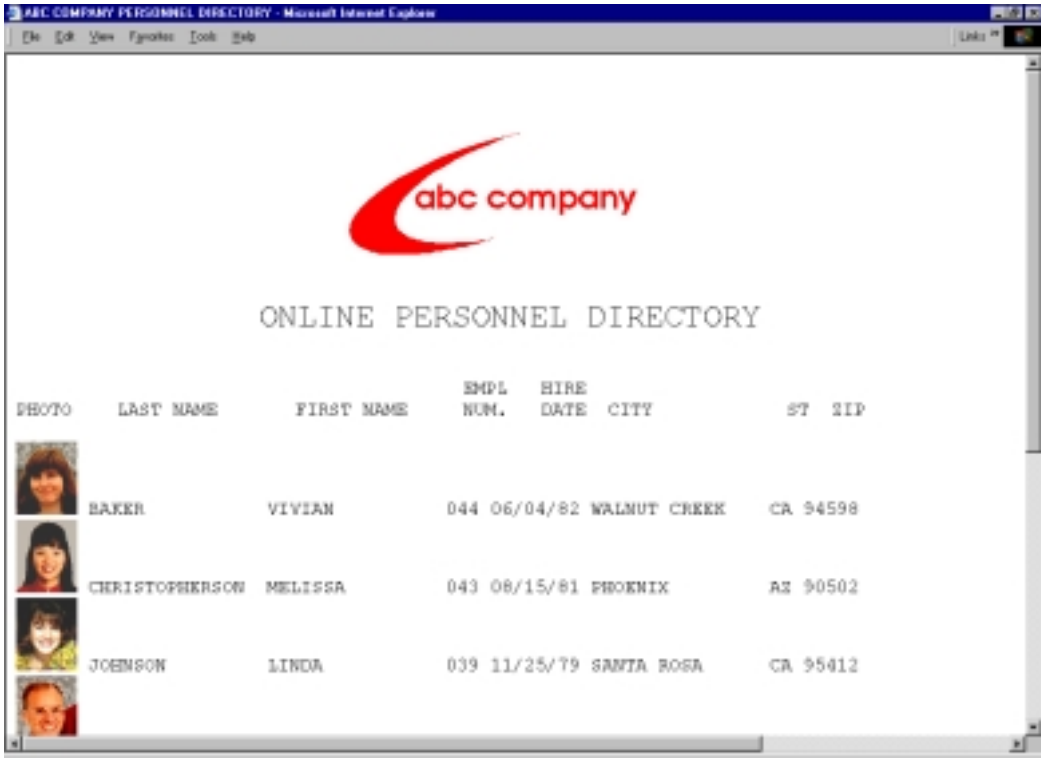


Figure 39. A Web report containing graphics in the title and body

Putting Graphics at Control Breaks

You can also use <IMG> tags in the BREAK statement, to place graphics at your report's control breaks. The report in **Figure 41** (page 310) shows an example of this.

Putting Hot Links in your Web Report

Hot links (or "hypertext links") are areas of your Web report that viewers can click on to obtain some action. For example, clicking on a hot link might cause the browser to jump to a different part of the report (the Grand Totals, for example). Or it might jump to a different Web page altogether (for example, to a Web page describing a particular product in a sales report).

Hot links can also invoke an audio player or even a video clip player. Viewers could click on a product name in your report, for example, and automatically be shown a video demonstrating that product in use.

To create a hot link in your report, surround your "hot text" (that is, the text that the viewer should click on) with the <A HREF="url" > and </A> ("anchor") tags. The url in the tag will determine what action is taken when the viewer clicks on the text.

If the url is the name of a Web page (for example, an HTML file), the browser will jump to that Web page when the viewer clicks on the hot link:

```
TITLE: '<A HREF="WWW.ABC.COM/INDEX.HTML">' 0
       'CLICK HERE TO READ MORE ABOUT THE ABC COMPANY</A>'
```

If the url is the name of a label within an HTML file (even within the Web report itself) the browser will jump to that specific location within that Web page:

```
TITLE: '<A HREF=#EAST>CLICK HERE TO JUMP TO EAST REGION</A>'
```

If the url is the name of an audio file (such as a .WAV, .AU or .MID file), the browser will play that sound file when the viewer clicks the hot link:

```
TITLE: '<A HREF="WELCOME.WAV">' 0
       'CLICK HERE TO HEAR THE CHAIRMAN''S GREETING</A>'
```

If the url is the name of a video clip (such as an .AVI file) the browser will play that video clip when the viewer clicks the hot link.

```
TITLE: '<A HREF="PRODUCT.AVI">' 0
       'CLICK HERE TO SEE A VIDEO ABOUT ABC COMPANY''S PRODUCTS</A>'
```

As mentioned, hot links can be used to let viewers easily move around within the report itself. The report in **Figure 41** (page 310) shows an example of using hot links in this way. Viewers can click on one of the hot texts at the beginning of the report to go directly to the region they are interested in, or to the Grand Totals. And at the bottom of the report, there is a another hot link to take them back to the top of the report.

```
OPTION: HTML('ABC COMPANY -- SALES BY REGION')
   NOUNDERSCORES
   PRESCRIPT('<A NAME=TOP>')
   PRESCRIPT('<A HREF=#EAST>CLICK HERE TO JUMP TO EAST REGION</A>')
   PRESCRIPT('<A HREF=#NORTH>CLICK HERE TO JUMP TO NORTH REGION</A>')
   PRESCRIPT('<A HREF=#SOUTH>CLICK HERE TO JUMP TO SOUTH REGION</A>')
   PRESCRIPT('<A HREF=#WEST>CLICK HERE TO JUMP TO WEST REGION</A>')
   PRESCRIPT('<A HREF=#GRAND>CLICK HERE TO JUMP TO GRAND TOTALS</A>')
   POSTSCRIPT('<A HREF=#TOP>CLICK HERE TO RETURN TO TOP OF REPORT</A>')
TITLE:'<H1><CENTER>ABC COMPANY SALES BY REGION</CENTER></H1>'
INPUT:  SALES-FILE
COMPUTE: REGION-IMG = '<IMG SRC=REG' + #LEFT(REGION,1) +
                      '.GIF ALIGN=MIDDLE>'
COLUMNS: REGION(NOREPEAT) EMPL-NAME SALES-DATE SALES-TIME
         CUSTOMER AMOUNT TAX
SORT:    REGION
BREAK:   REGION  NOTOTALS
 HEADING('<A NAME=' 0 REGION 0 '>')
 HEADING(REGION-IMG)
 HEADING(' ')
 FOOTING(' ')
 FOOTING('<FONT SIZE=5><B><I>' 0
         'TOTALS FOR' REGION 'REGION' 7 AMOUNT(TOTAL) TAX(7,TOTAL)
         '</I></B></FONT>')
 FOOTING('<HR>')
BREAK: #GRAND
 FOOTING('<A NAME=GRAND>')
```

Figure 40. Control statements used to create a Web report with “hot links”

Figure 40 shows the control statements used to create the report in **Figure 41**. As you can see in **Figure 40**, we used PRESCRIPT options to put five hot link texts ahead of the report. (The PRESCRIPT option puts lines of text before the beginning of the report itself.) The urls in these links are HTML "labels" within our own report named EAST, NORTH, SOUTH, WEST and GRAND. We used a POSTSCRIPT option to write one other hot link text at the end of the report. It references a label called TOP.

To make these links work, the output must contain the six labels (TOP, EAST, NORTH, etc.) at the appropriate place within the report. HTML labels are assigned with <A NAME=label> tags. We used another PRESCRIPT option to put the TOP label at the top of the report. And we used BREAK statement HEADING parms to put the four regional labels at the beginning of their respective control groups. Finally we used the Grand Total control break FOOTING parm to supply the GRAND label at the Grand Total lines.

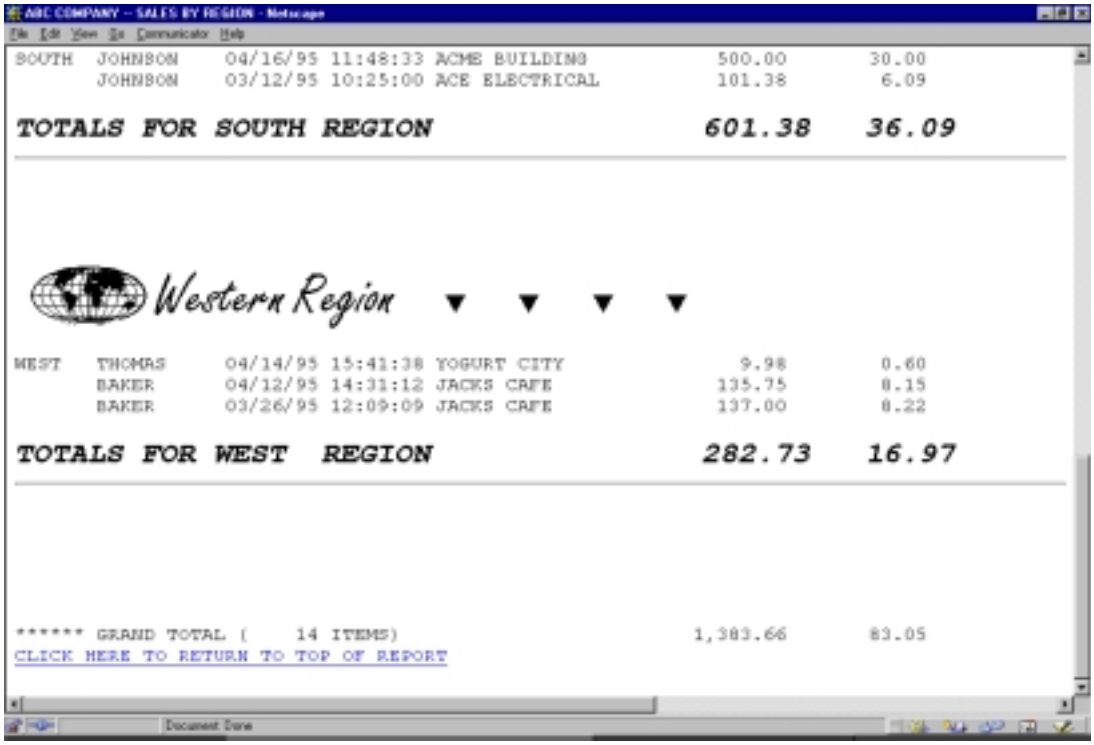
Figure 42 (page 311) shows the actual HTML output file created by Spectrum Writer.

> **Note:** We were able to provide hot links to each control group in this report because we knew ahead of time all the values that the REGION variable would have. This particular technique can only be used when you know the values of your control break variable in advance.

CLICK HERE TO JUMP TO EAST REGION
CLICK HERE TO JUMP TO NORTH REGION
CLICK HERE TO JUMP TO SOUTH REGION
CLICK HERE TO JUMP TO WEST REGION
CLICK HERE TO JUMP TO GRAND TOTALS

ABC COMPANY SALES BY REGION

REGION	EMPL NAME	SALES DATE	SALES TIME	CUSTOMER	AMOUNT	TAX
Eastern Region ▼ ▼ ▼ ▼						
EAST	SIMPSON	04/30/95	15:30:21	J & S LUMBER	23.87	1.43
	SIMPSON	04/01/95	08:17:57	EUROPEAN DELI	14.99	0.90
	MORRISON	03/30/95	19:05:41	A1 PHOTOGRAPHY	29.65	1.78
	MORRISON	03/29/95	15:30:22	STAR MARKET	44.35	2.66
TOTALS FOR EAST REGION					***112.86***	***6.77***



SOUTH	JOHNSON	04/16/95	11:48:33	ACME BUILDING	500.00	30.00
	JOHNSON	03/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
TOTALS FOR SOUTH REGION					***601.38***	***36.09***
Western Region ▼ ▼ ▼ ▼						
WEST	THOMAS	04/14/95	15:41:38	YOGURT CITY	9.98	0.60
	BAKER	04/12/95	14:31:12	JACKS CAFE	135.75	8.15
	BAKER	03/26/95	12:09:09	JACKS CAFE	137.00	8.22
TOTALS FOR WEST REGION					***282.73***	***16.97***
****** GRAND TOTAL (14 ITEMS)					1,383.66	83.05

CLICK HERE TO RETURN TO TOP OF REPORT

Figure 41. Two screens from a Web report with “hot links”

```
<HTML>
<HEAD>
<TITLE>ABC COMPANY -- SALES BY REGION</TITLE>
</HEAD>
<BODY>
<PRE>
<A NAME=TOP>
<A HREF=#EAST>CLICK HERE TO JUMP TO EAST REGION</A>
<A HREF=#NORTH>CLICK HERE TO JUMP TO NORTH REGION</A>
<A HREF=#SOUTH>CLICK HERE TO JUMP TO SOUTH REGION</A>
<A HREF=#WEST>CLICK HERE TO JUMP TO WEST REGION</A>
<A HREF=#GRAND>CLICK HERE TO JUMP TO GRAND TOTALS</A>
            <H1><CENTER>ABC COMPANY SALES BY REGION</CENTER></H1>

          EMPL     SALES    SALES
REGION    NAME      DATE     TIME      CUSTOMER         AMOUNT         TAX

<A NAME=EAST >
<IMG SRC=REGE.GIF ALIGN=MIDDLE>

EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER             23.87        1.43
       SIMPSON    04/01/95 08:17:57 EUROPEAN DELI            14.99        0.90
       MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY           29.65        1.78
       MORRISON   03/29/95 15:30:22 STAR MARKET              44.35        2.66

<FONT SIZE=5><B><I>TOTALS FOR EAST  REGION               112.86    6.77 </I></B></FONT>
<HR>


<A NAME=NORTH>
<IMG SRC=REGN.GIF ALIGN=MIDDLE>

NORTH  JONES      04/15/95 13:52:41 TOY TOWN                10.25        0.62
       JONES      04/15/95 08:01:59 TOY TOWN               121.76        7.31
       JONES      04/15/95 07:58:32 EZ GROCERY              10.25        0.62
       JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES           9.98        0.60
       JOHNSON    04/01/95 17:02:47 VILLA HOTEL            234.45       14.07

<FONT SIZE=5><B><I>TOTALS FOR NORTH REGION               386.69   23.22 </I></B></FONT>
<HR>
                              (additional lines not shown)

<A NAME=WEST >
<IMG SRC=REGW.GIF ALIGN=MIDDLE>

WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY              9.98        0.60
       BAKER      04/12/95 14:31:12 JACKS CAFE             135.75        8.15
       BAKER      03/26/95 12:09:09 JACKS CAFE             137.00        8.22

<FONT SIZE=5><B><I>TOTALS FOR WEST  REGION               282.73   16.97 </I></B></FONT>
<HR>



<A NAME=GRAND>
****** GRAND TOTAL (    14 ITEMS)                         1,383.66       83.05
<A HREF=#TOP>CLICK HERE TO RETURN TO TOP OF REPORT</A>
</PRE>
</BODY>
</HTML>
```

Figure 42. the HTML output file for a Web report with “hot links”

Using HTML Tables in your Web Report

If you like, you can use HTML tags to format your Web report as a table. HTML tables are often used as a tool for organizing data on a Web page. One common use of tables is to align multiple lines of text with a single image.

HTML tables consist of a number of rows. Within each row is one or more data cells (which form the columns). Each data cell can contain report text and/or an image.

Surround your HTML table with <TABLE> and </TABLE> tags. Use the <TR> tag to start a new table row. Use the <TD> tag to start each new data cell within a row.

The Web report in **Figure 43** incorporates two separate HTML tables. (**Figure 44** (page 313) shows the control statements used to create this report. **Figure 45** (page 314) shows the actual HTML output file created by Spectrum Writer for this report.)

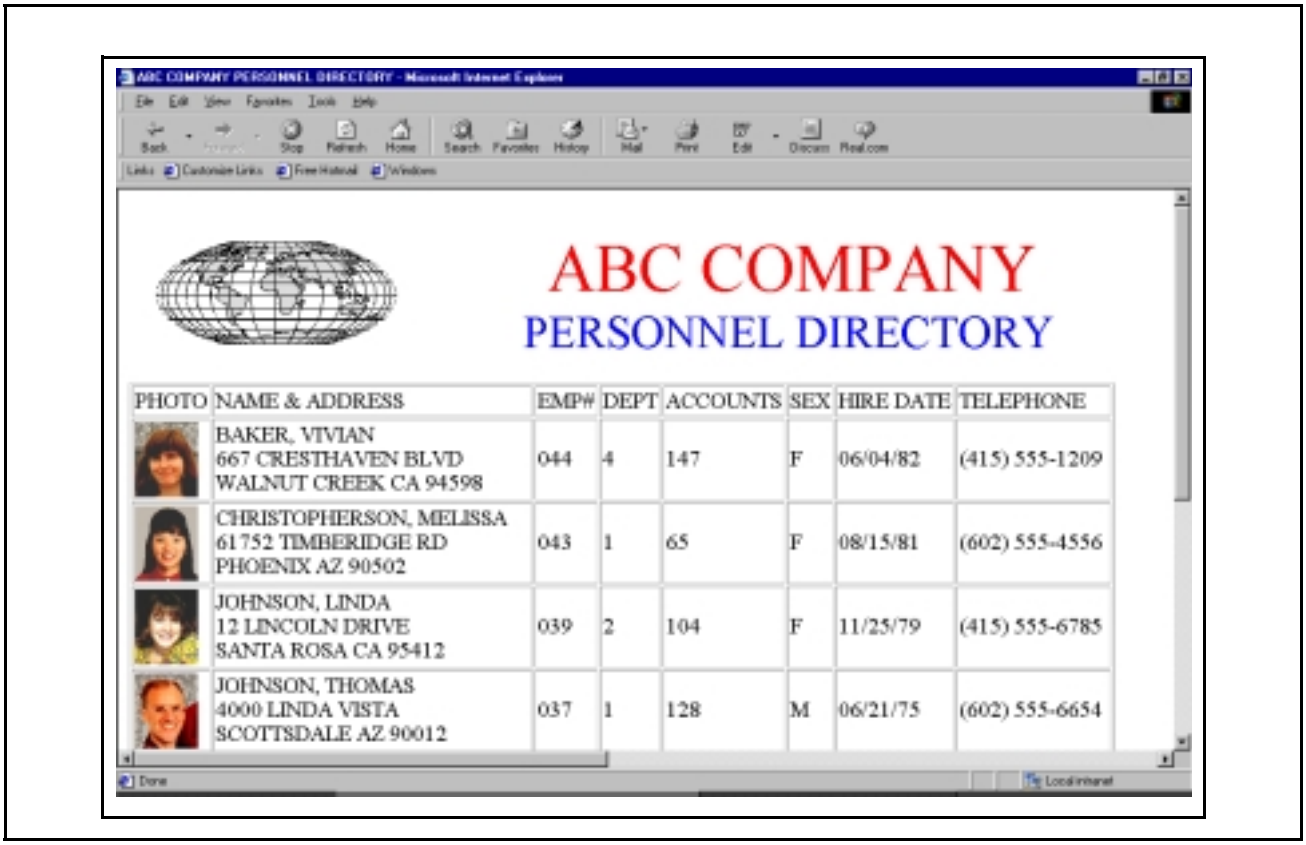


Figure 43. A Web report that uses “tables”

The first table contains only the title information. (This table was drawn without borders, so it doesn’t look like a typical table.) We used the table as a convenient way to align multiple lines of text with the logo graphic. This table has just one row. That row contains two cells. The first cell contains the logo image. The second cell contains the two lines of title text. Here are the statements we used to build this table:

```
PRESCRIPT('<TABLE><TR><TD><IMG SRC="ABC.GIF" WIDTH=275 ALIGN=MIDDLE>')
PRESCRIPT('<TD ALIGN=CENTER><FONT COLOR=RED SIZE=7 FACE="CUPERTINO">')
PRESCRIPT('ABC COMPANY</FONT>')
```

```
PRESCRIPT('<FONT COLOR=BLUE SIZE=6 FACE=BASSOON>PERSONNEL DIRECTORY')
PRESCRIPT('</FONT></TABLE>')
```

Notice that we used PRESCRIPT options to write out this HTML code once at the beginning of the report. That means that our title will only appear once at the top of the whole report. By specifying the NOTITLES option (see **Figure 44**), we suppressed the regular report titles at the top of each page.

```
OPTION: NOTITLES  NOGRANDTOTALS COLSPACE(0)
 HTML('ABC COMPANY PERSONNEL DIRECTORY')
 PRESCRIPT('<TABLE><TR><TD><IMG SRC=WORLD.GIF WIDTH=275 ALIGN=MIDDLE>')
 PRESCRIPT('<TD ALIGN=CENTER><FONT COLOR=RED SIZE=7 FACE="CUPERTINO">')
 PRESCRIPT('ABC COMPANY</FONT>')
 PRESCRIPT('<FONT COLOR=BLUE SIZE=6 FACE=BASSOON>PERSONNEL DIRECTORY')
 PRESCRIPT('</FONT></TABLE>')
 PRESCRIPT('<TABLE BORDER=1 CELLPADDING=2>')
 PRESCRIPT('<TR><TD>PHOTO<TD>NAME & ADDRESS<TD>EMP#<TD>DEPT<TD>ACCOUNTS<
 TD>SEX<TD>HIRE DATE<TD>TELEPHONE')
 POSTSCRIPT('</TABLE>')
INPUT:  EMPL-FILE
COMPUTE: PICTURE-TAG = '<IMG SRC=EMP' + EMPL-NUM + '.JPG WIDTH=60>'
COMPUTE: NAME = #COMPRESS(LAST-NAME 0 ',' 1 FIRST-NAME)
SORT:    NAME
COLUMNS: '<TR><TD>' PICTURE-TAG
         '<TD>' NAME '<BR>' ADDRESS '<BR>' CITY 1 STATE 1 ZIP
         '<TD>' EMPL-NUM
         '<TD>' DEPT-NUM
         '<TD>' NUM-ACCOUNTS
         '<TD>' SEX
         '<TD>' HIRE-DATE
         '<TD>' TELEPHONE
```

Figure 44. Control statements used to create a Web report with “tables”

In the PRESCRIPT statements above, the <TABLE> tag defines the beginning of the table. Since we did not specify a "BORDER=n" parm in this tag, this table has no visible border or grid lines. The first (and only) row in this table is defined with the <TR> tag. The <TD> tag marks the beginning of the first cell in this row. In this cell, we just specified the <IMG> tag for the company's logo graphic. We then started the second cell with another <TD> tag. Within this cell are two lines of text. We put different <FONT> tags around each line to make them different colors, fonts and sizes. Finally, we ended the table by specifying the </TABLE> tag.

The second table in our report contains the main body of the report. We also used PRESCRIPT options to write the HTML code for the *initial part* of this table:

```
PRESCRIPT('<TABLE BORDER=1 CELLPADDING=2>')
PRESCRIPT('<TR><TD>PHOTO<TD>NAME & ADDRESS<TD>EMP#<TD>DEPT<TD>ACCOUNTS<
 TD>SEX<TD>HIRE DATE<TD>TELEPHONE')
```

For this table, we chose to show grid lines by specifying BORDER=1 in the <TABLE> tag. The first row of our table, also defined by a PRESCRIPT option, contains our table's column headings. (Notice that this PRESCRIPT text was too long to fit onto a single line. We continued the literal by typing up to column 72 in the first line, and then continuing in column 2 of the next line.)

```
<HTML>
<HEAD>
<TITLE>ABC COMPANY PERSONNEL DIRECTORY</TITLE>
</HEAD>
<BODY>
<PRE>
<TABLE><TR><TD><IMG SRC=WORLD.GIF WIDTH=275 ALIGN=MIDDLE>
<TD ALIGN=CENTER><FONT COLOR=RED SIZE=7 FACE="CUPERTINO">
ABC COMPANY</FONT>
<FONT COLOR=BLUE SIZE=6 FACE=BASSOON>PERSONNEL DIRECTORY
</FONT></TABLE>
<TABLE BORDER=1 CELLPADDING=2>
<TR><TD>PHOTO<TD>NAME & ADDRESS<TD>EMP#<TD>DEPT<TD>ACCOUNTS<TD>SEX<TD>HIRE DATE<TD>TELEPHONE
<TR><TD><IMG SRC=EMP044.JPG WIDTH=60><TD>BAKER, VIVIAN <BR>667 CRESTHAVEN BLVD <BR>WALNUT CREEK CA 94598<TD>044<TD> 4<TD> 147<TD>F<TD>06/04/82<TD>(415) 555-1209
<TR><TD><IMG SRC=EMP043.JPG WIDTH=60><TD>CHRISTOPHERSON, MELISSA <BR>61752 TIMBERIDGE RD <BR>PHOENIX AZ 90502<TD>043<TD> 1<TD> 65<TD>F<TD>08/15/81<TD>(602) 555-4556
<TR><TD><IMG SRC=EMP039.JPG WIDTH=60><TD>JOHNSON, LINDA <BR>12 LINCOLN DRIVE <BR>SANTA ROSA CA 95412<TD>039<TD> 2<TD> 104<TD>F<TD>11/25/79<TD>(415) 555-6785
<TR><TD><IMG SRC=EMP037.JPG WIDTH=60><TD>JOHNSON, THOMAS <BR>4000 LINDA VISTA <BR>SCOTTSDALE AZ 90012<TD>037<TD> 1<TD> 128<TD>M<TD>06/21/75<TD>(602) 555-6654
<TR><TD><IMG SRC=EMP036.JPG WIDTH=60><TD>JONES, JERRY <BR>125 MAIN STREET <BR>SAN FRANCISCO CA 94012<TD>036<TD> 2<TD> 78<TD>M<TD>01/31/80<TD>(415) 555-7653
<TR><TD><IMG SRC=EMP040.JPG WIDTH=60><TD>MACDONALD, RICHARD <BR>525 FOOTHILL DRIVE <BR>PLEASANTON CA 94568<TD>040<TD> 2<TD> 6<TD>M<TD>07/04/82<TD>(415) 555-9887
<TR><TD><IMG SRC=EMP042.JPG WIDTH=60><TD>MORRISON, MICHAEL <BR>98 SOUTH LAKESIDE DR<BR>GLENDALE CA 91202<TD>042<TD> 3<TD> 154<TD>M<TD>11/30/79<TD>(818) 555-4748
<TR><TD><IMG SRC=EMP041.JPG WIDTH=60><TD>SIMPSON, TIMOTHY <BR>89876 WEST 53 STREET<BR>ARCADIA CA 91006<TD>041<TD> 3<TD> 16<TD>M<TD>12/01/82<TD>(818) 555-1887
<TR><TD><IMG SRC=EMP045.JPG WIDTH=60><TD>THOMAS, MARTIN <BR>77812 S. HUNTINGTON <BR>CONCORD CA 94519<TD>045<TD> 4<TD> 118<TD>M<TD>06/04/82<TD>(415) 555-1152
</TABLE>
</PRE>
</BODY>
</HTML>
```

Figure 45. HTML file for a Web report with "tables"

We used a POSTSCRIPT option to specify the *closing* </TABLE> tag:

```
POSTSCRIPT('</TABLE>')
```

The POSTSCRIPT text will appear after the entire report. It simply closes the main report table.

Between the PRESCRIPT lines and the POSTSCRIPT line will come the actual report lines specified by the COLUMNS statement:

```
COLUMNS: '<TR><TD>' PICTURE-TAG
         '<TD>' NAME '<BR>' ADDRESS '<BR>' CITY 1 STATE 1 ZIP
         '<TD>' EMPL-NUM
         '<TD>' DEPT-NUM
         '<TD>' NUM-ACCOUNTS
         '<TD>' SEX
         '<TD>' HIRE-DATE
         '<TD>' TELEPHONE
```

The COLUMNS statement above begins with a literal containing a <TR> tag. That means that each detail line of our report will begin with a <TR> tag and become a new table row. We then used <TD> tags to put each data item into its own separate cell — with one exception. In the "Name & Address" table column, we included multiple data items in the same cell. We used
 (break) tags to specify where a new line should be started *within* the cell. Note that we used spacing factors of "1" between the fields within the same cell. That overrides the default spacing factor of "0" that we specified with the COLSPACE(0) option (see page 313.) We specified the COLSPACE(0) option just to keep from having to type 0's around all of the HTML tag literals in the COLUMNS statement. The PICTURE-TAG field used in the above statement was discussed on page 306.

Using Dynamic HTML Tags

When you put a literal containing an HTML tag in your COLUMNS statement, that tag appears in all of the report detail lines. (It is also copied into the column heading lines and control break total lines.) What if you want to use a different HTML tag for different report lines? You can build dynamic HTML tags to do that.

Dynamic HTML tags are tags whose contents vary, depending on the other data in the report line. Dynamic HTML tags are assigned to COMPUTE fields instead of being specified as literals.

We have already seen some examples of dynamic HTML tags. The reports on page 307 and page 312 built dynamic <IMG> tags for each employee's photograph. And on page 310 we built dynamic <IMG> tags to display regional logos at the beginning of each region's data.

Now let's look at another use of dynamic HTML tags. Assume that we want our Web report to show sales amounts that are over $100 in green and all other sales amounts in red. Instead of putting a literal <FONT> tag in our COLUMNS statement, we would use these statements to compute a dynamic <FONT> tag:

```
COMPUTE: COLOR-TAG = WHEN(AMOUNT > 100) ASSIGN('<FONT COLOR=GREEN>')
                     ELSE               ASSIGN('<FONT COLOR=RED  >')
```

Notice that we padded the "red" tag with enough blanks to make it the same size as the "green" tag. This is important to keep the resulting report properly aligned. Remember that the Web browser strips all HTML tags from the report before displaying it to the viewer. To preserve column alignment, the same number of bytes must be stripped from *all* report lines.

In our COLUMNS statement, we now use this COLOR-TAG field instead of a literal <FONT> tag:

```
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER
         COLOR-TAG('<!--xxxxxxxxxxx-->') 0 AMOUNT TAX 0 '</FONT>'('<!---->')
```

In the report detail lines, either "<FONT COLOR=GREEN>" or "<FONT COLOR=RED >" will now appear before the AMOUNT field. (You can see this on page 318.) In both cases, the same closing tag ("</FONT>") appears after the TAX column.

The report in **Figure 46** uses the above statements.

Notice the unusual column heading for the COLOR-TAG column. Tags that begin with "<!--" and end with "-->" are considered HTML "comment" tags. You can place any number of other characters between these tags. Comment tags do not affect the formatting of the Web report in any way. However, they are stripped from the Web page like all other HTML tags.

This special column heading was needed to preserve correct alignment of the report. Remember that when you put an HTML tag *literal* in the COLUMNS statement, Spectrum Writer automatically copies the literal tag into all of the column heading lines as well. That ensures that the same amount of HTML text is stripped from the column headings as from the detail lines, which allows the column headings to remain aligned over the data.

However, Spectrum Writer does not copy dynamic HTML tags (that is, tags contained within COMPUTE fields) into the column heading. This means that in the resulting report, the detail lines would have an HTML tag which the column heading lines do not have. When those HTML tags are stripped from the detail lines (but not from the column heading lines) the column headings become skewed.

Thus, for our dynamic COLOR-TAG field we specified our own column heading containing an HTML comment tag. We made sure that the HTML comment in the column heading was the *same length* as the HTML tags appearing in the detail lines. (You can see this by looking at the output on page 318.) As a result, the same number of HTML bytes are removed from the column headings lines and from the detail lines. The resulting report remains aligned in your web browser.

If you look at the actual output on page 318, you will notice something else. Although we only specified one line of override column headings, Spectrum Writer used it for *all three* column heading lines. This is necessary to keep *all* column headings properly aligned. Spectrum Writer does this for you automatically (as long as either the HTML or HTMLAID option is specified).

We also specified an HTML comment as the column heading for the closing </FONT> literal. Without this override column heading, Spectrum Writer would have copied the </FONT> literal itself into the column headings. That would have caused an HTML error since there is no corresponding *opening* <FONT> tag in the column heading lines.

The default Grand Total line would also have been skewed for similar reasons. (It would not contain the dynamic <FONT COLOR=xxxxx> tag, and *would* contain an unmatched

These Control Statements:

```
OPTION: HTML('ABC COMPANY -- COLOR CODED SALES')
TITLE: '<CENTER><IMG SRC="LOGO.GIF"></CENTER>'
TITLE: '<CENTER><H2><I>SALES OVER $100 IN GREEN</I></H2></CENTER>'
INPUT:  SALES-FILE
COMPUTE: COLOR-TAG = WHEN(AMOUNT > 100) ASSIGN('<FONT COLOR=GREEN>')
                     ELSE               ASSIGN('<FONT COLOR=RED  >')
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER
         COLOR-TAG('<!--xxxxxxxxxxx-->') 0 AMOUNT TAX 0 '</FONT>'('<!---->')
BREAK: #GRAND NOTOTALS
       FOOTING('GRAND TOTALS' 40 AMOUNT(TOTAL) TAX(TOTAL))
```

Result in this Web Report:

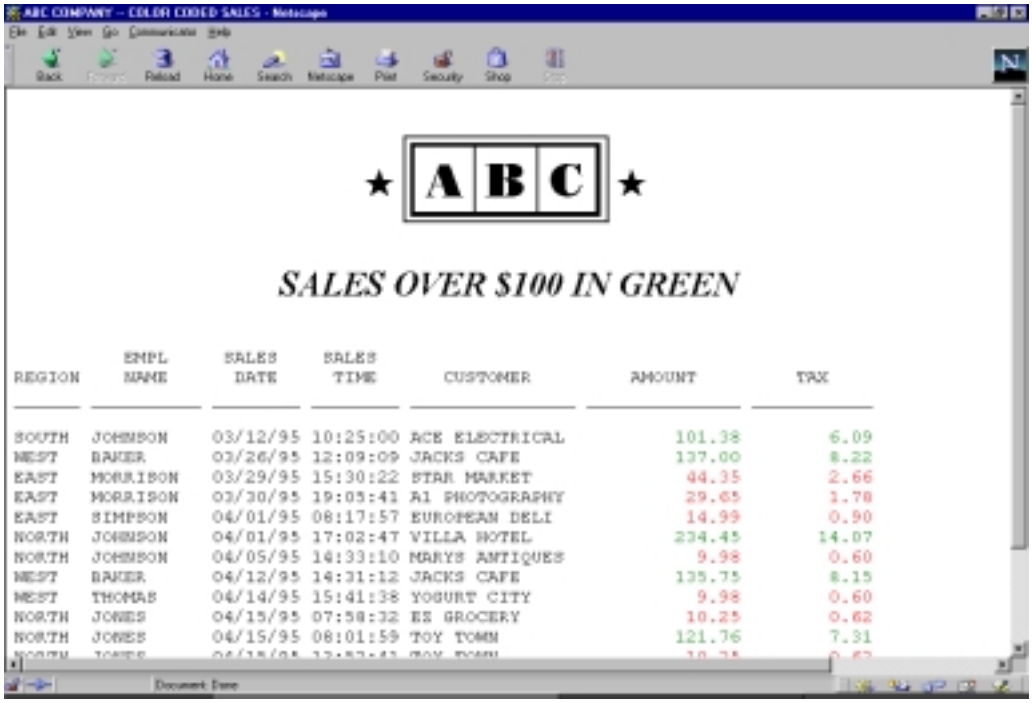


Figure 46. A Web report that uses dynamic HTML tags

```
<HTML>
<HEAD>
<TITLE>ABC COMPANY -- COLOR CODED SALES</TITLE>
</HEAD>
<BODY>
<PRE>
                              <CENTER><IMG SRC="ABCLOGO.GIF"></CENTER>
                        <CENTER><H2><I>SALES OVER $100 IN GREEN</I></H2></CENTER>

          EMPL     SALES    SALES                  <!--xxxxxxxxxxx-->                          <!---->
REGION    NAME      DATE     TIME      CUSTOMER     <!--xxxxxxxxxxx-->    AMOUNT         TAX    <!---->
______ __________ ________ ________ _____________ <!--xxxxxxxxxxx-->_____________ __________<!---->

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL  <FONT COLOR=GREEN>        101.38        6.09</FONT>
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE      <FONT COLOR=GREEN>        137.00        8.22</FONT>
EAST   MORRISON   03/29/95 15:30:22 STAR MARKET     <FONT COLOR=RED  >         44.35        2.66</FONT>
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY  <FONT COLOR=RED  >         29.65        1.78</FONT>
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI   <FONT COLOR=RED  >         14.99        0.90</FONT>
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL     <FONT COLOR=GREEN>        234.45       14.07</FONT>
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES  <FONT COLOR=RED  >          9.98        0.60</FONT>
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE      <FONT COLOR=GREEN>        135.75        8.15</FONT>
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY     <FONT COLOR=RED  >          9.98        0.60</FONT>
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY      <FONT COLOR=RED  >         10.25        0.62</FONT>
NORTH  JONES      04/15/95 08:01:59 TOY TOWN        <FONT COLOR=GREEN>        121.76        7.31</FONT>
NORTH  JONES      04/15/95 13:52:41 TOY TOWN        <FONT COLOR=RED  >         10.25        0.62</FONT>
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING   <FONT COLOR=GREEN>        500.00       30.00</FONT>
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER    <FONT COLOR=RED  >         23.87        1.43</FONT>


GRAND TOTALS                                                 1,383.66       83.05
</PRE>
</BODY>
</HTML>
```

Figure 47. HTML file with dynamic HTML tags

</FONT> tag.) Therefore, we used a BREAK statement to suppress the default Grand Totals. Then we used a FOOTING parm to specify precisely how the Grand Total line should look.

There is also another technique you can use to align the column headings in reports that have dynamic HTML tags. That is to suppress the default column headings by specifying the NOCOLHDGS option. Then use TITLE statements to specify your own column headings. The report in **Figure 39** (page 307) illustrates this.

Using the PRESCRIPT and POSTSCRIPT Options

The PRESCRIPT option is used to write one or more lines of text before the beginning of the report. The POSTSCRIPT option is used to write lines of text after the end of the report.

In **Figure 41** (page 310), we used these options to put hot link texts and HTML labels at the beginning and end of the report.

Another use of the PRESCRIPT and POSTSCRIPT options is to provide formatting tags that apply to your entire report. For example:

```
OPTION: HTML
OPTION: PRESCRIPT('<FONT COLOR=RED>')
OPTION: POSTSCRIPT('</FONT>')
```

The first statement above tells Spectrum Writer to begin your output file with the standard opening HTML tags. These tags will be followed by the <FONT COLOR=RED> tag specified in the PRESCRIPT option above. Then the actual report will follow. After the report, the closing </FONT> tag specified by the POSTSCRIPT option will appear, followed by Spectrum Writer's other standard closing HTML tags. When viewed on the Web, all text in the report (titles, column headings, data, Grand Totals) will be red.

The above example uses the PRESCRIPT and POSTSCRIPT options in addition to the HTML option. You can also use these options *instead* of the standard HTML option. This allows you to specify *all* of the HTML tags yourself. For example:

```
OPTION: HTMLAID
OPTION: PRESCRIPT('<HTML>')
OPTION: PRESCRIPT('<HEAD>')
OPTION: PRESCRIPT('<TITLE>ABC COMPANY SALES REPORT</TITLE>')
OPTION: PRESCRIPT('</HEAD>')
OPTION: PRESCRIPT('<BODY BACKGROUND="BACKLOGO.JPG">')
OPTION: PRESCRIPT('<PRE>')
*
OPTION: POSTSCRIPT('</PRE>')
OPTION: POSTSCRIPT('</BODY>')
OPTION: POSTSCRIPT('</HTML>')
```

The statements above show an alternative to using the HTML option. These statements result in a Web report similar to that produced by just using the HTML option. However, we did add one special option to the BODY tag. The BACKGROUND option names an image file on the Web site that contains a corporate logo graphic. That image will be repeated as necessary and used as the background for the Web report. This option can point to any image file on your site. For example, instead of a logo, you might want to use an image file that contains a "textured" background for your report.

Another option you can specify in the BODY tag is the BGCOLOR option. Use it to specify a solid background color for your Web report. For example:

```
OPTION: PRESCRIPT('<BODY BGCOLOR=PINK>')
```

Note: If you choose to write all of your own HTML tags (rather than use the HTML option), you should also specify the HTMLAID option (page 564). That option helps solve some potential alignment problems in reports that contain HTML tags.

Summary of Options for Web Reports

The following table summarizes some of the OPTIONS statement options that are useful when creating Web reports.

OPTIONS RELATED TO WEB REPORTS	
OPTION	**DESCRIPTION**
HTMLAID	Tells Spectrum Writer that you will be putting HTML tags within your report and that Spectrum Writer should recognize and support those tags. This option itself does not cause Spectrum Writer to add any HTML code to your report. See the OPTIONS statement syntax for a complete description of the HTMLAID option (page 564). Example: `OPTION: HTMLAID`

HTML[('title')]	Tells Spectrum Writer to wrap standard HTML code around the report. It also lets you specify an optional HTML title for the Web page. This option also turns on the HTMLAID option. Examples:
	`OPTION: HTML`
	`OPTION: HTML('ABC COMPANY SALES REPORT')`

NOCC	Tells Spectrum Writer not to begin each output line with a "carriage control" character. Such characters are only necessary when the output is being sent to a mainframe printer. You do not need to specify NOCC if you specify either the HTML or HTMLAID options, since these options automatically invoke the NOCC option. Example: `OPTION: NOCC`
NOCOLHDGS	Tells Spectrum Writer to not print column headings for the report. When the default column headings have alignment problems, specify this option and then use TITLE statements to specify your own column headings. Example: `OPTION: NOCOLHDGS`
NOUNDERSCORES	Tells Spectrum Writer not to underscore the column headings in the report. This is often desirable for reports that will be viewed online, since the underscore line uses up an additional line on the screen. Example: `OPTION: NOUNDERSCORES`

OPTIONS RELATED TO WEB REPORTS (CONTINUED)	
OPTION	**DESCRIPTION**
POSTSCRIPT('text')	Tells Spectrum Writer to write this text once, after the actual report. This option allows you to specify any closing HTML tags that you need. You can have as many POSTSCRIPT options as you want. If the HTML option is also specified, the POSTSCRIPT text(s) will appear *after* the report and immediately *before* the closing HTML tags created by the HTML option. Example: `OPTION: POSTSCRIPT(<'/FONT>')`
PRESCRIPT('text')	Tells Spectrum Writer to write this text once, before the actual report. This option allows you to specify any opening HTML tags that you need. You can have as many PRESCRIPT options as you want. If the HTML option is also specified, the PRESCRIPT text(s) will appear *before* the report and immediately *after* the initial HTML tags created by the HTML option. Example: `OPTION: PRESCRIPT('<FONT COLOR=BLUE>')`
TITLEONCE	Tells Spectrum Writer to write out the report titles (and column headings) only once — at the beginning of the report. This prevents titles from seeming to appear "randomly" throughout the report when a viewer pages through it online. Example: `OPTION: TITLEONCE`

Common HTML Tags

The following table lists some common HTML tags that you may want to use in your Web reports. Please note, however, that there are a number of different versions of HTML and it is constantly evolving as new versions of Web browsers are released. In addition, different browsers sometimes process the same HTML tag in slightly different ways. At the time of publication, the HTML tags shown below work as described. Of course, we cannot guarantee they will always work the same way in the future. Also, future versions of HTML will undoubtedly include many new features not documented here. You can always perform a web search to learn what features are available in the current version of HTML. At the time of this writing, one online resource for the current HTML specifications is at WWW.W3.ORG.

COMMON HTML TAGS	
HTML TAG	**DESCRIPTION**

<A HREF="url"> ... </A>	Specifies the beginning of a **hypertext link**. When the user clicks on the text or graphic within these tags, the action implied by the "url" will be performed.
	
	`TITLE: '<A HREF="WWW.COMPANY.COM/INDEX.HTML">' 0`
	`'CLICK HERE TO READ ABOUT THE ABC COMPANY</A>'`

<A NAME=label>	Specifies a **label** that can be referred to in a hypertext link. It requires no closing tag. `OPTION: PRESCRIPT('<A NAME=TOP>')`
<B> ... </B>	Makes the text within the tags **bold**. `TITLE:'SALES BY <B>REGION</B>'`
<BLINK> ... </BLINK>	Makes the text within the tags **blink on and off**. (This option may only work with Netscape browsers.) `TITLE: 'SALES BY <BLINK>REGION</BLINK>'`
** **	Inserts a **line "break"** in the report. It requires no closing tag. `COLUMNS: REGION CUSTOMER AMOUNT ' '`
<CENTER>...</CENTER>	**Centers** the text within the tags. `TITLE: '<CENTER>SALES BY REGION</CENTER>'`

<FONT
[SIZE=n]
[FACE="fontname"]
[COLOR=color]
...</FONT>
	This specifies information about the font to be used for the text within the tags. **Font sizes** between 1 (smallest) and 7 (largest) are currently supported. For example:
	
	`TITLE: 'SALES BY <FONT SIZE=5>REGION</FONT>'`
	
	The 16 **colors** currently supported by name in most browsers are: black, olive, teal, red, blue, maroon, navy, gray, lime, fuchsia, white, green, purple, silver, yellow and aqua. Example:
	
	`TITLE: 'SALES BY <FONT COLOR=GREEN>REGION</FONT>'`

COMMON HTML TAGS (CONTINUED)	
HTML TAG	**DESCRIPTION**
<FONT [SIZE=n] [FACE="fontname"] [COLOR=color] ...</FONT>	The face parm specifies the **name of the font** to be used. However, to preserve the column alignment of your report, in most cases you should only use *non-proportional* fonts. Another caution: the font you specify may not be available on your viewer's PC, and a substitute font may be substituted by the browser, causing unexpected results. Use this option with caution. Example: `TITLE: 'SALES BY <FONT FACE="HELVETICA">REGION</FONT>'`
<H1>...</H1> <H2>...</H2> ... <H6>...</H6>	This formats the text within the tags as a "level 1" (or 2, 3, etc.) **header**. `TITLE: '<H1>SALES BY REGION</H1>'`

<HR>	This produces a **"horizontal rule" (a line)** in your report. It requires no closing tag.
	`TITLE: 'ABC COMPANY SALES REPORT'`
	`TITLE: '<HR>'`

<I> ... </I>	This makes the text within the tags **italic**.
	`TITLE: 'SALES BY <I>REGION</I>`
	`'`

<IMG SRC="url" [ALIGN=TOP/ MIDDLE/BOTTOM] [WIDTH=nnn]>	Specifies that a **graphic image** should be placed here. The "url" must be the name of the graphic file, which will often be a .gif or a .jpg file. The optional ALIGN parm determines how any text that follows the image will be aligned with it. The optional WIDTH parm specifies the size that the image should take up in the display. This tag requires no closing tag. `TITLE: '<IMG SRC="LOGO.GIF">'`
<MARQUEE>...</MARQUEE>	Makes the text within the tag **scroll across the screen.** (This option may only work with Microsoft browsers.) `TITLE: 'SALES BY<MARQUEE>REGION</MARQUEE>'`
<SMALL> ... </SMALL>	Makes the text within the tags **smaller**. `TITLE: 'SALES BY <SMALL>REGION</SMALL>'`

COMMON HTML TAGS (CONTINUED)	
HTML TAG	**DESCRIPTION**
_{...}	Makes the text within the tags **subscripts** (smaller and lower than the regular baseline). `TITLE: 'SALES BY <SUB>REGION</SUB>'`
^{...}	Makes the text within the tags **superscripts** (smaller and higher than the regular baseline). `TITLE: 'SALES BY <SUP>REGION</SUP>'`
<TABLE **[BORDER=n]** **[CELLPADDING=n]** **[CELLSPACING=n]>** **...** **</TABLE>**	Specifies that the text and images within these tags should be formatted as a table. The optional BORDER parm determines whether there will be a border around the cells of the table. The optional CELLPADDING value tells how much space to leave between the border of a cell and its contents. The optional CELLSPACING value tells how much space to leave between the cells of the table. For example: `OPTION: PRESCRIPT('<TABLE BORDER=1 CELLPADDING=2>')`
<TD>	Specifies the beginning of a new table item ("data cell") within a table row. It requires no closing tag. For example: `COLUMNS: '<TR>' '<TD>' REGION '<TD>' EMPL-NAME`
<TR>	Specifies the beginning of a new table row (within a table). It requires no closing tag. For example: `COLUMNS: '<TR>' '<TD>' REGION '<TD>' EMPL-NAME`
<U> ... </U>	**Underlines** the text within the tags. `TITLE: 'SALES BY <U>REGION</U>'`
<!-- ... -->	This is an HTML comment. It has no effect on what the viewer sees on the Web browser. It can be used to document your HTML code. It can also be used in some situations to help align a Web report. Use it to strip excess bytes from a line of Spectrum Writer output. (See page 315 for an example of this.)

Chapter 6. How to Define Your Input Files

Chapter Table of Contents

Chapter 6. How to Define Your Input Files 325

How to Define a File 328
- How to Use the FILE Statement — OS/390 328
- How to Use the FILE Statement — VSE 331

How to Define a Field 333
- How to Define a Character Field 333
- How to Define a Numeric Field 335
- Should You Define a Field as Character or Numeric? 339
- How to Define a Date Field 340
- How to Define a Time Field 344
- How to Define a Bit Field 347
- How to Specify a Field's Column Heading 350
- How to Specify a Field's Location in a Record 350
- Variably Located Record Segments 353
- How to Define Arrays 355
- How to Specify What File a Field Belongs To 356
- How to Define a Field Created by a Data Exit 357

Keeping Your File Definitions in a Copy Library 360
- Including the Definition Statements "In–Line" 360
- Using the Spectrum Writer Copy Library 363
- How to Use a Copy Library Alias 367
- Defining One–Time Fields 368

Using Cobol and Assembler Record Layouts 369
- Live Runs Using Cobol Record Layouts 370
- Live Runs Using Assembler Record Layouts 372
- Handling Date and Time Fields in Record Layouts 375
- How Spectrum Writer Handles Cobol Arrays 377
- Converting Cobol and Assembler Layouts to FIELD Statements 378
- How to Copy Cobol and Assembler Record Layouts from Libraries 382
- Mixing FIELD Statements with COBOL and ASM Statements 383
- The Starting Column of a Cobol or Assembler Layout 384
- The "Default Location" After a Cobol or Assembler Layout 384
- The Scope of the COBOL and ASM Statements 384
- Technical Notes on Cobol Support 385
- Technical Notes on Assembler Support 387

Chapter 6. How to Define Your Input Files

This chapter is intended primarily for programmers "setting up" new files for Spectrum Writer. Users who simply request reports and PC files from input files that have already been set up do not need to read this chapter.

Spectrum Writer needs to know a few things about your company's files before it can use those files to produce reports. For example, it needs to know: whether a file is a VSAM file or not; the names of the *fields* present in the file; which column each field begins in, and so on.

There are two control statements that supply this information about your files to Spectrum Writer:

- the FILE statement, which gives information about the overall characteristics of a file
- the FIELD statement, which gives information about one individual field within a file

A Spectrum Writer file definition simply consists of a single FILE statement, followed by a number of FIELD statements. (Appendix F, "Files Used in Examples" on page 648 shows some sample file definitions.)

Defining a file is a *one–time* thing. You will write these definition statements once and then save them in Spectrum Writer's copy library. After that, you can produce as many different reports and PC files from the file as you like, without having to worry about these definition statements again.

For this reason a certain amount of care should be given to writing these definition statements. For example, a little time spent at this point in assigning useful **column headings** to each field may save you a lot of time in the future. If you specify a HEADING parm in your FIELD statement, you will not have to specify column headings in the COLUMNS statement of every report requested in the future. (Of course, if the field name itself makes a suitable column heading, then there's no need to specify a column heading at all.) Here is an example of specifying a column heading when defining a field:

```
FIELD: RECA-MSTR-EMPL-FIRST-NAME  LEN(20)  HEADING('FIRST NAME')
```

Another example is the use of the **NOACCUM parm**. When defining numeric fields that *should not be totalled* (such as employee numbers, cost center numbers, telephone numbers, social security numbers, etc.) specify the NOACCUM parm in the FIELD statement to prevent totalling. This keeps the user from having to specify it in each report requested later on. Here is an example of specifying NOACCUM when defining a field that should not be totalled:

```
FIELD: DEPT-NUM  TYPE(NUM)  LEN(1)  NOACCUM
```

Also, you should specify a FORMAT parm for any field that should normally be displayed in a special way. For example, a U.S. telephone number will normally be display with parentheses around the first three digits (the area code) and with a dash before the last 4 digits. If you specify such a PICTURE in the FIELD statement, users won't need to specify it

These Cobol Statements:

```
FILE-CONTROL.
    SELECT RECA-MSTR-FILE  ASSIGN TO UT-S-MSTRDD.
...
FD  RECA-MSTR-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS
    BLOCK CONTAINS 0 RECORDS.

01  RECA-MSTR-RECORD.
    05 RECA-MSTR-LAST-NAME             PIC X(20).
    05 RECA-MSTR-FIRST-NAME            PIC X(20).
    05 RECA-MSTR-JULIAN-BIRTH-DATE     PIC 9(5).
    05 RECA-MSTR-SALARY                PIC S9(7)V99 COMP-3.
    05 RECA-MSTR-DEPARTMENT-NUM        PIC 9.
    05 RECA-MSTR-HIRE-DATE.
       10 RECA-MSTR-HIRE-DATE-YY       PIC 99.
       10 RECA-MSTR-HIRE-DATE-MM       PIC 99.
       10 RECA-MSTR-HIRE-DATE-DD       PIC 99.
    05 RECA-MSTR-QUARTERLY-SALES-TABLE      OCCURS 4 TIMES.
       10 RECA-MSTR-SALES-QTR          PIC S9(5)V9(2)  COMP-3.
    05 RECA-MSTR-NUMBER-OF-SALES       PIC S9(4) COMP.
    05 FILLER                          PIC X(5).
```

Are Equivalent to these Spectrum Writer **Statements:**

```
FILE:  MSTR-FILE        DDNAME(MSTRDD)  LRECL(80)

FIELD: LAST-NAME        LENGTH(20)
FIELD: FIRST-NAME       LENGTH(20)
FIELD: BIRTH-DATE                   TYPE(YYDDD)
FIELD: SALARY           LENGTH(5)   TYPE(COMP-3)   DECIMAL(2)
FIELD: DEPARTMENT-NUM   LENGTH(1)   TYPE(NUM)      NOACCUM
FIELD: HIRE-DATE                    TYPE(YYMMDD)
FIELD: HIRE-DATE-YY     LENGTH(2)   TYPE(NUM)      COLUMN(* - 6)
FIELD: HIRE-DATE-MM     LENGTH(2)   TYPE(NUM)
FIELD: HIRE-DATE-DD     LENGTH(2)   TYPE(NUM)
FIELD: SALES-QTR-1      LENGTH(4)   TYPE(COMP-3)   DECIMAL(2)
FIELD: SALES-QTR-2      LENGTH(4)   TYPE(COMP-3)   DECIMAL(2)
FIELD: SALES-QTR-3      LENGTH(4)   TYPE(COMP-3)   DECIMAL(2)
FIELD: SALES-QTR-4      LENGTH(4)   TYPE(COMP-3)   DECIMAL(2)
FIELD: NUMBER-OF-SALES  LENGTH(2)   TYPE(COMP)
```

Remarks:

- the FILE statement for Spectrum Writer VSE would be:
  `FILE: MSTR-FILE ATTR(DASD,'MSTRDD',80,160)`
- the common prefix (RECA–MSTR) was dropped to make the field names more user friendly
- for numeric fields, Spectrum Writer always requires the length (in bytes) that a field occupies in the input record — not necessarily the number of digits it contains
- the DECIMAL parm specifies the number of decimal digits in a field
- the COLUMN(* – 6) parm for HIRE–DATE–YY is used to "back up 6 bytes" to redefine the HIRE–DATE field
- the OCCURS table in the Cobol layout is defined as four individual fields for Spectrum Writer

Figure 48. A sample Spectrum Writer file definition

in COLUMNS statements later on. You may also want to specify the NOCOMMA format for numeric fields that should not be displayed with commas (such as cost centers, subscription numbers, etc.) Here are some examples of specifying a display format when defining fields:

```
FIELD: TELEPHONE    TYPE(NUM)  LEN(10)  FORMAT(PIC'(999) 999-9999')
FIELD: COST-CENTER  TYPE(NUM)  LEN(7)   FORMAT(NOCOMMA)  NOACCUM
```

The remainder of this chapter is divided into four sections.

- the first section explains how to use the FILE statement to define the overall characteristics of a file (below)
- the second section explains how to use FIELD statements to define the individual fields within the file (page 333)
- the third section describes how to store these statements in a Spectrum Writer Copy Library, to make requesting reports easy (page 360)
- the fourth section shows how to use Cobol or Assembler record layouts to define your files to Spectrum Writer. You can use such record layouts in place of a Spectrum Writer file definition. Or, you can use the record layouts to create a standard Spectrum Writer file definition (page 369).

Sometimes a picture is worth a thousand words. So, before we get into the details of how to define files, notice **Figure 48** (page 327). It gives you an idea of how a typical Cobol definition might be defined to Spectrum Writer.

How to Define a File

This section explains:

- how to use the FILE statement to **define a file** to Spectrum Writer
- how you can later **override aspects of a file definition** in the INPUT or READ statement

Input files are defined to Spectrum Writer with the FILE statement. The parms used in the FILE statement differ between Spectrum Writer OS/390 and Spectrum Writer VSE. Please refer to the correct section for your operating system:

- for OS/390, see below
- for VSE, see page 331

How to Use the FILE Statement –– OS/390

There are a number of parms that can be used in a FILE statement to provide information about a file. (The complete syntax of the FILE statement is found beginning on page 531.) Only a few of these parms are actually *required*. The others are optional, and are only needed in unusual cases.

The four things that Spectrum Writer must know about a file are:

- the **filename** (that is, the "user friendly" name by which it will be referred to in other Spectrum Writer control statements)
- the **TYPE** of file (that is, the *access method* to be used when reading the file)
- the **LRECL** of the file (that is, the size of the largest record that Spectrum Writer could encounter when reading the file)
- the **DDNAME** that identifies the file in the *job control language* (JCL)

The first item in a FILE statement is always the **filename**. For example:

```
FILE: SALES-FILE
```

The above statement defines a file named SALES–FILE. You may choose any name you like for a file (within the rules governing file names given on page 446). This is the name that will be used in Spectrum Writer control statements when referring to this file. It does *not* have to be the actual DSNAME ("data set name") or DDNAME of the file.

After the filename parm, the other parm(s) may appear in any order in the FILE statement.

Use the **TYPE parm** to tell Spectrum Writer what type of file is being defined. This tells Spectrum Writer which access method to use when performing I/O to the file. Spectrum Writer supports two types of files:

- SEQUENTIAL (or just SEQ)
- VSAM

If the TYPE parm is not specified, the default file type is SEQUENTIAL. The FILE statement shown above did not specify a file type, so the SALES–FILE is assumed to be sequential. Spectrum Writer uses SAM/QSAM I/O with sequential files. The "sequential" file type covers most non–VSAM files. Sequential files include:

- "flat" disk files, such as those maintained with TSO editors
- members of partitioned data sets (PDS)
- most files stored on magnetic tapes

The second type of file supported by Spectrum Writer is a VSAM file:

```
FILE: EMPL-FILE  TYPE(VSAM)
```

The above statement defines a file named EMPL–FILE as being a VSAM file. Spectrum Writer supports KSDS, ESDS and RRDS VSAM files.

> **Note:** you can also use other types of files with Spectrum Writer. However, you will need to write an I/O Exit program in order to do that. I/O Exits are discussed in Appendix I, "I/O Exits" (page 673).

Use the **DDNAME parm** to supply the name of a DD statement that will be present in the execution JCL. This DD statement will contain the actual DSNAME (data set name) of the file. Spectrum Writer uses the DDNAME in order to "open" an input file and read from it. For example:

```
FILE: SALES-FILE  DDNAME(SALESDD)
```

The above statement defines a file named SALES–FILE. When Spectrum Writer needs this file to produce a report, it will open and read the dataset named in the SALESDD DD statement in the JCL.

Use the **LRECL** (logical record length) parm to specify the size of the largest record that the file will possibly contain. For example:

```
FILE: SALES-FILE  DDNAME(SALESDD)  LRECL(5000)
```

The above statement specifies that a record as large as 5000 bytes may be encountered in the SALES–FILE. This statement tells Spectrum Writer to provide a 5000–byte I/O area to use when reading records from this file. If no LRECL parm is present, Spectrum Writer reserves a 1000 byte I/O area as a default.

Note: When defining variable length (VB) SEQ files, the LRECL parm should include the length of the 4–byte "record descriptor word" (RDW) at the beginning of each record.

Note: It is not a problem to specify a larger LRECL value than is actually needed. In fact, if you suspect that a file's LRECL may grow in the future, you may want to specify a larger LRECL with some "growth" room in it. On the other hand, specifying an excessively large LRECL may result in higher CPU usage in certain circumstances.

The **NORMALIZE parm** can be useful if the file you are defining contains an array. (See "Using Normalization to Process Arrays" on page 237 for detailed information.) For example, the EMPL-FILE has an array of four quarterly sales amounts (page 649). If we wished to normalize that array whenever the EMPL-FILE was used as an input file, we would add a NORMALIZE parm:

```
FILE: EMPL-FILE TYE(VSAM) NORMALIZE(SALES-QTR1, 4)
```

Other optional parms that can be specified in the FILE statement are:

- EXITPARM (for use with file fields created in user-written data exits — see page 534)
- IOEXIT (for files which will be accessed from a user-written I/O Exit — see page 534 and page 673)
- KEEPRDW (use with VB files if you want "column 1" of the record to point to the "record descriptor word" (RDW), rather than to the data after the RDW. See page 352 and page 535.)
- NORMWHEN (for normalizing only certain types of records — see page 536 and page 247)
- STOPWHEN (if you always want to process a file only up to a certain point — see page 536)

How to Override a File Definition

Remember that the FILE statement simply *defines* a file to Spectrum Writer for later use. It *does not* make that file an input file to a report. The INPUT and READ statements request that a file be used as input for a report. When an INPUT or READ statement specifies a particular file, Spectrum Writer will know all about that file from the FILE statement processed earlier.

Sometimes you may want to change one or more aspects of the file definition for a certain run. You can do this by specifying one or more file definition parms directly in the INPUT or READ statement. These parms will override any such parm specified in the FILE statement — but only for the current run. All of the file definition parms can be specified in the INPUT and READ statements.

For example, to override the DDNAME used for a particular run, you could specify:

```
INPUT: SALES-FILE DDNAME(TEMPSALE)
```

How to Use the FILE Statement –– VSE

The FILE statement's ATTR parm is used to describe the attributes of a VSE file to Spectrum Writer. Here is an example of an ATTR parm in a FILE statement:

```
FILE: SALES-FILE  ATTR(DASD,'SALEFIL',80,160)
```

The statement above defines a file called SALES–FILE. It has the following attributes:

- it is a SAM file on DASD. (Other possibilities are SAM files on TAPE, and VSAM files)
- the DLBL name used for this file in the JCL is SALEFIL
- the records in this file are 80 bytes long, and the blocks are 160 bytes long

Note: The complete syntax of the **ATTR** parm is shown on page 532.

Here is another example of defining a VSE file with the ATTR parm. In this example, we define a VSAM file to Spectrum Writer:

```
FILE: EMPL-FILE  ATTR(VSAM,'EMPFILE',150)
```

The EMPL–FILE defined above is a **VSAM** file. The DLBL name used in the JCL is EMPFILE. The records in the file may be up to 150 bytes long. No block size is used with VSAM files.

Note: Use VSAM only for true VSAM ESDS, KSDS or RRDS datasets. DASD should be used for all SAM files on disk — even SAM files that are in VSAM-*managed* space.

Here is an example of defining a file with variable-length blocked records:

```
FILE: VAR-FILE  ATTR(TAPE,'FILEIN',SYS009,V,100,5000)
```

The file defined above is a SAM file on **TAPE**. The TLBL name used in the JCL is FILEIN. The tape will be mounted on the tape drive at logical unit SYS009. The records are variable length. The largest record that the file might contain is 100 bytes long. The longest block that the file might contain is 5000 bytes long.

Note: When defining variable length SAM files, the record size should include the length of the 4-byte "record descriptor word" (RDW) at the beginning of each record. Likewise, the block size should include the 4-byte block prefix.

The **NORMALIZE parm** can be useful if the file you are defining contains an array. (See "Using Normalization to Process Arrays" on page 237 for detailed information.) For example, the EMPL-FILE has an array of four quarterly sales amounts (page 649). If we

wished to normalize that array whenever the EMPL-FILE was used as an input file, we would add a NORMALIZE parm:

```
FILE: EMPL-FILE ATTR(VSAM,'EMPFILE',150) NORMALIZE(SALES-QTR1, 4)
```

Other optional parms that can be specified in the FILE statement are:

- EXITPARM (for use with file fields created in user-written data exits — see page 534)
- IOEXIT (for files which will be accessed from a user-written I/O Exit — see page 534 and page 673)
- KEEPRDW (use with VB files if you want "column 1" of the record to point to the "record descriptor word" (RDW), rather than to the data after the RDW. See page 352 and page 535.)
- NORMWHEN (for normalizing only certain types of records — see page 536 and page 247)
- STOPWHEN (if you always want to process a file only up to a certain point — see page 536)

How to Override a File Definition

Remember that the FILE statement simply *defines* a file to Spectrum Writer for later use. It *does not* make that file an input file to a report. The INPUT and READ statements request that a file be used as input for a report. When an INPUT or READ statement specifies a particular file, Spectrum Writer will know all about that file from the FILE statement processed earlier.

The ATTR parm can also be used directly in the INPUT and READ statements. This temporarily changes the way a file is defined for a single Spectrum Writer run.

If an INPUT or READ statement contains an ATTR parm, the information from that ATTR parm overrides the information from the ATTR parm in the FILE statement. Also, you may omit the ATTR parm in the FILE statement altogether, as long as you specify it each time in the INPUT or READ statement.

For example, assume that for a single run we wanted to use a tape backup copy of the SALES–FILE defined above (instead of the copy on disk). Rather than changing the FILE statement, we could just use an ATTR parm in our INPUT statement, like this:

```
INPUT: SALES-FILE  ATTR(TAPE,'SALEFIL',SYS004,80,160)
```

The statement above changes the attributes of the SALES–FILE (for the current run only) to the following:

- the file is on tape
- the TLBL name for this file in the JCL is SALEFIL
- the tape will be mounted on the tape drive at logical unit SYS004
- the records in the file are 80 bytes long, and the blocks are 160 bytes long

Note that even though the record size and block size did not change from their values in the FILE statement, we had to specify them in this ATTR parm. If you specify an ATTR parm in an INPUT or READ statement, you must specify *all of the required items* in that parm. None of the ATTR information from the FILE statement is retained.

How to Define a Field

This section explains:

- how to use the FIELD statement to **define individual fields** to Spectrum Writer

There are five general types of fields used in Spectrum Writer:

- character
- numeric
- date
- time
- bit

Each type of field is defined somewhat differently. For example, the following statement defines a *character field*:

```
FIELD: LAST-NAME  LENGTH(15)
```

The FIELD statement necessary to define a *numeric* field that is stored in packed format and which includes two decimal digits is a little longer:

```
FIELD: TOTAL-SALES  LENGTH(7)  TYPE(PACKED)  DECIMAL(2)
```

In the sections that follow we discuss how to define each type of field. The complete syntax of the FIELD statement is given on page 522.

> **Note:** Spectrum Writer OS/390 and Spectrum Writer VSE both use exactly the same FIELD statements.

How to Define a Character Field

This section explains:

- what a **character** field is
- which parms are **required** to define a character field
- which **optional** parms can be used when defining character fields

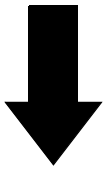
Most of the examples used in this section are illustrated in the sample report in **Figure 49**.

Character fields can contain any combination of letters, numerals, spaces, punctuation marks, and other special characters. Character fields contain such things as names, addresses, descriptions, etc.

> **Note:** Fields defined as character fields *cannot* be used in arithmetic comparisons or calculations — even if the field contains only numeric characters. If you wish to treat such fields as numeric data, define them as numeric rather than character fields. See page 339 for more on this subject.

These Control Statements:

```
FILE:  EMPL-FILE  DDNAME(EMPLFILE)  TYPE(VSAM)
FIELD: LAST-NAME        COLUMN(4)  LENGTH(15)
FIELD: FIRST-NAME                  LENGTH(15)
FIELD: STATUS-BYTE      COLUMN(42) LENGTH(1)
FIELD: HEX-STATUS-BYTE  COLUMN(42) LENGTH(1)  FORMAT(HEX)
                        HEADING('EMPLOYEE|STATUS BYTE')

INPUT:    EMPL-FILE
TITLE:    'EXAMPLES OF DEFINING CHARACTER FIELDS'
SORT:     LAST-NAME  FIRST-NAME
COLUMNS:  LAST-NAME  FIRST-NAME  STATUS-BYTE
          HEX-STATUS-BYTE
```



Produce this Report:

```
       EXAMPLES OF DEFINING CHARACTER FIELDS

     LAST            FIRST      STATUS  EMPLOYEE
     NAME            NAME        BYTE  STATUS BYTE

BAKER           VIVIAN            A        C1
CHRISTOPHERSON  MELISSA           A        C1
JOHNSON         LINDA             A        C1
JOHNSON         THOMAS            A        C1
JONES           JERRY             A        C1
MACDONALD       RICHARD                    40
MORRISON        MICHAEL           A        C1
SIMPSON         TIMOTHY           A        C1
THOMAS          MARTIN            A        C1


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- a COLUMN parm was used in the first FIELD statement, since the LAST–NAME field does not begin in the first column of the record
- no COLUMN parm was required for FIRST–NAME, since that field begins immediately after the previously defined field
- the HEX–STATUS–BYTE field occupies the same byte in the record as the STATUS–BYTE field. It simply has a different default display format.
- the HEADING parm specifies the column heading to use when the HEX–STATUS–BYTE field appears as a report column — the other columns have the field names themselves for column headings

Figure 49. A report with FIELD statements that define character fields

Character fields are the easiest kind of field to define. When no TYPE parm is supplied in a FIELD statement, a character field is assumed. Therefore, the only parms *required* to define a character field are:

- fieldname
- LENGTH

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 448.

After the fieldname, the other parm(s) may appear in any order in the FIELD statement.

The **LENGTH parm** is required to tell Spectrum Writer how many bytes (or "characters") the field occupies in the record. For example:

```
FIELD: LAST-NAME  LENGTH(15)
```

The above example defines a field named LAST–NAME that occupies 15 bytes of the input record. It is a character field by default, since no TYPE parm was specified. If you wish to include the **TYPE parm** for clarity or consistency, you can do so like this:

```
FIELD: LAST-NAME  LENGTH(15)  TYPE(CHAR)
```

Spectrum Writer assumes that the LAST–NAME field occupies the 15 bytes immediately after the previously defined field. If you want to explicitly specify where the 15–byte field is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed on page 350. As an example, if the LAST–NAME field begins in the fourth byte of the record, we could define it like this:

```
FIELD: LAST-NAME  LENGTH(15)  COLUMN(4)
```

The **FORMAT parm** of the FIELD statement specifies the default display format to use when displaying a field in a report. The FORMAT parm is not normally used when defining character fields. One instance when you might want to use the FORMAT parm is when you have a character field that you normally want to display in its hexadecimal representation. (A status byte or flag byte are examples of such fields.) You can specify a display format of HEX when defining such a field. The following statement defines a 1–byte character field named STATUS–BYTE and specifies that, by default, it should be displayed in hexadecimal notation when it appears in a report.

```
FIELD: STATUS-BYTE  LENGTH(1)  FORMAT(HEX)
```

How to Define a Numeric Field

This section explains:

- what a **numeric** field is
- which parms are **required** to define a numeric field
- which **optional** parms can be used when defining numeric fields

Most of the examples used in this section are illustrated in the sample report in **Figure 50**.

Numeric fields contain numeric values. Examples of numeric fields are costs, salaries, sales volumes, interest rates, etc. There are a number of different ways that a numeric field

These Control Statements:

```
FILE:  EMPL-FILE  DDNAME(EMPLFILE)  TYPE(VSAM)

FIELD: LAST-NAME      COL(4)  LEN(15)

FIELD: DEPT-NUM       COL(40) LEN(1) TYPE(NUM) NOACCUM

FIELD: TOTAL-SALES    COL(56) LEN(7) TYPE(NUM) DEC(2)
                              HEADING('YEARLY SALES TOTAL')

FIELD: DOLLAR-SALES   COL(56) LEN(7) TYPE(NUM) DEC(2)
                              FORMAT(PIC'$$$,$$$,$$$')

FIELD: TELEPHONE      COL(153) LEN(10) TYPE(NUM)
                               FORMAT(PIC'(999) 999-9999')

INPUT:    EMPL-FILE
TITLE:    'EXAMPLES OF DEFINING NUMERIC FIELDS'
COLUMNS:  LAST-NAME  TELEPHONE  TOTAL-SALES
          DOLLAR-SALES  DEPT-NUM
```

Produce this Report:

```
                EXAMPLES OF DEFINING NUMERIC FIELDS

     LAST                                              DOLLAR     DEPT
     NAME        TELEPHONE     YEARLY SALES TOTAL      SALES      NUM

BAKER           (415) 555-1209          92,125.89     $92,126       4
CHRISTOPHERSON  (602) 555-4556          47,665.31     $47,665       1
JOHNSON         (415) 555-6785          75,023.55     $75,024       2
JOHNSON         (602) 555-6654          86,999.24     $86,999       1
JONES           (415) 555-7653          42,509.89     $42,510       2
MACDONALD       (415) 555-9887           2,560.98      $2,561       2
MORRISON        (818) 555-4748          98,054.99     $98,055       3
SIMPSON         (818) 555-1887           8,723.88      $8,724       3
THOMAS          (415) 555-1152          60,193.49     $60,193       4


*** GRAND TOTAL (9 ITEMS)              513,857.22    $513,857
```

Remarks:

- we used abbreviations for the COLUMNS, LENGTH and DECIMAL parms. See page 522 for a list of abbreviations allowed in the FIELD statement
- the NOACCUM parm prevents the DEPT–NUM column from being totalled
- the PICTURE in the FORMAT parm causes DOLLAR–SALES to be displayed with a leading dollar sign, and with no decimal digits
- the use of special characters (namely, the parentheses) in the PICTURE for TELEPHONE keeps that column from being totalled

Figure 50. A report with FIELD statements that define numeric fields

can be stored in a record. It can be stored as character–type digits, as packed data, or as binary data, to name a few possibilities. The FIELD statement's TYPE parm tells Spectrum Writer exactly how the number is stored in the record.

> **Note:** Once a numeric field has been defined, you do *not* need to remember how it is stored in the record. You may freely compare *any kind* of numeric field (packed, binary, etc.) with any other numeric field. Spectrum Writer automatically takes care of any required conversion. You may also mix any combination of numeric fields when performing arithmetic computations.

The only parms *required* to define a numeric field are:

- fieldname
- TYPE
- LENGTH

The following *optional* parms also relate specifically to numeric fields:

- DECIMAL
- ACCUM/NOACCUM

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 446.

After the fieldname, the other parm(s) may appear in any order in the FIELD statement.

When defining a numeric field to Spectrum Writer, the **TYPE parm** is required. This parm indicates the exact way in which the numeric data is stored in the record. There are several ways that are commonly used to store numeric values in a record. Spectrum Writer needs to know which method is used for a particular field in order to process it correctly. A complete list of numeric data types appears on page 610. Here is an example of defining a numeric field:

```
FIELD: TOTAL-SALES  TYPE(NUM)  LENGTH(7)
```

The above statement defines a numeric field named TOTAL–SALES. Its data is stored in the record in "display numeric" format (that is, using numeric digits in character format). Spectrum Writer's NUM data type is equivalent to Cobol's USAGE DISPLAY. Other common numeric data types are:

- PACKED or COMP–3, which correspond to Cobol's COMP–3, and
- BINARY or COMP, which correspond to Cobol's COMP

The **LENGTH parm** is required to tell Spectrum Writer how many bytes the field occupies in the record. (Note that for some types of numeric data the LENGTH parm is not necessarily the same as the number of *digits*.)

> **Note:** To determine how many bytes a PACKED (COMP–3) field occupies in a record, use this formula: add 1 to the total number of *digits*; then divide this sum by 2, throwing away any remainder. The result is the number of *bytes* the field occupies in the record.
>
> As an example, take the RECA–MSTR–SALARY field (in **Figure 48** on page 327). It has a total of 9 digits (seven before the decimal point and two after). Adding 1 to this gives us 10. Dividing 10 by 2 gives us its length — 5 bytes.

Fields stored as BINARY data (COMP) are usually either 2 or 4 bytes long. If the BINARY field contains no more than 4 digits, it is usually 2 bytes long. If the field has more than 4 digits, it is generally 4 bytes long.

Spectrum Writer assumes that the TOTAL–SALES field defined in the previous example occupies the 7 bytes immediately after the previously defined field. If you want to explicitly specify where the 7–byte field is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed on page 350. As an example, if the TOTAL–SALES field began in the 56th byte of a record, we could define it like this:

```
FIELD: TOTAL-SALES  TYPE(NUM)  LENGTH(7)  COLUMN(56)
```

Since no **DECIMAL parm** was specified in the preceding examples, Spectrum Writer would assume that the TOTAL–SALES field contained no decimal digits. If a numeric field does contain one or more decimal digits, use the DECIMAL parm to indicate that. For example, if the data for TOTAL–SALES includes two decimal digits, we would use the following statement to define the field:

```
FIELD: TOTAL-SALES  TYPE(NUM)  LENGTH(7)  DECIMAL(2)
```

The DECIMAL parm above tells Spectrum Writer that the last two digits in the field are to be considered decimal digits. The DECIMAL parm may be used with any numeric field, regardless of which TYPE parm is used.

The **ACCUM** and **NOACCUM parms** can also be used when defining numeric fields. They specify whether or not to *accumulate* the field when it appears as a column in a report. Fields which are accumulated receive Grand Totals at the end of the report and subtotals at control breaks. Accumulated fields also appear in any other statistical lines that appear in a report (such as average lines, maximum lines, etc.)

By default, all numeric fields (except those displayed with certain non–numeric PICTUREs) are accumulated. Some numeric fields, such as a telephone number, a department number, or an employee number, *should not* be totalled. Use the NOACCUM parm to prevent these kinds of numeric fields from appearing in the total lines. For example:

```
FIELD: DEPT-NUM  LENGTH(1)  TYPE(NUM)  NOACCUM
```

The above statement specifies that the DEPT–NUM field should not be accumulated when it appears as a column in a report. Therefore, the DEPT–NUM column will not be totalled at control breaks and at the end of the report, even though it is defined as a numeric field. For a more detailed discussion about which fields appear in the total lines, see page 148.

Another parm you may want to use when defining numeric fields is the **FORMAT parm**. By default, all numeric fields (regardless of their TYPE) are *displayed* with the NUMERIC display format. The NUMERIC display format: suppresses leading zeros; uses commas to separate groups of three digits; and adds a leading minus sign for negative values. If you want a numeric field to have a different default display format, use the FORMAT parm. For example:

```
FIELD: COST-CENTER  TYPE(NUM)  FORMAT(NOCOMMA)  NOACCUM
```

The above statement specifies that whenever the COST–CENTER field is displayed in a report, the NOCOMMA format should be used. The NOCOMMA format does *not* use commas to separate groups of digits. When displaying fields like cost centers, employee numbers, account numbers, etc., you normally do not want them formatted with commas. (You also

do not want them totalled, which is why we also specified NOACCUM in the above statement.)

A complete list of numeric display formats is found on page 619.

The PICTURE display format gives you great flexibility in describing how a numeric field should be formatted. For example:

```
FIELD: DOLLAR-SALES  LENGTH(7)  TYPE(PACKED)  DECIMAL(2)
       FORMAT(PIC'$$$,$$$,$$$')
```

The above statement uses a PICTURE to specify the display format of the DOLLAR–SALES field. In this example, a total of 11 positions (the size of the PICTURE text) will be reserved for displaying the field. (That will accommodate a value with as many as nine numeric digits.) A floating dollar sign will precede the first non–zero digit in the amount. No decimal digits will be displayed. (The two decimal digits contained in the raw data will be rounded out when the field is formatted for the report.)

Here is another example of using a PICTURE in the FORMAT parm to customize the way a numeric field is displayed:

```
FIELD: TELEPHONE  LENGTH(10)  TYPE(NUM)
       FORMAT(PIC'(999) 999-9999')
```

This example uses parentheses and a dash as part of the PICTURE in order to display the TELEPHONE field's 10 digits in the standard format:

```
(415) 555-1212
```

The section titled "PICTURE Display Formats" (page 451) explains the rules for writing PICTUREs.

> **Note:** The FORMAT parm specifies the *default* display format that will be used for a field. Override display formats can always be used in other control statements to change the way the field is displayed in a particular report.

Should You Define a Field as Character or Numeric?

This section explains:

- how to decide whether a field that contains only numeric digits should be defined as a character field or as a numeric field

Most files have some fields that contain only numeric digits, stored in "display numeric" format. When defining these fields you must decide whether you want to define them as *character* or *numeric* fields.

It is better to define certain types of fields as **character fields**, even though they contain only numeric digits. Examples of such fields are: employee numbers, department numbers, and product code numbers. If such fields were defined as numeric, they would be *formatted as numbers* (by default), with commas inserted among the digits. They would also be *totalled* (by default) at the end of the report. They would appear in any statistical lines printed in the report. This kind of processing is not normally wanted for such things as employee numbers and department numbers. To avoid this, define the fields as *character* fields rather than as *numeric* fields. Character fields are always displayed just as they are

(no commas are inserted) and they are never totalled. Remember to use **character literals** (in quotation marks) when working with fields defined as character:

```
INCLUDEIF: EMPL-NUM = '037'
```

On the other hand, there is one advantage to defining certain of these fields as **numeric fields**. You can use a PICTURE to specify special display formats for numeric fields. Some examples of fields that you might want to use a PICTURE with are telephone numbers and social security numbers. For example, you might want to use a PICTURE such as PIC'(999) 999–9999' to format a telephone number in a report. Or, you way want to format a social security number using PIC'999–99–9999'. If you want to use a PICTURE to specify a customized display format, you must define the field as numeric. (PICTUREs are not allowed for character fields.) Remember to use **numeric literals** (no quotation marks) when working with fields defined as numeric:

```
INCLUDEIF: TELEPHONE = 4155557653
```

Once you have decided how to define a field, you can still "change your mind."

If you find that you need to treat a character field as a number, you can convert it to a numeric value by using the #MAKENUM built–in function in a COMPUTE statement. (See page 636.) For example, if EMPL–NUM has been defined as a character field, and you want to add 900 to it, you could do that by first converting it to a numeric value:

```
COMPUTE: NEW-EMPL-NUM = #MAKENUM(EMPL-NUM) + 900
```

The result field (NEW–EMPL–NUM) will be *numeric*, since the computational expression was numeric. (It involved the addition of two numeric operands.) You would use numeric literals (no quotes) when working with this field:

```
INCLUDEIF: NEW-EMPL-NUM = 937
```

If you find the need to treat a numeric field as a character field, you can convert it to a character value using the #FORMAT built–in function. (See page 632.) Assume that TELEPHONE has been defined as a numeric field. You can make a character field that contains the formatted telephone number by using the following statement:

```
COMPUTE: CHAR-TELEPHONE = #FORMAT(TELEPHONE, PIC'(999) 999-9999')
```

The result field (CHAR–TELEPHONE) will be a 14–byte character field (the size of the PICTURE). You would use character literals (with quotes) when working with this field:

```
INCLUDEIF: CHAR-TELEPHONE = '(415) 555-7653'
```

You could also extract certain digits out of this telephone number now that it is character data:

```
COMPUTE: AREA-CODE = #SUBSTR(CHAR-TELEPHONE,2,3)
```

How to Define a Date Field

This section explains:

- what a **date** field is
- which parms are **required** to define a date field

- which **optional** parms can be used when defining date fields

Most of the examples used in this section are illustrated in the sample report in **Figure 51**.

Date fields contain calendar dates. Examples of date fields are birth dates, hire dates, expiration dates, sales dates, etc. There are a number of different ways that a date field can be stored in a record. It can be stored as a 6–byte character YYMMDD date, as a packed Julian date, or as a 3–byte hexadecimal MMDDYY field (to name just a few possibilities). The FIELD statement's TYPE parm tells Spectrum Writer exactly how the date is stored in the record.

> **Note:** Once a date field has been defined, you do not need to remember how it is stored in the record. You may freely compare *any kind* of date field with any other date field. Spectrum Writer automatically takes care of any required conversion.

The only parms *required* to define a date field are:

- fieldname
- TYPE

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 446.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

When defining a date field to Spectrum Writer the **TYPE parm** is required. This parm indicates the exact way in which the date is stored in the record. There are a number of ways that are commonly used to store dates in a record. Spectrum Writer needs to know which method is used for a particular field in order to process it correctly. A complete list of date data types appears on page 611. Here are two examples:

```
FIELD: HIRE-DATE   TYPE(YYMMDD)
FIELD: BIRTH-DATE  TYPE(H-MMDDYY)
```

The first example above defines a field named HIRE–DATE that contains a date in character YYMMDD format (for example, `"951231"` for December 31, 1995). This type of date field takes up 6 bytes in the record. The second statement specifies that the BIRTH–DATE field is stored in hexadecimal MMDDYY format (for example, `X'123195'` for the same date). This type of date requires only 3 bytes in the record.

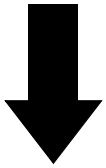
The **LENGTH parm** is generally not required for date fields. Depending on the particular data type, Spectrum Writer assumes a default length for each date field. For example, the length of a date field in YYMMDD form is 6 bytes. The length of a date field in H–MMDDYY form is 3 bytes, and so on. The default length and the allowable lengths for each date data type are shown in the table on page 611. If Spectrum Writer's default length is correct, you do not need to specify the LENGTH parm (although you may do so). However, if your field size is different than the default, you must specify its actual length using the LENGTH parm.

Spectrum Writer assumes that the HIRE–DATE field defined in the preceding example occupies the six bytes immediately after the previously defined field. If you want to explicitly specify where the 6–byte field is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed on page 350. For example, if HIRE–DATE begins in the 34th column of a record, we could define it like this:

```
FIELD: HIRE-DATE  TYPE(YYMMDD)  COLUMN(34)
```

These Control Statements:

```
FILE:   EMPL-FILE   DDNAME(EMPLFILE)   TYPE(VSAM)

FIELD:  LAST-NAME       COLUMN(4)   LENGTH(15)
FIELD:  HIRE-DATE       COLUMN(34)  TYPE(YYMMDD)
FIELD:  LONG-HIRE-DATE  COLUMN(34)  TYPE(YYMMDD)
                                    FORMAT(LONG1)
                                    HEADING('DATE HIRED')

INPUT:   EMPL-FILE
TITLE:   'EXAMPLES OF DEFINING DATE FIELDS'
COLUMNS: LAST-NAME    HIRE-DATE    LONG-HIRE-DATE
```



Produce this Report:

```
      EXAMPLES OF DEFINING DATE FIELDS

      LAST         HIRE
      NAME         DATE     DATE HIRED

BAKER           06/04/82 JUNE 4, 1982
CHRISTOPHERSON  08/15/81 AUGUST 15, 1981
JOHNSON         06/21/75 JUNE 21, 1975
JOHNSON         11/25/79 NOVEMBER 25, 1979
JONES           01/31/80 JANUARY 31, 1980
MACDONALD       07/04/82 JULY 4, 1982
MORRISON        11/30/79 NOVEMBER 30, 1979
SIMPSON         12/01/82 DECEMBER 1, 1982
THOMAS          06/04/82 JUNE 4, 1982


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the HIRE–DATE field and the LONG–HIRE–DATE field both point to the same data in the record (at column 34)
- the HIRE–DATE field is printed in the default display format since no FORMAT parm is specified in its FIELD statement
- the FORMAT parm causes the LONG–HIRE–DATE field to be printed in the LONG1 format, with the month name spelled out
- the HEADING parm specifies the column heading to use for the LONG–HIRE–DATE field

Figure 51. A report with FIELD statements that define date fields

By default, all date fields are *displayed* in MM/DD/YY format when they appear in a report (regardless of how they are stored in the record). If you would like a date field to have a different default display format, use the **FORMAT parm**. For example:

```
FIELD: HIRE-DATE  TYPE(YYMMDD)  FORMAT(LONG1)
```

The above example specifies that whenever the HIRE–DATE field is printed in a report, the LONG1 format should be used. The LONG1 format spells out the name of the month completely (for example, "JANUARY 31, 1999"). A complete list of date display formats is found on page 620.

Note: The FORMAT parm specifies the *default* display format that will be used for a field. Override display formats can always be used in other control statements to change the way the field is displayed in a particular report.

A Note About Julian Dates

You may wonder if Julian date fields require different handling from other kinds of date fields (such as Gregorian). The answer is no. Once you have defined a field with the appropriate Julian data type, you (and other users of the file) can simply forget that the date was originally stored in Julian format. You will work with that date field in exactly the same way as you work with any other date field.

Internally, Spectrum Writer converts *all* date fields from their input record format into its own standard format. Thus, all conversions required to make various kinds of date fields compatible is done for you automatically.

Specifically, this means that — even for fields stored in Julian format — you will use **date literals** in the standard MM/DD/YYYY (or MM/DD/YY) format when making comparisons to them. Here is an example of comparing a Julian date with two date literals:

```
INCLUDEIF: MY-JULIAN-DATE > 1/1/2001 AND < 12/31/2001
```

You can also compare a Julian date field with another date field stored in a different format without any special effort on your part:

```
INCLUDEIF: MY-JULIAN-DATE = MY-GREGORIAN-DATE OR MY-SMF-DATE OR MY-STCK-DATE
```

Note: The one exception to the above is if your Julian date fields contain **non-date values** with special significance (perhaps all zeros, all nines, high-value, etc.) Since such values are not valid Julian dates, Spectrum Writer considers these values to be "invalid" data. (Thus, you would see ***I*** in your report in such cases.) It is possible to test for these special cases. But to do so, you *will* need to be aware of how the field is stored in the input record. You should compare the field to an explicit character or hexadecimal literal of the correct length. For example:

```
INCLUDEIF: MY-JULIAN-DATE <> '99999'          /*COMPARE CHAR JULIAN DATE TO NINES*/
INCLUDEIF: MY-JULIAN-PACKED-DATE <> X'000000' /*COMPARE PACKED DATE TO LOW-VALUES*/
```

As far as report output goes, by default Spectrum Writer formats Julian date fields (like all other date fields) in the standard MM/DD/YY format. So again, you don't need to do anything special to have a Julian date field "re-formatted" into Gregorian in your report. Of course, you can also use an override display format to format a Julian date in any of the other date formats available.

How to Define a Time Field

This section explains:

- what a **time** field is
- which parms are **required** to define a time field
- which **optional** parms can be used when defining time fields

Most of the examples used in this section are illustrated in the sample report in **Figure 52**.

Time fields contain a time value consisting of a number of hours and minutes. Time fields can optionally contain seconds as well, and even decimal portions of a second.

Time fields often indicate the time of day that an event occurred. They can also indicate an elapsed time (the time interval between two events). There are a number of different ways that a time field can be stored in a record. Often they are stored as a 6–byte character HHMMSS fields. CICS stores some time fields as binary hundredths of seconds since midnight. The S/390 STCK machine instruction represents times as the number of "timer units" since the beginning of the 20th century.

Spectrum Writer supports all of these kinds of time fields and about two dozen others. The FIELD statement's TYPE parm tells Spectrum Writer exactly how a time field is stored in the record.

> **Note:** Once a time field has been defined, you do *not* need to remember how it is stored in the record. You may freely compare *any kind* of time field with any other time field. Spectrum Writer automatically takes care of any conversion that may be necessary.

The only parms required to define a time field are:

- fieldname
- TYPE

The following optional parms can also be used to define a time field:

- LENGTH
- DECIMAL
- ACCUM/NOACCUM

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 446.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

The **TYPE parm** indicates the exact way in which the time is stored in the record. The valid time data types are listed on page 613. Use these data types in the FIELD statement to define time fields. For example:

```
FIELD: SALES-TIME  TYPE(HHMMSS)
```

The above statement defines a field called SALES–TIME which is a 6–byte field containing a time in HHMMSS format.

These Control Statements:

```
FILE:   SALES-FILE  DDNAME(SALEFILE)
*
FIELD:  EMPL-NAME                                LENGTH(10)
FIELD:  CUSTOMER         COLUMN(48)              LENGTH(15)
FIELD:  SALES-TIME       COLUMN(42) TYPE(HHMMSS)
FIELD:  SALES-TIME-B     COLUMN(42) TYPE(HHMMSS)  FORMAT(HH-MM)
FIELD:  TIME-ON-PHONE    COLUMN(73) TYPE(SECS)    LENGTH(4) DEC(1)
FIELD:  TIME-ON-PHONE-B  COLUMN(73) TYPE(SECS)    LENGTH(4) DEC(1)
                                    FORMAT(TPIC'99:99:99') ACCUM
FIELD:  TIME-ON-PHONE-C  COLUMN(73) TYPE(SECS)    LENGTH(4) DEC(1)
                                    FORMAT(SECS)  ACCUM
                                    HEADING('SECONDS ON|TELEPHONE')
*
INPUT:   SALES-FILE
TITLE:   'EXAMPLES OF DEFINING TIME FIELDS'
COLUMNS: EMPL-NAME CUSTOMER
         SALES-TIME SALES-TIME-B
         TIME-ON-PHONE  TIME-ON-PHONE-B  TIME-ON-PHONE-C
```

Produce this Report:

```
                  EXAMPLES OF DEFINING TIME FIELDS

                                                    TIME
                                  SALES    TIME       ON
   EMPL                  SALES    TIME      ON      PHONE   SECONDS ON
   NAME     CUSTOMER     TIME     B       PHONE     B       TELEPHONE

JOHNSON    ACE ELECTRICAL  10:25:00 10:25 00:00:07.9 00:00:08        7.9
BAKER      JACKS CAFE      12:09:09 12:09 00:00:10.2 00:00:10       10.2
MORRISON   STAR MARKET     15:30:22 15:30 00:00:59.9 00:01:00       59.9
MORRISON   A1 PHOTOGRAPHY  19:05:41 19:06 00:01:00.0 00:01:00       60.0
SIMPSON    EUROPEAN DELI   08:17:57 08:18 00:00:15.0 00:00:15       15.0
JOHNSON    VILLA HOTEL     17:02:47 17:03 00:01:32.9 00:01:33       92.9
JOHNSON    MARYS ANTIQUES  14:33:10 14:33 00:00:00.0 00:00:00        0.0
BAKER      JACKS CAFE      14:31:12 14:31 00:00:23.1 00:00:23       23.1
THOMAS     YOGURT CITY     15:41:38 15:42 00:09:02.1 00:09:02      542.1
JONES      EZ GROCERY      07:58:32 07:59 00:01:21.0 00:01:21       81.0
JONES      TOY TOWN        08:01:59 08:02 00:02:00.0 00:02:00      120.0
JONES      TOY TOWN        13:52:41 13:53 00:00:52.3 00:00:52       52.3
JOHNSON    ACME BUILDING   11:48:33 11:49 00:01:42.5 00:01:43      102.5
SIMPSON    J & S LUMBER    15:30:21 15:30 00:04:05.1 00:04:05      245.1

*** GRAND TOTAL (14 ITEMS)                           00:23:32    1,412.0
```

Remarks:

- the HH–MM display format causes SALES–TIME–B to be rounded to the nearest minute
- only those fields defined with the ACCUM parm are totalled
- TIME–ON–PHONE–B uses a TPICTURE that does not include any decimal digits. The value is rounded to the nearest whole second.

Figure 52. A report with FIELD statements that define time fields

The **LENGTH parm** is generally not required for time fields. Depending on the particular data type, Spectrum Writer assumes a default length for each time field. For example, the default length of a time field in HHMMSS format is six bytes. The default length of a time field in P–HHMM format is three bytes, and so on. The default length of each time data type is shown in the table on page 613. If Spectrum Writer's default length is correct, you do not need to specify the LENGTH parm (although you may do so). However, if your field size is different than the default, you must specify its actual length using the LENGTH parm. For example:

```
FIELD: LOG-START-TIME TYPE(HHMMSS) LENGTH(8) DECIMAL(2)
```

The above statement defines a time field that occupies eight bytes in the input record. Its data type is HHMMSS, which by default is assumed to be only a 6-byte field. The extra two bytes in this field contain *hundredths of seconds*. The **DECIMAL parm** tells Spectrum Writer that the last two digits in the field are decimal digits — that is, decimal portions of a second. (You might think of this as a field containing HHMMSSHH.)

Spectrum Writer assumes that the LOG–START–TIME field defined in the preceding example occupies the eight bytes immediately after the previously defined field. If you want to explicitly specify where the field is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed on page 350. For example, if LOG–START–TIME begins in the 100th column of a record, we could define it like this:

```
FIELD: LOG-START-TIME TYPE(HHMMSS) LENGTH(8) DECIMAL(2) COLUMN(100)
```

As mentioned, you may use the **DECIMAL parm** in the FIELD statement. Do this whenever the time field contains decimal portions of seconds (for example, tenths of seconds, or hundredths of seconds). For example:

```
FIELD: LOG-TIME  LENGTH(4)  TYPE(B-SECS) DEC(2)
```

The above statement defines a field called LOG–TIME which is stored as a 4–byte B–SECS ("binary seconds") value. B–SECS fields have their time stored as the number of seconds since midnight. The DEC(2) parm indicates that the binary value actually represents hundredths of seconds since midnight.

The **ACCUM** and **NOACCUM parms** can also be used when defining time fields. They specify whether or not to *accumulate* the field when it appears as a column in a report. Fields which are accumulated receive Grand Totals at the end of the report and subtotals at control breaks. Accumulated fields also appear in any other statistical lines that appear in a report (such as average lines, maximum lines, etc.)

By default, time fields are *not* accumulated (since it makes no sense to add up various times of day). However, if you have a time field which represents a time interval or a duration you may want to total that field. Use the ACCUM parm to cause a time field to be totalled. For example:

```
FIELD: LONG-DISTANCE-TIME  TYPE(HHMMSS)  LENGTH(8)  DEC(2)  ACCUM
```

The above statement specifies that the LONG-DISTANCE-TIME field should be accumulated when it appears as a column in a report. Therefore, the LONG-DISTANCE-TIME column will be totalled at control breaks and at the end of the report. For a more detailed discussion about which fields appear in the total lines, see page 148.

Time fields, regardless of how they are stored in the input file, are normally *formatted* in your reports and PC files like this:

```
HH:MM:SS
```

However, time fields defined as containing *only* hours and minutes (the HHMM data type, for example) are formatted by default like this:

```
HH:MM
```

If you would like a time field to have a different default display format, use the **FORMAT parm**. For example:

```
FIELD: SALES-TIME  TYPE(HHMMSS)  FORMAT(HH-MM-AMPM)
```

The above example specifies that whenever the SALES–TIME field is printed in a report, the HH-MM-AMPM format should be used. This format displays the hours (in 12-hour format) and the minutes, plus either AM or PM.The seconds are not shown and the time is rounded to the nearest minute. You can also use a "time picture" to indicate how a time is to be formatted. A complete list of time display formats is found on page 622.

> **Note:** The FORMAT parm specifies the *default* display format that will be used for a field. Override display formats can always be used in other control statements to change the way the field is displayed in a particular report.

How to Define a Bit Field

This section explains:

- what a **bit** field is
- which parms are **required** to define a bit field
- which **optional** parms can be used when defining bit fields

Most of the examples used in this section are illustrated in the sample report in **Figure 53**.

Bit fields consist of only a single bit within a byte. A single bit can only have a value of 0 (zero) or 1 (one). We say that a bit with a value of 0 is "off," while a bit with a value of 1 is "on." Bit fields are often used to indicate a *status*. For example, the FULL–TIME field in the EMPL–FILE is a bit field. If the bit is on, it means that the employee is full–time. If the bit is "off," the employee is not full–time.

The only parms *required* to define a bit field are:

- fieldname
- BIT

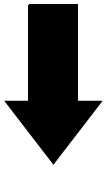
The following *optional* parms also relate specifically to bit fields:

- ONTEXT
- OFFTEXT

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 446.

These Control Statements:

```
FILE:  EMPL-FILE   DDNAME(EMPLFILE) TYPE(VSAM)

FIELD: LAST-NAME    LEN(15)  COL(4)
FIELD: FULL-TIME    BIT(1)   COL(42)

FIELD: EMPL-STATUS  BIT(1)   ONTEXT('FULL')
                             OFFTEXT('PART')
                             HEADING('FULL TIME STATUS')

FIELD: PART-TIME    BIT(1)   ONTEXT(' ')
                             OFFTEXT('PART TIME')

INPUT:    EMPL-FILE
TITLE:    'EXAMPLES OF DEFINING BIT FIELDS'
COLUMNS:  LAST-NAME  FULL-TIME  EMPL-STATUS  PART-TIME
```



Produce this Report:

```
             EXAMPLES OF DEFINING BIT FIELDS

     LAST           FULL                          PART
     NAME           TIME       FULL TIME STATUS   TIME

BAKER            FULL-TIME           FULL
CHRISTOPHERSON   FULL-TIME           FULL
JOHNSON          FULL-TIME           FULL
JOHNSON          FULL-TIME           FULL
JONES            FULL-TIME           FULL
MACDONALD        NOT FULL-TIME       PART       PART TIME
MORRISON         FULL-TIME           FULL
SIMPSON          FULL-TIME           FULL
THOMAS           FULL-TIME           FULL


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- all three bit fields point to the same bit in the record (bit 1 of the 42nd byte) since the "default location" is not incremented after FIELD statements that define bit fields
- the FULL–TIME field uses the default ONTEXT and OFFTEXT, which are based on the field name
- the EMPL–STATUS field specifies its own ONTEXT and OFFTEXT, as well as a column heading
- the PART–TIME field uses blanks for the ONTEXT, to make part time employees stand out better

Figure 53. A report with FIELD statements that define bit fields

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

The **BIT parm** is required to tell Spectrum Writer which specific bit (within a byte) the field refers to. Every byte contains 8 bits. Spectrum Writer numbers them from 1 to 8, starting with the leftmost (or "high order") bit. Here is an example of defining a bit field:

```
FIELD: FULL-TIME  BIT(1)
```

The above example defines a bit field named FULL–TIME. The BIT(1) parm specifies that the FULL–TIME field occupies the first (high order) bit within the byte.

Spectrum Writer assumes that the *byte* containing the FULL–TIME bit field occurs in the input record immediately after the previously defined field. If you want to explicitly specify where the byte containing a bit is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed on page 350. For example, if the FULL–TIME bit is located within the 42nd byte of the record, we could define it like this:

```
FIELD: FULL-TIME  BIT(1)  COLUMN(42)
```

The above statement explicitly specifies that the FULL–TIME bit is the first (high–order) bit in the 42nd byte of the record.

> **Note:** A single byte in a record will often contain more than one bit field. Therefore, the **"default location" is not incremented** after FIELD statements that define bit fields. This allows you to define multiple bit fields within the same byte of the record. For more information, see "The Default Location After Bit Fields" on page 352.

A bit field can be printed in a report just like any other kind of field. But remember that a bit field can have only one of two possible values: "on" or "off." Rather than just printing the words "on" or "off" in the report, more meaningful texts are used. One text (called the **ONTEXT**) will be printed if the bit is "on." Another text (the **OFFTEXT**) will be printed if the bit is "off."

By default, the ONTEXT is the name of the field itself, while the OFFTEXT is the word "NOT" followed by the field name itself. In the above example, the text "FULL–TIME" would print whenever the field's value is "on," and the text "NOT FULL–TIME" would print whenever the field is "off."

You may specify your own ONTEXT and OFFTEXT values by using the respective parms in the FIELD statement. For example:

```
FIELD: FULL-TIME  BIT(1)  ONTEXT('FULL')  OFFTEXT('PART')
```

The above statement causes the word FULL to print whenever the bit field is "on," and the word PART to print when the field is "off."

You may also use blanks as an ONTEXT or OFFTEXT. For example:

```
FIELD: FULL-TIME  BIT(1)  ONTEXT(' ')  OFFTEXT('PART TIME')
```

The above statement will print only a blank when the field is "on," but prints the words PART TIME when the field is "off." The use of blanks for one of the texts helps cause the other text to stand out whenever it appears in the report.

How to Specify a Field's Column Heading

This section explains:

- how to use the **HEADING parm** to specify column headings

By default, whenever a field appears as a column in a report, the **field name itself** is used as the column heading. (The name is split onto a different column heading line at each dash and underscore.)

To specify a different column heading, use the **HEADING parm** in the FIELD statement. The HEADING parm can be used when defining *any type* of field. It specifies the default column heading to be used whenever a field appears as a column in a report or PC file. For example:

```
FIELD: FIRST-NAME  LEN(15)  HEADING("EMPLOYEE'S|LAST NAME")
```

The vertical bar (|) in the HEADING parm above indicates that the column heading should be *split* onto separate lines at that point. The first part (EMPLOYEE'S) will go on one line, and the second part (LAST NAME) will go on the next line of the column heading. You may break your column heading into as many lines as you like.

> **Note:** The vertical bar is the "Shift 1" key on most mainframe terminals. When working at a PC running terminal emulation software, you will probably not see a key with this symbol on it. Some terminal emulator programs use the "pipeline" key as the vertical bar key. Some others use the right–hand square bracket key "]" for this purpose.

You can also use the HDGSEP parm of the OPTIONS statement to select a character other than the vertical bar (|) to use as the separator character. Here is an example of using a slash, rather than a vertical bar, to separate column headings lines:

```
OPTIONS: HDGSEP('/')
FIELD:   LAST-NAME  LEN(15)  HEADING("EMPLOYEE'S/LAST NAME")
```

Note that the HEADING parm simply specifies the *default* column headings that will be used for the field. Override column headings can still be specified in the COLUMNS statement to change the column heading for a particular run.

For more information on specifying column headings, see page 130.

How to Specify a Field's Location in a Record

This section explains how to specify where a field begins within a record. This discussion applies to fields of *all types*. Topics include:

- how a field's **default location** is determined
- how to use the **COLUMN or DISP parm** to specify a field's location
- how the default location works when defining **bit type fields**
- how columns are numbered in **variable length (VB)** files

Default Location

Some of the sample FIELD statements in the preceding sections did not use the COLUMN parm. When no parm is used to indicate where a field begins, a **default location** is assumed. By default, the *first* field defined for a file is assumed to begin in column one. Subsequent fields are assumed to begin immediately after the previously defined field. For example, assume that the following two statements appeared together:

```
FIELD: LAST-NAME   LENGTH(15)  COLUMN(4)
FIELD: FIRST-NAME  LENGTH(15)
```

The first field defined above (LAST–NAME) has a COLUMN parm specifying that the field begins in the 4th byte of the record. The field is 15 bytes long. The second field (FIRST–NAME) does not have a COLUMN parm. Therefore, this field is assumed to begin immediately after the LAST–NAME field. Since the LAST–NAME field begins in column 4 and occupies 15 bytes, the FIRST–NAME field would begin in column 19.

The COLUMN and DISP Parms

When defining consecutive fields in a file, you will not normally need a COLUMN parm. You will only need this parm in a few cases:

- after defining a **bit field** (the default location is *not* incremented after defining a bit field)
- when you want to "back up" and **redefine** part of a record
- when you want to **skip over** part of a record that doesn't need to be defined (such as filler)

Some companies prefer to think of fields in terms of **displacements**, rather than columns. A field's starting displacement is simply its starting column *minus one*. Spectrum Writer also lets you use the DISP (or DISPLACEMENT) parm to indicate a field's location in a record. For example, both of the following statements define the LAST–NAME field as beginning in the 4th byte of the record:

```
FIELD: LAST-NAME  LENGTH(15)  COLUMN(4)
FIELD: LAST-NAME  LENGTH(15)  DISP(3)
```

There are other methods you can use to specify a field's starting column. You can use the **location of another field** as a reference point, like this:

```
FIELD: LAST-NAME  LENGTH(15)  COLUMN(FIRST-NAME + 25)
```

The above example specifies that the LAST–NAME field begins 25 bytes after the starting column of the FIRST–NAME field. (For this statement to be acceptable, the FIRST–NAME field must have *already* been defined in a preceding FIELD statement.)

The following example specifies that the LAST–NAME field begins 20 bytes *before* the start of the FIRST–NAME field:

```
FIELD: LAST-NAME  LENGTH(15)  COLUMN(FIRST-NAME - 20)
```

Note: Be sure to put **blanks around dashes** that are used as *minus signs* (as above) to avoid confusion with dashes that are a part of the field name. (Blanks are optional around the plus sign.)

You may also use an **asterisk (*)** within the COLUMN or DISP parm. The asterisk represents the current location within the record. In other words, it represents the starting column that would be assigned if you did not specify a COLUMN parm at all. For example:

```
FIELD: LAST-NAME  LENGTH(15)  COLUMN(* + 7)
```

The above example specifies that the LAST–NAME field, rather than beginning immediately after the previously defined field, should begin 7 bytes *after* that.

You can also use the asterisk to **back up** the current location. This is useful when you want to define more than one field for a given part of the record. For example, assume the following two statements appeared together:

```
FIELD: HIRE-DATE  TYPE(MMDDYY)
FIELD: HIRE-YEAR  COLUMN(* - 2)  LENGTH(2)
```

The first statement above defines HIRE–DATE as a 6–byte date field in the format MMDDYY. The second field backs up 2 bytes and redefines the last 2 bytes of the hire date as a separate field named HIRE–YEAR. HIRE–YEAR is just a 2–byte character field containing the YY portion of the HIRE–DATE field.

The Default Location After Bit Fields

The default location is handled a little differently when working with bit fields. A single byte in a record often contains more than one bit field. Therefore, the default location *is not incremented after FIELD statements that define bit fields*. This allows you to define multiple bit fields within the same byte of the record. After the FIELD statement for the last bit that you wish to define within a byte, you *must* use the COLUMN (or DISP) parm to specify the location of the next field. For example:

```
FIELD: ACTIVE-FLAG    BIT(1)
FIELD: PARTTIME-FLAG  BIT(2)
FIELD: DELETE-FLAG    BIT(5)
FIELD: CUSTOMER       COLUMN(*+1)  LENGTH(20)
```

The first three FIELD statements above define bit fields. All three bit fields are located in the same byte of the record. The default location was not incremented after processing those FIELD statements since they defined bit fields. To define the CUSTOMER field, which begins in the *next byte* of the record, we used the COLUMN parm. The "*+1" within that parm specifies that the CUSTOMER field should begin in the current location (the byte containing the bit fields) *plus* one byte.

Column Numbering in Variable Length (VB) Files

Records in variable length (VB) flat files contain a 4–byte record prefix called the "record descriptor word" (RDW). This RDW appears before the actual user data in each record. By default, Spectrum Writer ignores this RDW. That is, a field defined as beginning in column 1 does *not* point to the RDW, but rather to the first byte of data *after* the RDW. Consider these statements:

```
FILE:  VAR-FILE  DDNAME(FILEIN)  LRECL(5000)
FIELD: NAME      COLUMN(1)       LENGTH(15)
```

Assuming that VAR–FILE is a variable length file, Spectrum Writer will ignore the 4–byte RDW at the beginning of each record. Thus, the field that begins in column 1 (NAME) is the first item we can define for this file. We cannot define a field that is within the RDW prefix of the record.

If you do not want Spectrum Writer to ignore the RDW, use the KEEPRDW keyword in the FILE statement (or in the INPUT statement). For example:

```
FILE:  VAR-FILE       DDNAME(FILEIN)  LRECL(5000)  KEEPRDW
FIELD: RECORD-LENGTH  COLUMN(1)  TYPE(HALFWORD)
FIELD: NAME           COLUMN(5)  LENGTH(15)
```

The KEEPRDW parm in the FILE statement above causes Spectrum Writer to treat the RDW as part of the input records. Thus, we defined a halfword field starting in column 1 that points within the RDW. That field (RECORD–LENGTH) will contain the length of the record (which is what is the first 2 bytes of the RDW contains). The first field after the RDW — the NAME field — now starts in column 5.

Variably Located Record Segments

Some records contain segments (consisting of one or more fields) that do not begin in a fixed column within the record. Instead, the segment begins at different offsets in different records. (SMF records are one common example of this.) Such records usually have a field that provides the "offset" to the beginning of the variably located segment.

The FIELD statement's OFFSET parm allows you to easily define fields within such variably located segments. The contents of the OFFSET parm tell Spectrum Writer the offset from the beginning of the record to the variably located segment. The OFFSET parm can contain any numeric expression. Spectrum Writer computes the value of the OFFSET parm anew for each input record (since the value can vary from record to record). It then *adds* this value to the location specified in the DISP or COLUMN parm (if there is one) to determine where the field is located within the input record.

> **Note:** In the discussion that follows, you can also use a COLUMNS parm anywhere that a DISP parm is mentioned. For brevity, we will just refer to the DISP parm.

Use an OFFSET parm in the FIELD statement of the *first field* in a variable segment. Specifying an OFFSET parm does this:

- It specifies that the field being defined is the first field in a variably located segment, and provides the offset (or "displacement") from the beginning of the record to that segment.
- It resets the default location pointer to "displacement zero." Thus, unless an explicit DISP parm is also specified, the field being defined will be at displacement zero within the variably located segment. (If needed, you can use an explicit DISP parm to specify that the field begins at some other displacement within the segment.)
- The OFFSET value remains in effect for *all subsequent FIELD statements* — until another OFFSET parm is used to change or cancel this one. This simplifies the definition of the other fields in the same segment.

For subsequent FIELD statements, the offset value remains in effect and the current location pointer is updated as usual. This lets you easily define the other fields in the same segment — generally without needing to specify either an OFFSET parm or a DISP parm. Just define the subsequent fields in the usual way. If they immediately follow the previous field, you don't need any DISP parm. If you are skipping over fields in the segment (or backing up), use a DISP parm as needed. Remember that he DISP parm now applies to the

displacement *within* the variably located segment — not to the displacement from the very beginning of the input record.

After you've defined all of the fields in the segment, you can write a FIELD statement with a *new* OFFSET parm to begin defining the fields in a different segment. Or simply cancel the offset parm by specifying OFFSET(0) in a FIELD statement.

> **Note:** Specifying OFFSET(0) parm in a FIELD statement cancels the OFFSET parm. That field (and subsequent ones) will be located at the fixed location specified by the DISP parm. Note that specifying OFFSET(0) also resets the default location pointer to zero — the beginning of the whole input record. Thus, you will probably want to use an explicit DISP (or COLUMN) parm along with the OFFSET(0) parm.

> **Note:** If you use an OFFSET parm in a member of the Spectrum Writer Copy Library, it is a good idea to have a final FIELD statement that contains an OFFSET(0) parm. (It can just be for a "dummy" field.) That way there will be no "surprises" if someone later adds additional FIELD statements "in line" for a report request. They might not be aware that an OFFSET value was still in effect and was being applied to their "in line" FIELD statements.

Let's look an example of defining variably located segments in a record. Here is a partial definition of the SMF type 30 record:

```
** FIELDS IN FIXED LOCATIONS
FLD: ID-OFFSET    DISP(32)  TYPE(FULLWORD)  /* DISP TO IDENTIFICATION SEGMENT */
FLD: ID-LEN                 TYPE(HALFWORD)
FLD: ID-NUM                 TYPE(HALFWORD)
FLD: IO-OFFSET              TYPE(FULLWORD)  /* DISP TO I/O ACTIVITY SEGMENT */
FLD: IO-LEN                 TYPE(HALFWORD)
FLD: IO-NUM                 TYPE(HALFWORD)
**
** SELECTED FIELDS FROM THE VARIABLY LOCATED IDENTIFICATION SEGMENT
FLD: JOBNAME                LEN(8)   OFFSET(ID-OFFSET)
FLD: PGMNAME                LEN(8)
FLD: STEPNAME               LEN(8)
**
** SELECTED FIELDS FROM THE VARIABLY LOCATED I/O ACTIVITY SEGMENT
FLD: NUM-CARDS              TYPE(FULLWORD)  OFFSET(IO-OFFSET)
FLD: NUM-TPUTS              TYPE(FULLWORD)  DISP(*+4)
FLD: NUM-TGETS              TYPE(FULLWORD)
**
FLD: FILLER OFFSET(0) LEN(1)     /* A "DUMMY" FIELD TO CANCEL THE OFFSET PARM */
```

In the fixed portion of the record — at displacement 32 — a fullword field name ID-OFFSET is defined. This field contains a binary number which is the offset from the beginning of the SMF record to the beginning of the "identification" segment. A few fields later is the IO-OFFSET field. Its value, similarly, is the offset from the beginning of the SMF record to the beginning of the "I/O activity" segment.

Later, you see a comment that the "identification segment" of the record follows. Then, the first field in this ID segment of the SMF record is defined — the JOBNAME field. It uses an OFFSET parm to tell Spectrum Writer where the segment containing the JOBNAME field begins. Since JOBNAME is located at displacement zero in the ID segment, no DISP parm was necessary. The next field within the ID segment, PGMNAME, also does not use a DISP parm, nor does it have an OFFSET parm. Thus, PGMNAME will immediately follow JOBNAME in the variably located ID segment. And STEPNAME similarly follows PGMNAME in that same segment.

Next comes the definition of the fields in *another* variably located segment — the I/O segment. Again, the first field in that segment, NUM-CARDS, uses an OFFSET parm to specify it's offset from the beginning of the record. The new OFFSET parm is now in effect for this field and the following fields. Notice that the NUM-TPUTS field has a DISP(* + 4) parm. Thus, instead of beginning immediately after the NUM-CARDS field, it begins 4 bytes further in the segment.

Last comes a final FIELD statement for a "dummy" field. That is, it doesn't define an actual field within the record. Its purpose is just to cancel the outstanding offset value. It does this by specifying OFFSET(0). It is a good idea to use such a trailing FIELD statement for file definitions that are stored in the copy library.

The OFFSET parms in the example above simply contained the name of a single field. However, you may use any numeric expression in the OFFSET parm. For example, to define a field that appears after an array of variable size, you might use statements similar to this:

```
FIELD: NUM-SLOTS   TYPE(COMP-3)  LENGTH(2)  /* NUMBER OF OCCURENCES IN SALE ARRAY */
...
FIELD: LAST-FIELD  OFFSET(75 + (NUM-SLOTS * 12))  LENGTH(10)
```

There are some additional examples of FIELD statements that use the OFFSET parm in "Working with SMF Records" on page 263.

Offset Errors

A *****F*** error indicator** in your report means that an "offset error" occurred for the field. Offset errors occur when the sum of the OFFSET value and the DISP value are not within the I/O area reserved for the input record. (The size of this I/O area is determined by the LRECL parm in a FILE, INPUT or READ statement.) Offset errors also occur when a computation error arises while trying to compute the OFFSET value. This includes division by zero, overflow, or any reference to a field that contains invalid data.

How to Define Arrays

Many records contain arrays. One useful way to process arrays is to "normalize" the records. (See "Using Normalization to Process Arrays" on page 237.) The FIELD statements that you use to define an array will depend on whether you plan to normalize that array.

If you are *not* normalizing a record and you want to access all the data from an array, you must define each occurrence of the array as a separate field. That is because the only way to access each occurrence of the array is to refer to it specifically by a unique fieldname.

However, if you *are* using normalization to process an array, it is not necessary to define all of the occurrences of the array. It is sufficient to define just the fields in the first occurrence of the array. (Plus, you may need to define one additional field that includes the entire first occurrence of the array. You will use that field in the NORMALIZATION parm of your INPUT statement.)

Figure 54 illustrates the different ways to define an array. Of course, for maximum flexibility to all users, you may want to include FIELD statements that make it possible to process the array using either method.

COBOL RECORD LAYOUT

```
01    SALES-HISTORY-REC
      05 NAME                  PIC X(10).
      05 CITY                  PIC X(10).
      05 NUM-SLOTS             PIC 9.
      05 SALE-ARRAY  OCCURS 6 TIMES.
         10 SALE-DATE          PIC 9(6).
         10 SALE-AMT           PIC 9(5)V9(2).
      05 RECORD-STATUS         PIC X(1).
```

SPECTRUM WRITER DEFINITION (IF NOT NORMALIZING)

```
FIELD: NAME          LENGTH(10)
FIELD: CITY          LENGTH(10)
FIELD: NUM-SLOTS     LENGTH(1) TYPE(NUM)
FIELD: SALE-DATE-1             TYPE(YYMMDD)
FIELD: SALE-AMT-1    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: SALE-DATE-2             TYPE(YYMMDD)
FIELD: SALE-AMT-2    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: SALE-DATE-3             TYPE(YYMMDD)
FIELD: SALE-AMT-3    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: SALE-DATE-4             TYPE(YYMMDD)
FIELD: SALE-AMT-4    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: SALE-DATE-5             TYPE(YYMMDD)
FIELD: SALE-AMT-5    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: SALE-DATE-6             TYPE(YYMMDD)
FIELD: SALE-AMT-6    LENGTH(7) TYPE(NUM)  DEC(2)
FIELD: RECORD-STATUS LENGTH(1)
```

SPECTRUM WRITER DEFINITION (IF NORMALIZING)

```
FIELD: NAME          LENGTH(10)
FIELD: CITY          LENGTH(10)
FIELD: NUM-SLOTS     LENGTH(1)      TYPE(NUM)
FIELD: SALE-ARRAY    LENGTH(13)     /*ENTIRE OCCURENCE*/
FIELD: SALE-DATE     COLUMN(* - 13) TYPE(YYMMDD)
FIELD: SALE-AMT      LENGTH(7)      TYPE(NUM) DEC(2)
FIELD: RECORD-STATUS COLUMN(SALE-ARRAY + 78) LENGTH(1)
```

Figure 54. Different ways to define an array depending on how it will be processed

How to Specify What File a Field Belongs To

Our examples up until now have not used the **FILE parm** of the FIELD statement. By default, fields are assumed to exist in the "current file" — the file defined in the most recent FILE statement. To specify that a field belongs to some other (previously defined) file, use the FILE parm. For example, assume that the following statements appeared together:

```
FILE:  EMPL-FILE
FIELD: LAST-NAME   COLUMN(4)  LENGTH(15)
FILE:  SALES-FILE
FIELD: EMPL-NAME   LENGTH(10)
FIELD: FIRST-NAME  LENGTH(15)  FILE(EMPL-FILE)
```

The first statement above defines a file named EMPL–FILE. The next statement defines a field named LAST–NAME. Since no FILE parm is used, that field is assumed to exist in the EMPL–FILE — the most recently defined file. The next statement defines a new file named SALES–FILE. The following statement defines a field named EMPL–NAME. It also has no FILE parm. So, it is assumed to exist in the SALES–FILE — the most recently defined file at that

point. The last statement defines a field named FIRST–NAME. This statement *does* have a FILE parm. That statement explicitly specifies that the FIRST–NAME field exists in the EMPL–FILE — *not* the most recently defined file (the SALES–FILE).

Since no COLUMN parm was specified, the FIRST-NAME field begins in the "default location" for the EMPL-FILE. Thus, the FIRST-NAME field will follow immediately after the LAST-NAME field in the EMPL-FILE.

How to Define a Field Created by a Data Exit

This section explains:

- what a **data exit** is
- which parms are **required** to define a field that uses a data exit
- which **optional** parms can be used when defining fields that use data exits

There are occasions when an external program, called a **data exit program** (or just "exit program"), must manipulate data before Spectrum Writer can use it. Examples of this include:

- data that is stored in encrypted format in a record
- date fields that are stored in an unusual format that Spectrum Writer does not directly support

Even in such situations, Spectrum Writer can still use the data to produce a report. But a data exit program must first be called to *convert* the data into a standard format that Spectrum Writer can process. For example, in the cases listed above, a data exit program could be used to:

- decrypt the encrypted data
- convert the unusual date field into a standard date field that Spectrum Writer can process

Note: Data exit programs are programs written by programmers at your own company. Appendix H, "Sample Data Exit Programs" (page 666) shows two examples of sample data exit programs.

When Spectrum Writer needs to use a data exit field in producing a report, it temporarily passes control to the data exit program. The exit program will be passed such information as: the name of the field that Spectrum Writer needs a value for; some portion of the current input record; and, a parm text.

The exit program then performs whatever processing is required, and passes back to Spectrum Writer one of the following:

- a **character** string, of the length specified in the DXRETLEN parm (explained below)
- a **numeric** value, stored as a 16–byte packed field
- a **date** value, stored as a 4–byte X'YYYYMMDD' field

- a **time** value, stored as a 16–byte packed number of seconds (or decimal parts of seconds)
- a **bit** value, stored as a 1–byte character "0" or "1"

Once an exit program has passed data back to Spectrum Writer, that data can then be used just like the data from any other field in producing reports and PC files. It can be printed, sorted on, compared with other fields, used in computations, etc.

When data exit fields are defined, several special parms must be used in the FIELD statement. These additional parms give information about: the *name* of the data exit program to execute; what data should be *passed to* that program; and, what kind of data Spectrum Writer can expect to get *back from* that program.

The parms *required* to define a field created by an exit program are:

- fieldname
- TYPE
- DXPROG
- DXRETLEN (for character fields)
- DXRETDEC (for numeric and time fields)

The following *optional* parm also relates specifically to fields created by data exits:

- DXPARM (used to pass an optional parameter string to the exit — see page 525)

In addition, the parms that specify how to *display* fields in a report (such as HEADING, FORMAT, ACCUM/NOACCUM, ONTEXT, and OFFTEXT) can also be specified for these fields.

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 446.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

The **TYPE parm** is required to tell Spectrum Writer that the field's data is not in the input record, but must be obtained by calling an exit program. It also tells what kind of data (character, numeric, date, time or bit) the exit program will return. The valid values for the TYPE parm are: CHAREXIT, NUMEXIT, DATEEXIT, TIMEEXIT and BITEXIT.

The **DXPROG parm** is required to tell Spectrum Writer the name of the program that should be called to create the field's data.

The **DXRETLEN parm** is required for CHAREXIT fields. This parm specifies the length of the character data that will be returned to Spectrum Writer by the exit program.

The **DXRETDEC parm** is required for NUMEXIT and TIMEEXIT fields. This parm specifies the number of decimal digits that will be included in the packed number returned to Spectrum Writer by the exit program.

Let's consider an example of a file that contains names stored in a special encrypted format. Assume that the encrypted name starts in column 15 and is 20 bytes long. Also

assume that a program named DCRYPROG can be used to decrypt such names into a clear text 18-byte name. Consider the following FIELD statement:

```
FIELD: CLEAR-NAME  COLUMN(15)  LENGTH(20)  TYPE(CHAREXIT)
       DXPROG('DCRYPROG')  DXRETLEN(18)
```

The above statement defines a field named CLEAR–NAME. The contents of this field will be the name in "clear" format (that is, not encrypted). But in order to get the decrypted name, Spectrum Writer must call an exit program. Therefore, the field is defined with a **TYPE parm** of CHAREXIT. This specifies that a data exit program will be used, and that the exit program will return *character* type data to Spectrum Writer.

The **DXPROG parm** supplies the name of the exit program to call. In this example, a program named DCRYPROG will be called. Under OS/390, a load module by this name must exist in one of the job's STEPLIB or JOBLIB libraries. (Under VSE, a phase by this name must be in a sublibrary named in the "// LIBDEF PHASE,SEARCH=..." JCL statement.)

When Spectrum Writer calls that program it will pass it the 20 byte encrypted name, which begins in column 15 of the record. Note that the **COLUMN** and **LENGTH** parms identify the data to be passed *to* the data exit program.

The **DXRETLEN** parm tell Spectrum Writer to expect an 18–byte character value *back* from the exit program. It is this 18-byte character value returned from the exit program that will be used whenever the CLEAR–NAME field appears in a report.

Here is an example of using a data exit to create a **date field**. Assume that in column 17 of the input record there are 2 bytes that contain a date, stored in a special "in–house" format. A program called DATECONV has been written to convert this date into the standard 4–byte X'YYYYMMDD' format date that Spectrum Writer uses internally. The following statement could be used to define the field:

```
FIELD: SPECIAL-DATE  COLUMN(17)  LENGTH(2)  TYPE(DATEEXIT)  DXPROG('DATECONV')
```

The above statement defines a field named SPECIAL–DATE that can be used just as any other date field in Spectrum Writer. It can be compared to other dates, printed using any date display format, etc.

Following is an example of a data exit used to create a **numeric field**. Assume that bytes 5 through 7 of the input record contain a key that can be used to read a special "in–house" data base file. The data base file contains the unit cost of a product. Since Spectrum Writer cannot read the data base file directly, an exit program named READCOST has been written to read a record from the file and return the unit cost as a 16–byte packed number. The packed numeric value returned by the exit program will contain 2 decimal digits.

```
FIELD: UNIT-COST  COLUMN(5)  LENGTH(3)  TYPE(NUMEXIT)
       DXPROG('READCOST')  DXRETDEC(2)
```

The last example is of a **bit field** that is created using a data exit program. In this example, we want to define a bit field that tells whether a report job is running on the shop's production machine, or on its development machine. This information is not stored in any record. But a program named CHEKMACH can determine which machine it is running on. In this example, we don't specify a COLUMN or LENGTH, because the data exit program does not require any data from our input file in order to do its processing. This exit program will return an "on" value ("1") if the production machine is running, and an "off" value ("0') if

the development machine is running. The optional ONTEXT and OFFTEXT parms have been used in this example.

```
FIELD: MACHINE  TYPE(BITEXIT)  DXPROG('CHEKMACH')  ONTEXT('PROD')  OFFTEXT('DEV')
```

Note: The EXITPARM parm in the *FILE statement* (page 534) may also be useful when working with fields created in data exits. It defines an exit parm text that is passed to the data exits for *all* fields in the file. This parm text is *in addition* to the parm text in the FIELD statement's DXPARM parm, which is specific to an individual field.

Keeping Your File Definitions in a Copy Library

This section explains:

- how to define files **without using a copy library**
- how to simplify the file definition process by **using a copy library** to store your FILE and FIELD statements

The first part of this chapter explained how to write FILE and FIELD statements (called "definition statements"). But where should you *put* your definition statements? This section discusses two approaches to handling these definition statements:

- you can code the definition statements "**in–line**," including them right along with the other control statements for each report
- or, a better way is to save the definition statements in the Spectrum Writer **Copy Library**, where they can be *automatically* accessed when needed

The following sections describe these two methods.

Including the Definition Statements "In–Line"

If you like, you can produce Spectrum Writer reports and PC files without using a copy library at all. Simply include the necessary FILE and FIELD statements *ahead of* the other control statements (that describe the report or PC file). **Figure 55** shows an example of a report which has the necessary definition statements included ahead of the other control statements. No copy library was involved in producing this report. (**Figure 56** on page 362 shows the same example under VSE.)

Note that if you use this method, you only need to define those fields that are actually used in the report. It is not necessary to define every field in the file.

If a report requires more than one input file (by using a READ statement) be sure to include the definition statements for *each* of those files at the beginning of your control statements.

This JCL:

```
//SPECTWTR JOB 'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,                  SPECTRUM WRITER
//               REGION=2048K
//STEPLIB  DD  DSN=SPECTWTR.LOADLIB,DISP=SHR    LOADLIB TO USE
//SWLIST   DD  SYSOUT=*                         CONTROL LISTING
//SWOUTPUT DD  SYSOUT=*                         REPORT OUTPUT
//SYSOUT   DD  SYSOUT=*                         SORT STATISTICS
//SYSUDUMP DD  SYSOUT=*                         DUMP OUTPUT
//SORTWK01 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))     SORT WORK FILE
//SORTWK02 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))     SORT WORK FILE
//SORTWK03 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))     SORT WORK FILE
//SALEFILE DD  DSN=PROD.SALES.DATA,DISP=SHR     SALES FILE
//SYSIN    DD  *                                CONTROL STATEMENTS
**** THESE STATEMENTS DEFINE THE SALES-FILE
FILE:    SALES-FILE  DDNAME(SALEFILE)  LRECL(80)
FIELD:   EMPL-NAME                LENGTH(10)
FIELD:   EMPL-NUM                 LENGTH(3)
FIELD:   AMOUNT      COLUMN(22)   LENGTH(6)  TYPE(NUM) DEC(2)
FIELD:   TAX                      LENGTH(4)  TYPE(NUM) DEC(2)
FIELD:   SALES-DATE  COLUMN(36)               TYPE(YYMMDD)
FIELD:   CUSTOMER    COLUMN(48)   LENGTH(15)
**** THESE STATEMENTS REQUEST A REPORT FROM THE SALES-FILE
INPUT:   SALES-FILE
COLUMNS: EMPL-NAME  EMPL-NUM  SALES-DATE  CUSTOMER  AMOUNT  TAX
SORT:    EMPL-NAME
//*
```

Produces this Report:

```
MON  06/12/95    9:02 AM      DATA FROM SALES-FILE        PAGE    1

   EMPL    EMPL  SALES
   NAME    NUM   DATE    CUSTOMER        AMOUNT      TAX

BAKER       044 03/26/92 JACKS CAFE             137.00        8.22
BAKER       044 04/12/92 JACKS CAFE             135.75        8.15
JOHNSON     037 03/12/92 ACE ELECTRICAL         101.38        6.09
JOHNSON     037 04/01/92 VILLA HOTEL            234.45       14.07
JOHNSON     039 04/05/92 MARYS ANTIQUES           9.98        0.60
JOHNSON     039 04/16/92 ACME BUILDING          500.00       30.00
JONES       036 04/15/92 EZ GROCERY              10.25        0.62
JONES       036 04/15/92 TOY TOWN                10.25        0.62
JONES       036 04/15/92 TOY TOWN               121.76        7.31
MORRISON    042 03/29/92 STAR MARKET             44.35        2.66
MORRISON    042 03/30/92 A1 PHOTOGRAPHY          29.65        1.78
SIMPSON     041 04/01/92 EUROPEAN DELI           14.99        0.90
SIMPSON     041 04/30/92 J & S LUMBER            23.87        1.43
THOMAS      045 04/14/92 YOGURT CITY              9.98        0.60


*** GRAND TOTAL (14 ITEMS)                   1,383.66       83.05
```

Remarks:

- the EMPL–FILE is defined with the FILE statement before the INPUT statement that refers to it
- each of the fields used in the report are defined with FIELD statements before being referred to
- no SWCOPY DD is needed in the JCL to run this report, since the copy library is not used

Figure 55. A Spectrum Writer report that does not use a copy library — OS/390

This JCL:

```
// JOB    SPECTWTR
// ASSGN  SYS010,SYSLST                CONTROL STATEMENT LISTING
// ASSGN  SYS011,006                   REPORT OUTPUT
// LIBDEF PHASE,SEARCH=LIB.SPECTWTR
// DLBL   SALEFIL,'SALES.MASTER.FILE'
// EXTENT SYS015,,,,6764,1000
// EXEC   SPECTWTR,SIZE=(SPECTWTR,300K)
**** THESE STATEMENTS DEFINE THE SALES-FILE
FILE:    SALES-FILE  ATTR(DASD,'SALEFIL',80,160)
FIELD:   EMPL-NAME               LENGTH(10)
FIELD:   EMPL-NUM                LENGTH(3)
FIELD:   AMOUNT     COLUMN(22)   LENGTH(6)  TYPE(NUM)  DEC(2)
FIELD:   TAX                     LENGTH(4)  TYPE(NUM)  DEC(2)
FIELD:   SALES-DATE COLUMN(36)               TYPE(YYMMDD)
FIELD:   CUSTOMER   COLUMN(48)   LENGTH(15)
**** THESE STATEMENTS REQUEST A REPORT FROM THE SALES-FILE
INPUT:   SALES-FILE
COLUMNS: EMPL-NAME  EMPL-NUM  SALES-DATE  CUSTOMER  AMOUNT  TAX
SORT:    EMPL-NAME
/*
/&
```

Produces this Report:

```
MON  06/12/95    9:02 AM      DATA FROM SALES-FILE        PAGE    1

   EMPL    EMPL  SALES
   NAME    NUM   DATE    CUSTOMER        AMOUNT      TAX

BAKER       044 03/26/92 JACKS CAFE             137.00        8.22
BAKER       044 04/12/92 JACKS CAFE             135.75        8.15
JOHNSON     037 03/12/92 ACE ELECTRICAL         101.38        6.09
JOHNSON     037 04/01/92 VILLA HOTEL            234.45       14.07
JOHNSON     039 04/05/92 MARYS ANTIQUES           9.98        0.60
JOHNSON     039 04/16/92 ACME BUILDING          500.00       30.00
JONES       036 04/15/92 EZ GROCERY              10.25        0.62
JONES       036 04/15/92 TOY TOWN                10.25        0.62
JONES       036 04/15/92 TOY TOWN               121.76        7.31
MORRISON    042 03/29/92 STAR MARKET             44.35        2.66
MORRISON    042 03/30/92 A1 PHOTOGRAPHY          29.65        1.78
SIMPSON     041 04/01/92 EUROPEAN DELI           14.99        0.90
SIMPSON     041 04/30/92 J & S LUMBER            23.87        1.43
THOMAS      045 04/14/92 YOGURT CITY              9.98        0.60


*** GRAND TOTAL (14 ITEMS)                   1,383.66       83.05
```

Remarks:

- the EMPL–FILE is defined with the FILE statement before the INPUT statement that refers to it
- each of the fields used in the report are defined with FIELD statements before being referred to
- no OPTIONS: SUBLIB parm is needed to run this report, since a copy library is not used

Figure 56. A Spectrum Writer report that does not use a copy library — VSE

Using the Spectrum Writer Copy Library

There is a better way to handle the definition statements. Spectrum Writer can *automatically* access the definition statements it needs for a particular report by using a "copy library." In OS/390, this copy library is just a regular partitioned data set (PDS). In VSE, this copy library is just a regular Librarian sublibrary. The copy library will have one **member** for each of your company's files that have been defined. The FILE and FIELD statements for each file will be kept in these members.

> **Note to OS/390 Programmers:** The Spectrum Writer Copy Library works in much the same way as a Cobol copybook library (SYSLIB DD). We recommend that you create a new PDS to serve exclusively as your Spectrum Writer Copy Library. (However, you can also use an existing 80–byte PDS, if you prefer.) Use the SWCOPY DD in your execution JCL to tell Spectrum Writer what PDS you are using as the copy library. See "Setting Up File Definitions — OS/390" (page 420) for detailed information on setting up the Spectrum Writer Copy Library.
>
> **Note to VSE Programmers:** The Spectrum Writer Copy Library works in much the same way as a Cobol copybook library. We recommend that you define a separate sublibrary to serve exclusively as your Spectrum Writer Copy Library. (However, you can also use an existing sublibrary, if you prefer.) Use the SUBLIB parm — in an OPTIONS statement — to tell Spectrum Writer the name of your copy library. The *member type* for all members should be SPECTWTR. (Or, use the OPTIONS statement MEMTYPE parm if you need to use a different member type. See page 566.) See "Setting Up File Definitions — VSE" (page 437) for detailed information on setting up the Spectrum Writer Copy Library.

There are several advantages to keeping the FILE and FIELD statements in a copy library. Among them are: easier maintenance of the definitions; standardization of file definition among the various jobs that use the same file; and the ability for users to request reports more easily, without concerning themselves each time with writing definition statements.

To add a file's definition to the copy library, simply create a new member in the copy library. The member name can be either the **file name** itself (if it conforms to the naming rules for PDS or Librarian members), or it can be some other name (in which case you'll create an alias entry for it, as described on page 367). After you have created a member in the copy library for a file, simply save its FILE and FIELD statements there. You can also include any COMPUTE statements that are commonly used with the file. That's all there is to adding a file to the Spectrum Writer Copy Library.

Automatic Copying

Once a file's definition statements have been stored in the copy library, Spectrum Writer will automatically copy and process those statements whenever they are needed in order to produce a report or PC file. You remember that the INPUT and the READ statement identify files as inputs in a run. By default, whenever either of these statements names a file that has not yet been defined, Spectrum Writer attempts to copy control statements from the copy library member that corresponds to that file. Those control statements then define the file for Spectrum Writer.

Thus, each input file to a report is *automatically* defined for you as it is needed. You don't need to concern yourself with the FILE and FIELD statements every time you request a report or PC file.

Figure 57 (OS/390) and **Figure 58** (VSE) show a sample report that allows the INPUT statement to automatically copy the FILE and FIELD control statements from the copy library. Most of the examples in this manual also use this method — that is why you don't see the FILE and FIELD statements explicitly specified in most cases. Appendix F, "Files Used in Examples" (page 648) show the contents of the copy library members that define the sample files used in this manual.

By default, the control statements copied from the copy library are not printed in the control listing. If you would like to see all of the control statements that are copied from the copy library, add the LIST(YES) parm to your INPUT or READ statement, like this:

```
INPUT: EMPL-FILE  LIST(YES)
```

The INPUT statement above will cause all of the statements copied from the copy library to be printed in the control listing. If you are having errors involving "undefined files" or "undefined fields," you should use the LIST(YES) parm to see exactly how the file and fields are being defined.

If for any reason you do not want an automatic copy performed for an INPUT or READ statement, you may use the COPY(NO) parm, like this:

```
INPUT: EMPL-FILE  COPY(NO)
```

The above statement specifies EMPL–FILE as the input file and requests that no automatic copy be performed from the copy library. (Also, remember that the default is *not* to perform a copy if the file named in the INPUT or READ statement has *already* been defined some other way.)

The COPY Statement

In addition to file definition statements, the copy library can also be used to store *any* commonly used group of control statements. To explicitly copy the contents of a copy library member into your control statements, use the COPY statement (page 516).

For example, you might store a set of complicated COMPUTE statements that are used by many reports. Or, if you frequently run reports that use multiple input files, you could store the INPUT statement, any COMPUTE statements needed to create the read keys, and the READ statements all as one member of the copy library. That way the end–users would not need to remember how to link all of the input files. They could just begin their report request with a COPY statement that accomplishes all of that for them.

Under OS/390, the COPY statement can also copy standard *sequential* datasets (not partitioned). If your FILE and FIELD statements are stored in a dataset other than a PDS, you may want to use the COPY statement to include them in your report request. Be sure to put the COPY statement *before* the INPUT or READ statement.

These Control Statements:

```
INPUT:    EMPL-FILE
TITLE:    'USING THE COPY LIBRARY TO DEFINE FIELDS'
COLUMNS:  LAST-NAME  FIRST-NAME  HIRE-DATE
```

Produce this Report:

```
USING THE COPY LIBRARY TO DEFINE FIELDS

     LAST            FIRST        HIRE
     NAME            NAME         DATE

BAKER           VIVIAN          06/04/82
CHRISTOPHERSON  MELISSA         08/15/81
JOHNSON         LINDA           11/25/79
JOHNSON         THOMAS          06/21/75
JONES           JERRY           01/31/80
MACDONALD       RICHARD         07/04/82
MORRISON        MICHAEL         11/30/79
SIMPSON         TIMOTHY         12/01/82
THOMAS          MARTIN          06/04/82


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the SWCOPY DD (in the execution JCL) would identify the PDS to use as the copy library
- as the INPUT statement is processed, the EMPLDEF copy library member is automatically copied into this report request. That member contains the definition statements for the EMPL–FILE.
- the following line appears in the SWALIAS member of the copy library:

```
EMPL-FILE = EMPLDEF
```

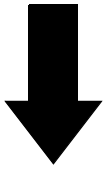
- the following statements are stored in the EMPLDEF member of the copy library:

```
FILE:  EMPL-FILE  DDNAME(EMPLDD)  TYPE(VSAM)
FIELD: LAST-NAME  COLUMN(4)       LENGTH(15)
FIELD: FIRST-NAME                 LENGTH(15)
FIELD: HIRE-DATE                  TYPE(YYMMDD)
```

Figure 57. A report which uses Spectrum Writer's Copy Library — OS/390

These Control Statements:

```
OPTIONS:  SUBLIB('LIB.SPECTWTR')
INPUT:    EMPL-FILE
TITLE:    'USING THE COPY LIBRARY TO DEFINE FIELDS'
COLUMNS:  LAST-NAME  FIRST-NAME  HIRE-DATE
```



Produce this Report:

```
USING THE COPY LIBRARY TO DEFINE FIELDS

     LAST            FIRST        HIRE
     NAME            NAME         DATE

BAKER           VIVIAN          06/04/82
CHRISTOPHERSON  MELISSA         08/15/81
JOHNSON         LINDA           11/25/79
JOHNSON         THOMAS          06/21/75
JONES           JERRY           01/31/80
MACDONALD       RICHARD         07/04/82
MORRISON        MICHAEL         11/30/79
SIMPSON         TIMOTHY         12/01/82
THOMAS          MARTIN          06/04/82


*** GRAND TOTAL (9 ITEMS)
```

Remarks:

- the OPTIONS statement names LIB.SPECTWTR as the Librarian sublibrary to use as the Spectrum Writer Copy Library for this run.
- as the INPUT statement is processed, the EMPLDEF.SPECTWTR copy library member is automatically copied into this report request. That member contains the definition statements for the EMPL–FILE.
- the following line appears in the SWALIAS.SPECTWTR member of the copy library:

```
EMPL-FILE = EMPLDEF
```

- the following statements are stored in the EMPLDEF.SPECTWTR member of the copy library

```
FILE:  EMPL-FILE  ATTR(VSAM,'EMPLDD',100)
FIELD: LAST-NAME  COLUMN(4)  LENGTH(15)
FIELD: FIRST-NAME            LENGTH(15)
FIELD: HIRE-DATE             TYPE(YYMMDD)
```

Figure 58. A report which uses Spectrum Writer's Copy Library — VSE

How to Use a Copy Library Alias

This section explains:

- **which member** of the copy library will be copied
- how to **create an alias entry** for use with the copy library

As mentioned in the preceding section, whenever an INPUT or READ statement is encountered for a file name which has not been defined, Spectrum Writer attempts to copy a member from the copy library to define the file. Which member of the copy library is copied? The member name used will be either:

- the member name specified by an "alias entry" for the file name, if any, or
- the file name itself, if that name is valid for use as a member name

If there is no alias entry for a file, and the file name is not valid as a member name, no copy is attempted. If a copy is attempted, but the member does not exist in the copy library, no copy is performed. Processing continues normally in either of these cases. The failure to find a member to copy is not considered an error when processing INPUT and READ statements.

Alias entries are kept in a special member of the copy library. That member is named SWALIAS. The purpose of an alias entry is to relate a Spectrum Writer file name (which can be up to 70 characters long) to the 8–byte name of the copy library member where that file's definitions are stored. When the file name and the member name are the identical, no alias is needed. Thus, if you have a file named PAYROLL and you keep its file definition statements in a member named PAYROLL, no alias entry would be needed for that file.

But, if you'd like to use longer, more user–friendly file names in your Spectrum Writer statements, you can certainly do so. You'll just need to add an alias entry to the special member named SWALIAS in your copy library. For example, let's say we wanted to call our payroll file HEADQUARTERS–PAYROLL. That name is too big to use as the member name in the copy library. So, you would pick a shorter member name to keep the file definition statements in — say HQPAYROL. Now just add an alias entry like this within SWALIAS:

```
HEADQUARTERS-PAYROLL = HQPAYROL
```

The above alias entry tells Spectrum Writer that the file definition statements for the HEADQUARTERS–PAYROLL file are stored in the member named HQPAYROL. "HEADQUARTERS–PAYROLL" is the name that users will use for the file in Spectrum Writer control statements (such as the INPUT statement). It's also the name you will use in the FILE statement when defining the file. "HQPAYROL" will only be used internally by Spectrum Writer as the member name for reading the definition statements from the copy library.

Consider the following statement:

```
INPUT: HEADQUARTERS-PAYROLL
```

When Spectrum Writer encounters the above INPUT statement it searches the SWALIAS copy library member for a line that begins with "HEADQUARTERS–PAYROLL." It will find the alias entry shown earlier that names HQPAYROL as the member name. Spectrum Writer will then copy the control statements from the HQPAYROL member of the copy library. Those statements define the HEADQUARTERS–PAYROLL file.

Here are some additional points about the SWALIAS member:

- The alias entries in SWALIAS *do not* have to be in alphabetical order.
- Each file name may appear only once in SWALIAS.
- You may include **comment lines** in the SWALIAS member by putting an asterisk in the first column of the line.

Appendix F, "Files Used in Examples" (page 648) shows the contents of the SWALIAS member used in producing the sample reports in this manual.

Defining One–Time Fields

The FIELD statements for a file are normally kept in the copy library member for that file. You may, however, want to add one or more FIELD statements *of your own* for a particular run.

This usually occurs when you want to define some part of a record differently than the way it is defined in the copy library. For example, you may want to *subdivide* a date field into its year, month, and day components. Or, you might want to define a cost center field as *numeric*, whereas it is defined as character in the copy library.

It is very easy to add your own FIELD statements for use in your report. Just include them in-line, along with your other control statements. Put them somewhere *after* the INPUT or READ statement for the file, and *before* the first statement that refers to the field. Remember to choose different names for your fields — ones that are not used in the copy library FIELD statements.

As an example, let's say that we want to produce a report of all employees hired in the month of January (of any year). To do this, we need a field that contains the month that an employee was hired. There is no such field defined in the regular FIELD statements contained in the copy library. The closest thing is the HIRE–DATE field, which is defined as a YYMMDD date. We could do the following:

```
INPUT:     EMPL-FILE
FIELD:     HIRE-MONTH  COLUMN(HIRE-DATE+2)  LENGTH(2)  TYPE(NUM)
INCLUDEIF: HIRE-MONTH = 1
TITLE:     'EMPLOYEES HIRED IN JANUARY OF SOME YEAR'
COLUMNS:   HIRE-MONTH  LAST-NAME  FIRST-NAME  HIRE-DATE
```

As soon as Spectrum Writer encounters the above INPUT statement, the copy library members for the EMPL–FILE are processed. These statements define all of the regular fields in the EMPL–FILE. However, in this report we want the 2–byte *month* portion of the HIRE–DATE field defined as a separate field. So, we add our own FIELD statement to define a new field called HIRE–MONTH. It is located 2 bytes after the start of the HIRE–DATE field — at the MM portion of the YYMMDD date. The HIRE–MONTH field is 2 bytes long, and is defined as a numeric field. The INCLUDEIF statement can now refer to the HIRE–MONTH field, and select just those records with a month value of 1. We also list the new HIRE–MONTH field in the COLUMNS statement, along with a number of the regular fields from the EMPL–FILE.

Here is another example of defining an additional field in-line.

```
OPTIONS:   MAINFRAME
INPUT:     EMPL-FILE
FIELD:     RECORD  COLUMN(1)  LENGTH(150)
INCLUDEIF: DEPT-NUM = 2
COLUMNS:   RECORD
```

In this example, we want to create an output file, rather than a report. We want to select just the EMPL–FILE records for employees in department 2, and write those records to an output file. To do this, we defined an additional field named RECORD. This is a character field that includes the entire EMPL–FILE record. We use the INCLUDEIF statement to select only those records whose DEPT–NUM field is equal to 2. Our COLUMNS statement simply lists the single RECORD field. Thus, our output file contains the complete EMPL–FILE record for those employees in department 2.

Note: If your report uses multiple input files, you may need to use the FILE parm in your FIELD statement to specify which file your new field exists in. If the FILE parm is omitted, your FIELD statements will be assumed to belong to the *most recently defined file*.

Using Cobol and Assembler Record Layouts

This section explains how to use Cobol or Assembler record layouts with Spectrum Writer.

Earlier in this chapter you learned how to use FIELD statements to define an input file to Spectrum Writer. Spectrum Writer can also interpret most Cobol and Assembler record layouts. If you have such a record layout for your file, it is not necessary to write FIELD statements to define it.

There are two ways to use Spectrum Writer's Cobol and Assembler interpreter.

1. **In a "live" run**. Provide a Cobol or Assembler record layout to Spectrum Writer and produce a custom report or PC file in the same run. With this method, you never create a standard Spectrum Writer file definition.
2. **In a "conversion" run**. Provide a Cobol or Assembler record layout to Spectrum Writer and let it write corresponding FIELD statements to an output file. Save these FIELD statements in your Spectrum Writer Copy Library for use in future runs. This method gives you greater flexibility because you can modify and customize the FIELD statements created by Spectrum Writer. This lets you take advantage of features available in FIELD statements that aren't available in Cobol or Assembler layouts (such as specifying column headings and display formats).

Spectrum Writer has two special control statements that are used when working with Cobol or Assembler record layouts. The COBOL statement tells Spectrum Writer that a Cobol record layout is about to follow. The ASM statement tells Spectrum Writer that an Assembler record layout follows. The following sections describe how to use these statements in both "live" and "conversion" runs.

Terminology: to avoid ambiguity when using the words "Cobol" and "COBOL statement," we have used the following convention throughout this chapter:

Cobol (spelled with mixed case letters) refers to the programming language. Thus "Cobol statement" refers to a line of code in a Cobol program.

COBOL (spelled in upper case letters) refers to the Spectrum Writer control statement by that name. Thus "COBOL statement" means the Spectrum Writer control statement that begins with the prefix "COBOL:".

Live Runs Using Cobol Record Layouts

This section shows how to request reports (or PC files) using a Cobol record layout to define the input file. No additional data definition is required.

Figure 59 shows an example of a such report. Let's examine the Spectrum Writer control statements shown in the top box of that figure.

A FILE statement is always required before fields can be defined. It tells Spectrum Writer the name of the file that the fields belong to. In this case, we named the file SALES–FILE. Normally a number of FIELD statements would then follow to define the fields in the file. But in this case a COBOL statement follows instead.

The COBOL statement tells Spectrum Writer that subsequent control statements will be in the Cobol language. After the COBOL statement, actual lines of a Cobol record layout appear. The Cobol code must be error-free and must be formatted according to the rules of Cobol syntax. (For example, the first 6 columns are reserved for sequence numbers, column 7 is reserved for continuation indicators or comment indicators, etc.) Spectrum Writer processes the Cobol record layout, noting the names of the Cobol fields and their characteristics. Internally, Spectrum Writer creates the equivalent of a FIELD statement for each Cobol field in the record layout.

> **Note:** See "Technical Notes on Cobol Support" (page 385) for certain limitations on the Cobol syntax that Spectrum Writer accepts.

Spectrum Writer continues treating each subsequent line as Cobol code until it reaches a line that begins with a Spectrum Writer control statement prefix. In this example, the line beginning "INPUT:" is recognized as a Spectrum Writer control statement. So, starting with the INPUT statement, the lines are no longer treated as Cobol code. (The scope of the COBOL statement is discussed more fully on page 384.)

After the Cobol record layout, we simply resumed the report request in the normal way. The INPUT statement specifies the input file for the report. It is the SALES–FILE that we just defined using the Cobol record layout.

The COLUMNS statement specifies which fields are wanted in the report. In the COLUMNS statement we can refer to any of the fields defined in the Cobol record layout. The field names used are the same names that appeared in the Cobol layout. By default, the column headings will also be the Cobol field names, broken apart at the dashes. Of course, you can specify an override column heading, if you like.

Fields defined by a Cobol record layout can be used in all of the same ways as fields defined with FIELD statements. You can use the Cobol field names in SORT statements, COMPUTE statements, BREAK statements, and so on.

These Control Statements:

```
FILE: SALES-FILE  DDNAME(SALEFILE)  LRECL(80)
COBOL:
       01    SALES-REC.
             05 EMPL-NAME        PIC X(10).
             05 EMPL-NUM         PIC X(3).
             05 BACKUP-EMPL-NUM  PIC X(3).
             05 REGION           PIC X(5).
             05 AMOUNT           PIC 9999V99.
             05 TAX              PIC 99V99.
             05 COMMISSION-RATE  PIC 9V999.
             05 SALES-DATE       PIC 9(6).
             05 SALES-TIME       PIC 9(6).
             05 CUSTOMER         PIC X(15).
             05 TELEPHONE        PIC 9(10).
             05 TIME-ON-PHONE    PIC 999V9.
             05 PRODUCT-CODE     PIC X(3).
             05 FILLER           PIC X(1).
INPUT:   SALES-FILE
COLUMNS: EMPL-NAME
         SALES-DATE(PIC'999999',NOACCUM)
         SALES-TIME(PIC'999999',NOACCUM)
         CUSTOMER
         AMOUNT
         TAX
```

Produce this Report:

```
TUE  02/14/95    8:27 AM      DATA FROM SALES-FILE       PAGE     1

   EMPL    SALES  SALES
   NAME    DATE   TIME    CUSTOMER        AMOUNT       TAX

JOHNSON    950312 102500 ACE ELECTRICAL          101.38        6.09
BAKER      950326 120909 JACKS CAFE              137.00        8.22
MORRISON   950329 153022 STAR MARKET              44.35        2.66
MORRISON   950330 190541 A1 PHOTOGRAPHY           29.65        1.78
SIMPSON    950401 081757 EUROPEAN DELI            14.99        0.90
JOHNSON    950401 170247 VILLA HOTEL             234.45       14.07
JOHNSON    950405 143310 MARYS ANTIQUES            9.98        0.60
BAKER      950412 143112 JACKS CAFE              135.75        8.15
THOMAS     950414 154138 YOGURT CITY               9.98        0.60
JONES      950415 075832 EZ GROCERY               10.25        0.62
JONES      950415 080159 TOY TOWN                121.76        7.31
JONES      950415 135241 TOY TOWN                 10.25        0.62
JOHNSON    950416 114833 ACME BUILDING           500.00       30.00
SIMPSON    950430 153021 J & S LUMBER             23.87        1.43


*** GRAND TOTAL (    14 ITEMS)                1,383.66       83.05
```

Figure 59. A report produced using a Cobol record layout

Notice in the COLUMNS statement that we used two special parms for the SALES–DATE and SALES–TIME fields. Those two fields were defined as numeric values in the Cobol record layout. The PIC'999999' parm specifies how those numeric values should be formatted in the report. Otherwise, they would have been formatted in the default way for numeric fields — as ZZZ,ZZ9. And, the NOACCUM parm indicates that those fields should not be accumulated (totalled). Otherwise, those columns would have been totalled in the Grand Total line. For more information, see "Handling Date and Time Fields in Record Layouts" on page 375.

Note: By default, the Cobol record layout defines fields for the most recently defined file. Use a FILE parm in the COBOL statement if the fields belong to some other file (see page 495).

Note: To see a listing of the internal FIELD statements that Spectrum Writer creates from a Cobol layout, add the **SHOWFLDS(YES)** parm to the COBOL statement:

```
COBOL: SHOWFLDS(YES)
```

If we had used the above statement in the report in **Figure 59**, the following FIELD statements would have appeared in the control listing:

```
FIELD: SALES-REC       LEN(80) COL(1)
FIELD: EMPL-NAME       LEN(10) COL(1)
FIELD: EMPL-NUM        LEN(3)
FIELD: BACKUP-EMPL-NUM LEN(3)
FIELD: REGION          LEN(5)
FIELD: AMOUNT          LEN(6)   TYPE(NUM) DEC(2)
FIELD: TAX             LEN(4)   TYPE(NUM) DEC(2)
FIELD: COMMISSION-RATE LEN(4)   TYPE(NUM) DEC(3)
FIELD: SALES-DATE      LEN(6)   TYPE(NUM)
FIELD: SALES-TIME      LEN(6)   TYPE(NUM)
FIELD: CUSTOMER        LEN(15)
FIELD: TELEPHONE       LEN(10)  TYPE(NUM)
FIELD: TIME-ON-PHONE   LEN(4)   TYPE(NUM) DEC(1)
FIELD: PRODUCT-CODE    LEN(3)
FIELD: FILLER#001      LEN(1)
```

Note: You can use the **NOSEQ parm** in the COBOL statement to prevent Spectrum Writer from validity-checking the sequence numbers in your Cobol record layout. Use this parm if the Cobol record layout you use has non–numerics in columns 1 through 6 and you do not want warning messages to appear in the control listing. (See page 495.)

Live Runs Using Assembler Record Layouts

This section shows how to request reports (or PC files) using an Assembler record layout to define the input file. No additional data definition is required.

Figure 60 shows an example of such a report. Let's examine the Spectrum Writer control statements shown in the top box of that figure.

A FILE statement is always required before fields can be defined. It tells Spectrum Writer the name of the file that the fields belong to. In this case, we named the file SALES–FILE. Normally a number of FIELD statements would then follow to define the fields in the file. But in this case an ASM statement followed instead.

These Control Statements:

```
FILE: SALES-FILE  DDNAME(SALEFILE)  LRECL(80)
ASM:
SALESREC DS   0CL80
EMPLNAME DS   CL10
EMPLNUM  DS   CL3
BACKEMPN DS   CL3
REGION   DS   CL5
AMOUNT   DS   ZL6'9999.99'
TAX      DS   ZL4'99.99'
COMMRATE DS   ZL4'9.999'
SALEDATE DS   CL6
SALETIME DS   CL6
CUSTOMER DS   CL15
TELEPHON DS   ZL10
TIMEPHON DS   ZL4'999.9'
PRODCODE DS   CL3
         DS   CL1
INPUT:   SALES-FILE
COLUMNS: EMPLNAME
         SALEDATE
         SALETIME
         CUSTOMER
         AMOUNT
         TAX
```

Produce this Report:

```
TUE  02/14/95    8:27 AM  DATA FROM SALES-FILE              PAGE     1

 EMPLNAME  SALEDATE SALETIME   CUSTOMER       AMOUNT      TAX

JOHNSON     950312   102500  ACE ELECTRICAL      101.38      6.09
BAKER       950326   120909  JACKS CAFE          137.00      8.22
MORRISON    950329   153022  STAR MARKET          44.35      2.66
MORRISON    950330   190541  A1 PHOTOGRAPHY       29.65      1.78
SIMPSON     950401   081757  EUROPEAN DELI        14.99      0.90
JOHNSON     950401   170247  VILLA HOTEL         234.45     14.07
JOHNSON     950405   143310  MARYS ANTIQUES        9.98      0.60
BAKER       950412   143112  JACKS CAFE          135.75      8.15
THOMAS      950414   154138  YOGURT CITY           9.98      0.60
JONES       950415   075832  EZ GROCERY           10.25      0.62
JONES       950415   080159  TOY TOWN            121.76      7.31
JONES       950415   135241  TOY TOWN             10.25      0.62
JOHNSON     950416   114833  ACME BUILDING       500.00     30.00
SIMPSON     950430   153021  J & S LUMBER         23.87      1.43


*** GRAND TOTAL (14 ITEMS)                    1,383.66     83.05
```

Figure 60. A report produced using an Assembler record layout

The ASM statement tells Spectrum Writer that subsequent control statements will be in the IBM S/370 Assembler language. After the ASM statement, actual lines of an Assembler record layout appear. The Assembler code must be error-free and must be formatted according to the rules of Assembler syntax. (That is, labels must begin in column 1, column 72 is the continuation column, etc.) Spectrum Writer processes the Assembler record layout, noting the names of the Assembler fields and their characteristics. Internally, Spectrum Writer creates the equivalent of a FIELD statement for each Assembler field in the record layout.

> **Note:** See "Technical Notes on Assembler Support" (page 387) for certain limitations on the Assembler syntax that Spectrum Writer accepts.

Spectrum Writer continues treating each subsequent line as Assembler code until it reaches a line that begins with a Spectrum Writer control statement prefix. In this example, the line beginning "INPUT:" is recognized as a Spectrum Writer control statement. So, starting with the INPUT statement the lines are no longer treated as Assembler code. (The scope of the ASM statement is discussed more fully on page 384.)

After the Assembler record layout, we simply resumed the report request in the normal way. The INPUT statement specifies the input file for the report. It is the SALES–FILE that we just defined using the Assembler record layout.

The COLUMNS statement specifies which fields are wanted in the report. In the COLUMNS statement we can refer to any of the fields defined in the Assembler record layout. The field names used are the same names that appeared in the Assembler layout. By default, the column headings will also be the Assembler field name. Of course, you can specify an override column heading, if you like.

Fields defined by an Assembler layout can be used in all of the same way as fields defined with FIELD statements. You can use the Assembler field names in SORT statements, COMPUTE statements, BREAK statements, and so on.

> **Note:** By default, the Assembler record layout defines fields for the most recently defined file. Use a FILE parm in the ASM statement if the fields belong to some other file (see page 495).

> **Note:** To see a listing of the internal FIELD statements that Spectrum Writer creates from an Assembler layout, add the **SHOWFLDS(YES)** parm to the ASM statement:

```
ASM: SHOWFLDS(YES)
```

> If we had used the above statement in the report in **Figure 60**, the following FIELD statements would have appeared in the control listing:

```
FIELD: SALESREC   LEN(80) COL(1)
FIELD: EMPLNAME   LEN(10) COL(1)
FIELD: EMPLNUM    LEN(3)
FIELD: BACKEMPN   LEN(3)
FIELD: REGION     LEN(5)
FIELD: AMOUNT     LEN(6)  TYPE(NUM-SLD)  DEC(2)
FIELD: TAX        LEN(4)  TYPE(NUM-SLD)  DEC(2)
FIELD: COMMRATE   LEN(4)  TYPE(NUM-SLD)  DEC(3)
FIELD: SALEDATE   LEN(6)
FIELD: SALETIME   LEN(6)
FIELD: CUSTOMER   LEN(15)
FIELD: TELEPHON   LEN(10) TYPE(NUM-SLD)
FIELD: TIMEPHON   LEN(4)  TYPE(NUM-SLD)  DEC(1)
FIELD: PRODCODE   LEN(3)
```

Handling Date and Time Fields in Record Layouts

Neither Cobol nor Assembly language have a way to explicitly define a field as a date or a time. Date and time fields are generally defined as numeric fields (or sometimes as character fields) in these languages. It is left up to the program code in those languages to know that the numeric value actually represents a date or a time.

For example, consider the SALES–DATE field in the Cobol example on page 371. The file actually contains a YYMMDD date for this field. But it is defined in Cobol simply as PIC 9(6). Spectrum Writer has no way of knowing that this field is anything other than a 6-digit numeric field. In the report, therefore, it doesn't appear in MM/DD/YY format as a date field normally would. It is treated as a numeric field. By default it would have appeared in "ZZZ,ZZ9" format in the report (for example: 920,415). Also, by default, that column would have been totalled at the end of the report (like all other numeric columns). To make the value look more like a date, we used override parms in the COLUMNS statement to change the display format to PIC'999999' and to suppress the totals.

The SALES–TIME field has the same problem. The field actually contains a HHMMSS time. But since it is defined in Cobol as PIC 9(6), it's just an ordinary numeric field to Spectrum Writer. Again, we used override parms in the COLUMNS statement to improve its appearance in the report.

However, there is a simple way to use Cobol and Assembler record layouts and *still* be able to define fields as true date or time fields. One extra step is all that's needed. Consider the example in **Figure 61**. In this example, we created a true date field simply by adding this statement after the Cobol record layout:

```
FIELD: SALES-DT  COLUMN(SALES-DATE) TYPE(YYMMDD)
```

This statement creates a new field named SALES–DT. The field starts in the same column as SALES–DATE, but has a data type of YYMMDD. Therefore, SALES–DT is a true date field. That means that it is formatted like a date in the report (MM/DD/YY). It also means that date literals can be used when comparing it in a conditional expression (for example, SALES–DT >= 12/31/1996).

By referring back to SALES–DATE in the COLUMN parm, we don't have to calculate the exact column that the field starts in. The SALES-DT field just starts in the same column as the SALES–DATE field. And, if the record layout is later changed and SALES–DATE moves to a different column, the FIELD statement for SALES–DT will still be correct.

We used the same technique to define a true time field:

```
FIELD: START-TM  COLUMN(START-TIME)  TYPE(HHMMSS)
```

START–TM is a time field that starts in the same column as the numeric field START–TIME. By using START–TM in the report, the data is formatted as a time (HH:MM:SS). And time literals can be used when comparing it in a conditional expression (for example, START–TM < 12:00:00).

The bottom box in **Figure 61** shows the report created using these true date and time fields. As you can see, the SALES–DT and SALES–TM fields are now formatted correctly. In this example, we no longer needed override parms in the COLUMNS statement.

You can use this same technique for any kind of date or time field. For example, assume that a file contains a Cobol field named JULIAN–DATE defined as PIC S9(5) COMP-3. Spectrum

These Control Statements:

```
FILE: SALES-FILE  DDNAME(SALEFILE)
COBOL:
       01    SALES-REC.
             05 EMPL-NAME        PIC X(10).
             05 EMPL-NUM         PIC X(3).
             05 BACKUP-EMPL-NUM  PIC X(3).
             05 REGION           PIC X(5).
             05 AMOUNT           PIC 9999V99.
             05 TAX              PIC 99V99.
             05 COMMISSION-RATE  PIC 9V999.
             05 SALES-DATE       PIC 9(6).
             05 SALES-TIME       PIC 9(6).
             05 CUSTOMER         PIC X(15).
             05 TELEPHONE        PIC 9(10).
             05 TIME-ON-PHONE    PIC 999V9.
             05 PRODUCT-CODE     PIC X(3).
             05 FILLER           PIC X(1).
FIELD: SALES-DT  COLUMN(SALES-DATE)  TYPE(YYMMDD)
FIELD: SALES-TM  COLUMN(SALES-TIME)  TYPE(HHMMSS)
INPUT:   SALES-FILE
COLUMNS: EMPL-NAME
         SALES-DT
         SALES-TM
         CUSTOMER
         AMOUNT
         TAX
```

Produce this Report:

```
TUE  02/14/95    8:27 AM  DATA FROM SALES-FILE              PAGE     1

   EMPL     SALES    SALES
   NAME      DT       TM      CUSTOMER        AMOUNT        TAX

JONES      04/15/92 13:15:00 TOY TOWN              10.25       0.62
JONES      04/15/92 12:43:56 TOY TOWN             121.76       7.31
JONES      04/15/92 10:30:32 EZ GROCERY            10.25       0.62
JOHNSON    04/01/92 16:48:59 VILLA HOTEL          234.45      14.07
JOHNSON    04/05/92 14:00:41 MARYS ANTIQUES         9.98       0.60
JOHNSON    03/12/92 08:05:02 ACE ELECTRICAL       101.38       6.09
JOHNSON    04/16/92 09:50:41 ACME BUILDING        500.00      30.00
SIMPSON    04/30/92 11:59:59 J & S LUMBER          23.87       1.43
MORRISON   03/29/92 12:40:11 STAR MARKET           44.35       2.66
MORRISON   03/30/92 15:00:02 A1 PHOTOGRAPHY        29.65       1.78
SIMPSON    04/01/92 17:01:38 EUROPEAN DELI         14.99       0.90
BAKER      03/26/92 17:01:29 JACKS CAFE           137.00       8.22
BAKER      04/12/92 16:00:00 JACKS CAFE           135.75       8.15
THOMAS     04/14/92 07:56:00 YOGURT CITY            9.98       0.60

*** GRAND TOTAL (14 ITEMS)                      1,383.66      83.05
```

Figure 61. Creating true date and time fields from a Cobol layout

Writer would treat this field like any other 5–digit packed number. But you could create a true Spectrum Writer date field by adding the following statement:

```
FIELD: JULIAN-DT  COLUMN(JULIAN-DATE)  TYPE(P-YYDDD)
```

JULIAN–DT will be a true date field (in packed Julian format). It is defined as starting in the same column as the numeric field JULIAN–DATE.

To avoid adding the extra FIELD statements in each run, you may want to create a copy library member that contains these extra FIELD statements along with your Cobol record layout. Such a member would include everything you see in the top box in **Figure 61** up until the INPUT statement. (That is, it would contain: a FILE statement; a COBOL (or ASM) statement; the record layout; and the additional FIELD statements for the date and time fields.)

This copy member could then be copied automatically whenever it is needed, just like regular Spectrum Writer file definitions are. For example, you could then request a report in the following manner:

```
INPUT:     SALES-FILE
COLUMNS:   CUSTOMER  SALES-DT  SALES-TM
INCLUDEIF: SALES-DT > 1/1/1995 AND SALES-TM > 12:00:00
```

In other words, you could request reports and PC files from the SALES–FILE just as easily as you do with any other file. The only difference is that the SALES–FILE would now be defined primarily via a Cobol record layout, rather than FIELD statements.

How Spectrum Writer Handles Cobol Arrays

Spectrum Writer requires that every field have a unique name. Therefore, when Spectrum Writer encounters a Cobol field with an OCCURS clause, it creates a separate field for each occurrence of the item. Spectrum Writer makes these field names unique by appending a numeric suffix to the end of the name. For example, consider the following Cobol statement with an OCCURS clause:

```
05 ADDR-LINE OCCURS 3 TIMES PIC X(30).
```

Spectrum Writer would create the following three internal FIELD statements as a result of the above statement:

```
FIELD: ADDR-LINE-1  LEN(30)
FIELD: ADDR-LINE-2  LEN(30)
FIELD: ADDR-LINE-3  LEN(30)
```

You would use the above field names in your report request (rather than ADDR–LINE alone). For example, to include the second occurrence of the ADDR-LINE array in your report, you would specify:

```
COLUMNS: ADDR-LINE-2
```

Spectrum Writer does the same thing for Assembler fields that have a repetition factor. Consider the following Assembler statement that includes a repetition factor:

```
FLAGS    DS    4CL1
```

Spectrum Writer would create the following four internal FIELD statements as a result of the above statement:

```
FIELD: FLAGS-1  LEN(1)
FIELD: FLAGS-2  LEN(1)
FIELD: FLAGS-3  LEN(1)
FIELD: FLAGS-4  LEN(1)
```

Spectrum Writer also supports nested arrays in Cobol. Spectrum Writer assigns one numeric suffix for each level of the array. The first suffix refers to the outer array, the second suffix refers to the inner array. (The suffixes work in the same way as, and appear in the same order as, Cobol subscripts.) For example, consider the following Cobol statements:

```
05 ADDRESS-ARRAY OCCURS 2 TIMES.
   10 ADDR-LINE OCCURS 3 TIMES PIC X(30).
```

Spectrum Writer would create the following internal FIELD statements as a result:

```
FIELD: ADDRESS-ARRAY-1 LEN(90)
FIELD: ADDRESS-ARRAY-2 LEN(90)
FIELD: ADDR-LINE-1-1   LEN(30) COL(1)
FIELD: ADDR-LINE-1-2   LEN(30)
FIELD: ADDR-LINE-1-3   LEN(30)
FIELD: ADDR-LINE-2-1   LEN(30)
FIELD: ADDR-LINE-2-2   LEN(30)
FIELD: ADDR-LINE-2-3   LEN(30)
```

If you're not sure what suffix Spectrum Writer has assigned, use the SHOWFLDS(YES) parm in your COBOL or ASM statement. That way you will see a complete listing of the internal FIELD statements that Spectrum Writer has created from your record layout.

Note: By default, Spectrum Writer creates internal FIELD statements for up to 100 occurrences of any item that has an OCCURS clause (or a repetition factor). This is to avoid wasting memory for items that may not actually be needed in the report run. If you want a higher (or lower) limit on the number of occurrences that will be individually defined, use the MAXOCCURS parm in your COBOL or ASM statement. (See page 495.) Note that even when all occurrences of a field are not individually defined, the record layout is still processed correctly. That is, items appearing after the array will still be defined in their correct locations.

Note: For Cobol items defined with the OCCURS DEPENDING ON clause, Spectrum Writer creates fields for the maximum possible number of occurrences (subject to the MAXOCCURS limit just described).

Note: See "Working With Arrays" (page 237) for more information on processing arrays with Spectrum Writer.

Converting Cobol and Assembler Layouts to FIELD Statements

Until now we have looked at examples of "live" runs. That is, runs where you provide a Cobol or Assembler layout to Spectrum Writer and then request a report in the same run. This is very convenient for occasions when you need to quickly produce a custom report from a file that you've never used with Spectrum Writer before.

However, for input files that will be used often with Spectrum Writer, it may be better to create a standard Spectrum Writer file definition (consisting of a FILE statement and multiple FIELD statements). This allows you to use features available in the FIELD statement that aren't available in Cobol or Assembler layouts. For example, in the FIELD statement you can specify your own default column headings. You can also specify special display formats that should be used with certain fields (for example, telephone numbers). Using FIELD statements also lets you define true date and time fields, which are not directly supported in either Cobol or Assembler.

But rather than create the FIELD statements by hand, you can use Spectrum Writer to perform a one–time conversion of your Cobol or Assembler layout into FIELD statements. Spectrum Writer does all of the hard work for you — it calculates the starting columns for each field, it figures out the length of packed items based on their PICTURE clause, it handles REDEFINES clauses, OCCURS clauses, etc. Use the resulting FIELD statements as your starting point. Then go through them and make whatever modifications you desire. The result will be a standard Spectrum Writer file definition, but without all the manual work normally involved in writing FIELD statements by hand.

How do you perform such a one–time conversion? You've seen that by using the SHOWFLDS(YES) parm in the COBOL (or ASM) statement, you can get a listing of FIELD statements that correspond to the Cobol (or Assembler) record layout. This listing appears imbedded in the normal control statement listing. By using a different parm, you can have Spectrum Writer write those same FIELD statements to a separate output file. **Figure 62** shows an example of converting a Cobol record layout to FIELD statements.

Notice the control statements in the top box in **Figure 62**. Once again, a FILE statement is required because fields must always be defined for a specific file. Spectrum Writer won't process record layouts or FIELD statements unless it has a file that it can associate those fields with. In this case, the file name specified isn't important (since no report will be produced from the file in this run). Use any file name you like in the FILE statement.

The COBOL statement tells Spectrum Writer to expect a Cobol record layout to follow. In this case, we used an additional parm in the COBOL statement. The **OUTDDN parm** tells Spectrum Writer the name of a DD statement in the JCL where the FIELD statements should be written. In this example, we told Spectrum Writer to write the FIELD statements to a DD named FLDOUT. The file named in this DD statement must have a record length of 80 bytes.

> **VSE Note:** Use the **OUTATTR parm**, rather than the OUTDDN parm, in the COBOL or ASM statement. The complete syntax of the OUTATTR parm is shown on page 496. Here is a typical example of a COBOL statement with an OUTATTR parm:
>
> ```
> COBOL:  OUTATTR(DASD,'FLDOUT')
> ```
>
> The above statement causes the FIELD statements to be written to a SAM output file on disk. It is identified in the JCL by a DLBL named FLDOUT. The file will be written as single blocked, 80-byte records.

Spectrum Writer examines the Cobol record layout and writes one FIELD statement to the output file for each field present in the Cobol layout.

Since we did not want to produce an actual report in this run, we did not follow the Cobol record layout with an INPUT statement or any other Spectrum Writer statements. Spectrum Writer writes the FIELD statements to the output file, and then ends execution. (You may see a message saying that no report was produced because no INPUT statement was found.

These Control Statements:

```
FILE: DUMMY
COBOL: OUTDDN(FLDOUT)
       01    SALES-REC.
             05 EMPL-NAME        PIC X(10).
             05 EMPL-NUM         PIC X(3).
             05 BACKUP-EMPL-NUM  PIC X(3).
             05 REGION           PIC X(5).
             05 AMOUNT           PIC 9999V99.
             05 TAX              PIC 99V99.
             05 COMMISSION-RATE  PIC 9V999.
             05 SALES-DATE       PIC 9(6).
             05 SALES-TIME       PIC 9(6).
             05 CUSTOMER         PIC X(15).
             05 TELEPHONE        PIC 9(10).
             05 TIME-ON-PHONE    PIC 999V9.
             05 PRODUCT-CODE     PIC X(3).
             05 FILLER           PIC X(1).
```

Write These FIELD Statements to a Special Output File:

```
FIELD: SALES-REC       LEN(80) COL(1)
FIELD: EMPL-NAME       LEN(10) COL(1)
FIELD: EMPL-NUM        LEN(3)
FIELD: BACKUP-EMPL-NUM LEN(3)
FIELD: REGION          LEN(5)
FIELD: AMOUNT          LEN(6)   TYPE(NUM) DEC(2)
FIELD: TAX             LEN(4)   TYPE(NUM) DEC(2)
FIELD: COMMISSION-RATE LEN(4)   TYPE(NUM) DEC(3)
FIELD: SALES-DATE      LEN(6)   TYPE(NUM)
FIELD: SALES-TIME      LEN(6)   TYPE(NUM)
FIELD: CUSTOMER        LEN(15)
FIELD: TELEPHONE       LEN(10)  TYPE(NUM)
FIELD: TIME-ON-PHONE   LEN(4)   TYPE(NUM) DEC(1)
FIELD: PRODUCT-CODE    LEN(3)
FIELD: FILLER#001      LEN(1)
```

File Definition After Typical Customization:

```
FILE:  SALES-FILE  DDNAME(SALEFILE)
FIELD: SALES-REC       LEN(80) COL(1)
FIELD: EMPL-NAME       LEN(10) COL(1)   HEADING('EMPLOYEE NAME')
FIELD: EMPL-NUM        LEN(3)
FIELD: BACKUP-EMPL-NUM LEN(3)
FIELD: REGION          LEN(5)
FIELD: AMOUNT          LEN(6)  TYPE(NUM)     DEC(2)
FIELD: TAX             LEN(4)  TYPE(NUM)     DEC(2)
FIELD: COMMISSION-RATE LEN(4)  TYPE(NUM)     DEC(3)
FIELD: SALES-DATE      LEN(6)  TYPE(YYMMDD)
FIELD: SALES-TIME      LEN(6)  TYPE(HHMMSS)
FIELD: CUSTOMER        LEN(15)
FIELD: TELEPHONE       LEN(10) TYPE(NUM)  FORMAT(PIC'(999) 999-9999')   NOACCUM
FIELD: TIME-ON-PHONE   LEN(4)  TYPE(NUM) DEC(1)
FIELD: PRODUCT-CODE    LEN(3)
FIELD: FILLER#001      LEN(5)
```

Figure 62. Converting a Cobol record layout to Spectrum Writer FIELD statements

This is normal.) The middle box in **Figure 62** shows the FIELD statements produced by Spectrum Writer.

Having created the FIELD statements automatically, you can now modify them as desired. For example, you could add HEADING parms or FORMAT parms to specify column headings and display formats for any or all of the fields. The bottom box in **Figure 62** shows an example of how the FIELD statements might be modified. In this example, we added a HEADING parm for EMPL–NAME. And, we changed the TYPE parm in the SALES–DATE field from NUM to YYMMDD. Now SALES–DATE is defined as a true date field. We also made SALES–TIME a true time field by changing its TYPE parm to HHMMSS. We added a FORMAT parm and the NOACCUM parm to the FIELD statement for TELEPHONE. That prevents the telephone number from being accumulated (totalled) and causes it to be formatted attractively.

If the Cobol field names in your record layout are long and cumbersome, you might also want to perform some global changes on the names themselves. For example, if all fields in your Cobol layout began with a prefix (like "SALES–REC–EMPL–NAME", "SALES–REC–EMPL–NUM", etc.) you might want to perform a global edit to drop the common prefix ("SALES–REC–") from the field names.

> **Note:** When modifying the FIELD statements, be careful not to make any change that would affect subsequent FIELD statements. For example, changing the length of a field might cause the following field to start in the wrong column. Also be careful about removing FIELD statements or changing their order.

You should also add an appropriate FILE statement ahead of the FIELD statements. When you're satisfied with your file definition, save it in your Spectrum Writer Copy Library. You can then produce reports and PC files using this file definition in the normal manner. You will not need to use the Cobol record layout in subsequent runs, because you now have a standard Spectrum Writer file definition for your file.

> **Note:** The example discussed above used a Cobol record layout. You can also create FIELD statements from an Assembler layout in the same way. Just use the OUTDDN parm (or OUTATTR parm) in your ASM statement.

There are some additional parms available for use with the COBOL and ASM statements.

- the **RELOC parm** (page 497) which specifies that the FIELD statements that are written out should be "relocatable" whenever possible. This option may make it easier for you to modify your Spectrum Writer file definition when a record layout changes
- the **COLUMN** and **DISP parms** (page 494), which let you specify which of those parms should be used in the FIELD statements created by Spectrum Writer. These parms can also specify that *all* FIELD statements should have either a COLUMN or DISP parm. (Otherwise, only fields that are defined out of sequence will have such a parm.)

How to Copy Cobol and Assembler Record Layouts from Libraries

Our examples until now have used Cobol and Assembler record layouts written "in line." That is, they have been imbedded directly among the Spectrum Writer control statements. But normally Cobol and Assembler record layouts are stored as members in copy libraries, to be accessed by their respective compilers. Spectrum Writer also allows you to copy such record layouts directly from those libraries. Just use Spectrum Writer's COPY statement wherever you want the Cobol or Assembler lines to be included. For example:

```
FILE:  SALES-FILE  DDNAME(SALEFILE)  LRECL(80)
COBOL:
COPY:  SALEREC
INPUT: SALES-FILE
...
```

In this example, a Cobol record layout still follows the COBOL statement. But this time it's copied from a member named SALEREC in a copy library.

We mentioned in an earlier section that Cobol processing begins immediately after the COBOL statement and ends when the next Spectrum Writer control statement is encountered. The COPY statement is an exception to this rule. A COPY statement does *not* signal the end of the Cobol (or Assembler) code. This allows you to embed COPY statements within sections of Cobol or Assembler code.

You may use multiple COPY statements. You may also intermix in–line Cobol code with Cobol code copied via COPY statements. For example, the following is valid:

```
FILE:  SALES-FILE  DDNAME(SALEFILE)  LRECL(80)
COBOL:
       01 REC-A.
COPY:  SALERECA
       01 REC-B REDEFINES REC-A.
COPY:  SALERECB
INPUT: SALES-FILE
...
```

The following sections explain **which library** the member will be copied from.

OS/390 COPY Statement

Normally the COPY statement is used to read a member from the Spectrum Writer Copy Library — the one pointed to by the SWCOPY DD in the JCL. However, Cobol and Assembler record layouts are generally kept in different libraries from your Spectrum Writer definitions (and even in different libraries from each other). Therefore, when processing Cobol code, Spectrum Writer copies members from the PDS named in the COBLIB DD in the JCL, if one is present. When processing Assembler record layouts, members are copied from the PDS named in the ASMLIB DD, if one is present. If the appropriate DD (COBLIB or ASMLIB) is not present, Spectrum Writer then attempts to perform the copy from the standard copy library (SWCOPY DD).

If you prefer, you can override these defaults and used any DDNAME you like to identify the PDS to copy from. Just add a PDSDDN parm to the COPY statement:

```
COPY: SALEREC  PDSDDN('COPYLIB')
```

The above statement would cause the member named SALEREC to be copied from the PDS identified by the COPYLIB DD in the JCL. The PDSDDN parm is useful if you need to perform multiple copies in a run and each copy must come from a different library.

You can also use the COPY statement to copy a "flat" sequential file. This may be necessary if your shop stores copy members in a proprietary library, such as PANVALET or LIBRARIAN. Add a utility step to your jobstream that writes the desired member out to a sequential dataset. Then in the Spectrum Writer step, you can copy that sequential dataset by using the DDNAME parm in the COPY statement:

```
COPY: DDNAME(SALEREC)
```

The above example causes Spectrum Writer to read in the records from the sequential file pointed to by the SALEREC DD in the JCL.

VSE COPY Statement

Normally the COPY statement is used to read a member from the Spectrum Writer Copy Library — the sublibrary named in an OPTIONS statement SUBLIB parm. However, Cobol and Assembler record layouts are generally kept in different sublibraries from your Spectrum Writer definitions. Therefore, when processing Cobol code, Spectrum Writer copies members from the sublibrary named in an OPTIONS statement COBLIB parm, if one has been specified. When processing Assembler record layouts, members are copied from the sublibrary named in an OPTIONS statement ASMLIB parm, if one has been specified. If the appropriate option (COBLIB or ASMLIB) was not specified, Spectrum Writer then attempts to copy the member from the standard copy library (identified by an OPTIONS statement SUBLIB parm).

If you prefer, you can override these defaults and copy the member from any sublibrary that you like. Just add a SUBLIB parm to the COPY statement:

```
COPY: SALEREC  SUBLIB('COPYLIB.PROD')
```

The above statement would cause the member named SALEREC to be copied from the sublibrary named "COPYLIB.PROD." The SUBLIB parm is useful if you need to perform multiple copies in a run and each copy comes from a different sublibrary.

The **member type** used for COPY statements within the scope of a COBOL statement is "C." Within the scope of an ASM statement, member type "A" is used. You can also override these defaults by specifying a member type directly in the COPY statement, like this:

```
COPY: SALEREC.COB
```

Mixing FIELD Statements with COBOL and ASM Statements

You may use any combination of FIELD statements, COBOL statements and ASM statements to define an input file. For example, the following is valid:

```
FILE:  SALES-FILE  DDNAME(SALEFILE)
COBOL:
       01 REC-A.
COPY:  SALEREC1
ASM:   STARTCOL(1)
RECB     DS    0CL80
COPY:  SALEREC2
FIELD: REC-C COLUMN(1) LENGTH(80)
INPUT: SALES-FILE
...
```

The above example uses a Cobol record layout to define the fields in one type of record for the SALES–FILE. It then uses an Assembler record layout to define the fields in a second type

of record for the file. Note the STARTCOL(1) parm in the ASM statement causes the first field from the Assembler code to begin in column 1 (rather than picking up after the last field defined by the Cobol record layout). Lastly, an explicit FIELD statement defines a field called REC–C. The COLUMN parm causes it to start in column 1 also.

The Starting Column of a Cobol or Assembler Layout

By default, Spectrum Writer defines the first item within a Cobol record layout at the file's "default location." Thus, if you have no explicit FIELD statements before your Cobol record layout, the first item in the Cobol layout will be defined as beginning in column 1 of the record. If you do have preceding FIELD statements (or preceding Cobol or Assembler record layouts), the first item in the Cobol record layout will begin in the column immediately following the last field defined. Use the **STARTCOL** or **STARTDISP parm** (in the COBOL statement) if you want the fields from the Cobol record layout to begin in some other column.

The location of the first field in an Assembler record layout is handled the same way.

The "Default Location" After a Cobol or Assembler Layout

Spectrum Writer updates a file's "default location" pointer while processing Cobol (and Assembler) layouts just as it does when processing FIELD statements. Thus, the default location after processing a Cobol layout is immediately after the last field within the layout. Any FIELD statements appearing after the Cobol layout which do not contain a COLUMN or DISP parm would be defined as starting immediately after the last field from the Cobol layout. Similarly, if you use a second COBOL statement, the first item in that record layout would immediately follow the last field from the previous Cobol layout (unless you override this with a STARTCOL or STARTDISP parm in the second COBOL statement).

> **Caution:** If your Cobol code contains multiple 01–level record layouts, remember that the last field present in the record layout may *not* be the field that actually occupies the last bytes within the record. This happens when a shorter record layout redefines a larger record layout. In that case, the default location counter would be immediately after the last field from the second, shorter record layout — not after the last field in the larger record layout. The same thing is possible within a single record layout if it ends after an explicit REDEFINES of a larger object. Within Assembler code, a similar situation arises when a smaller DSECT follows a larger DSECT.

The Scope of the COBOL and ASM Statements

Beginning immediately after a COBOL statement, Spectrum Writer treats input lines as Cobol code. (However, the COBOL statement itself may be continued onto multiple lines if necessary.) After the complete COBOL statement, subsequent lines, including lines copied via a COPY statement, are treated as Cobol code. The Cobol code is assumed to end when

the first Spectrum Writer control statement prefix is encountered. There are, however, two exceptions to this rule.

1. A COPY statement does not end the scope of the COBOL statement. This lets you use the COPY statement to include additional lines of Cobol code from a library. (Of course, if one of the *copied lines* contains a Spectrum Writer control statement, that line will end the scope of the COBOL statement.)
2. A Spectrum Writer comment line does not end the scope of the COBOL statement. Thus, a line beginning with an asterisk in column 1 would be treated as Cobol code and not as a comment line.

The scope of the ASM statement is the same as described above for COBOL.

If you have any question whether Spectrum Writer is treating a particular input line as Cobol, Assembler, or native Spectrum Writer, just check the control listing output. The word "COBOL" or "ASM" will appear beside each line that Spectrum Writer is interpreting as Cobol or Assembler code. Use the LIST(YES) option on any COPY statements to ensure that the copied lines also appear in the control listing.

Technical Notes on Cobol Support

Spectrum Writer will accept the vast majority of Cobol record layouts used in most shops. Still, Spectrum Writer is not a complete Cobol compiler and there are some valid Cobol features that Spectrum Writer does not support at present.

Even when Spectrum Writer doesn't support a particular Cobol statement, you will still save much time by using the FIELD statement output from Spectrum Writer as the beginning point of your file definition. Many FIELD statements will be correct, and you can modify any incorrect ones as needed.

It is important that the Cobol record layout be completely error free. Spectrum Writer does not attempt to perform all of the functions of the Cobol compiler, and may not notify you of syntax errors. Do not try to *develop* your Cobol record layouts with Spectrum Writer. Use the Cobol compiler for that purpose and use only clean, tested record layouts in Spectrum Writer.

In general, if a record layout would be accepted in the Record Description entry of an FD (File Description), Spectrum Writer will also accept it. In addition, Spectrum Writer accepts many types of edited PICTURES (like PIC $$$,$$9.99). This means that Spectrum Writer can support many report line structures taken from Cobol report programs. This is useful when a report written by a Cobol program will be used as input to Spectrum Writer.

Level Indicators

Spectrum Writer supports level indicators between 01 and 49. Level 77 is not allowed. Levels 66 and 88 are ignored but do not interfere with the correct interpretation of the other statements.

REDEFINES Clauses

If an item contains a REDEFINES clause, both the item and the object of its REDEFINES clause must be within the scope of the same COBOL control statement. That is, an item within the scope of one COBOL statement may not redefine an item within the scope of an earlier COBOL control statement.

01–Level Implicit Redefines

As with Cobol in a FD clause, Spectrum Writer treats each 01 level item as an implicit redefine of the entire record. Items beginning with the 01 level are assumed to begin in the same column as the first field following the COBOL control statement.

Unique Field Names

In Cobol, different records may contain fields with the same name. You use the "field OF qualifier" notation in Cobol to avoid any ambiguity. Spectrum Writer requires unique field names for each field within a file. Therefore, if you copy multiple record layouts and the same field name is used more than once, Spectrum Writer makes the second field name unique by appending a "tiebreaker" to it. The tiebreaker has the format "#nnn". For example, if the Cobol layout(s) you use contain two fields with the name DATE, Spectrum Writer would use DATE for the first item and DATE#001 for the second item. A message is printed in the control listing whenever Spectrum Writer modifies a name in this way to make it unique.

Handling FILLER

Spectrum Writer just defines all FILLER items as character fields. However, Spectrum Writer *always* appends a tiebreaker to FILLER fields. No message is printed when this happens (since it is so common), but you can see the actual name of all fields, including FILLER fields, by using the SHOWFLDS(YES) parm on the COBOL statement.

Handling 88–Level Items

Spectrum Writer does not process 88 level field definitions automatically. However, it is not difficult to create Spectrum Writer equivalents for 88 items yourself. Following is an example of how several 88 level items would be defined with Spectrum Writer.

For often-used 88 items, you may want to manually add such statements to your file definition. Consider these Cobol statements:

```
05 STATUS-CODE  PIC X(1).
   88 PART-TIME VALUE '1'.
   88 FULL-TIME VALUE '2', '4'.
   88 TERMINATED VALUE '5' THRU '9'.
```

In the example above, Spectrum Writer would create the 05 level field, STATUS–CODE, for you. It would then ignore the 88 level statements. To define the 88 fields to Spectrum Writer, you could add the following statements somewhere after the Cobol record layout.

```
COMPUTE: PART-TIME  = WHEN(STATUS-CODE = '1')          ASSIGN(#ON)
COMPUTE: FULL-TIME  = WHEN(STATUS-CODE = '2' OR '4')   ASSIGN(#ON)
COMPUTE: TERMINATED = WHEN(STATUS-CODE >= 5 AND <= '9') ASSIGN(#ON)
```

The above COMPUTE statements define bit–type fields which can be used in conditional expressions in Spectrum Writer statements just like they are used in Cobol. For example:

```
INCLUDEIF: FULL-TIME
```

The above statement would include all records where the FULL–TIME field was on. That would be all records whose STATUS–CODE field contained a 2 or a 4. Unlike Cobol, you can also *print* these bit fields with Spectrum Writer. For example:

```
COLUMNS: FULL-TIME
```

The above statement causes a column to appear in the report for the FULL–TIME field. The report column will contain (by default) the words FULL–TIME or NOT FULL–TIME for each input record.

SIGN IS SEPARATE Clause

Spectrum Writer supports the SIGN IS SEPARATE clause for elemental items, but not for group items.

Technical Notes on Assembler Support

Spectrum Writer will accept most of the Assembler record layouts used in a typical shop. Still, Spectrum Writer is not a complete assembler and there are some valid Assembler features that Spectrum Writer does not support at present.

Even when Spectrum Writer doesn't support a particular Assembler statement, you will still save much time by using the FIELD statement output from Spectrum Writer as the beginning point of your file definition. Many FIELD statements will be correct, and you can modify any incorrect ones as needed.

It is important that the Assembler record layout be completely error free. Spectrum Writer does not attempt to perform all of the functions of the assembler, and may not notify you of syntax errors. Do not try to *develop* your Assembler record layouts with Spectrum Writer. Use the assembler for that purpose and use only clean, tested record layouts in Spectrum Writer.

In general, Spectrum Writer supports the following Assembler statements:

- DS and DC statements
- EQU statements
- ORG statements
- DSECT statements

Character–Numeric Data

One problem with many Assembler record layouts is that they often use the "C" (Character) data type to define numeric fields. Consider the following Assembler statement:

```
AMOUNT   DS    CL6            SALES AMOUNT IN CENTS
```

Spectrum Writer can only treat this AMOUNT field as a 6-byte character field. There is nothing to tell Spectrum Writer that its value is actually numeric and that it contains 2 decimal digits.

There is a different way to define such fields in Assembler which allows Spectrum Writer to correctly interpret them. It is to use the "Z" (Zoned) data type, and to include a sample initial value that indicates the number of decimal digits that the data contains. Consider the following Assembler statement:

```
AMOUNT   DS    ZL6'9999.99'    SALES AMOUNT IN CENTS
```

Spectrum Writer would correctly interpret this field by creating the following FIELD statement:

```
FIELD: AMOUNT  LEN(6)  TYPE(NUM-SLD)  DEC(2)
```

You may want to consider this when creating future Assembler record layouts, if you wish to use them with Spectrum Writer.

Another way to handle this problem (without modifying your record layout) is to use a COMPUTE statement. For example, if AMOUNT is defined simply as CL6, you could still get a numeric field that has 2 decimal digits by adding this COMPUTE statement somewhere after your record layout:

```
COMPUTE: REAL-AMOUNT(2) = #MAKENUM(AMOUNT) / 100
```

The above statement uses the #MAKENUM built–in function to convert the 6–byte character value into a numeric value. It is then divided by 100 to get the correct number of decimal digits.

If you will be using a particular file often with Spectrum Writer, it may be better to create a standard Spectrum Writer file definition for it. Use Spectrum Writer to convert the record layout into FIELD statements. Then modify the FIELD statements as necessary to correctly define the numeric fields.

Decimal Digits

Spectrum Writer creates a DEC(n) parm whenever the Assembler DS or DC statement has an initial value that includes one or more decimal digits. Consider this DS statement for a packed field:

```
SALARY   DS    PL4            SALARY (WITH 2 DECIMAL DIGITS)
```

Spectrum Writer would have no information about decimal digits and would define it like this:

```
FIELD: SALARY LEN(4) TYPE(PACKED)
```

But if you used this statement:

```
SALARY   DS    PL4'12345.67'   SALARY (WITH 2 DECIMAL DIGITS)
```

then Spectrum Writer could correctly create the following FIELD statement:

```
FIELD: SALARY LEN(4) TYPE(PACKED) DEC(2)
```

Another way to handle decimal problems (without modifying your record layout) is to use a COMPUTE statement. For example, if SALARY is defined simply as PL4, you could still get a field that has 2 decimal digits by adding this COMPUTE statement somewhere after your record layout:

```
COMPUTE: REAL-SALARY(2) = SALARY / 100
```

Support for expressions

Spectrum Writer supports some, *but not all*, types of expressions allowed by the IBM assembler. The following kinds of Assembler "terms" are supported within expressions:

- previously defined symbols (that is, field names created as a result of earlier Assembler statements). The value of such symbols is their *displacement* within

the record. The symbol must have been defined within the scope of the same ASM statement.

- length constants (example: L'AMOUNT)
- numeric, character and hex literals (examples: 123, C'ABC', X'FFFF')

The following operations are supported *as long as they are not nested and require no implicit ranking of operation*:

- addition
- subtraction
- multiplication
- division (with the remainder being dropped)

Following are some examples of statements containing expressions that Spectrum Writer *does* support:

```
LABEL    DS    XL100
LABEL1   DS    XL(L'LABEL)
LABEL2   DS    XL(L'LABEL-50)
LABEL3   EQU   LABEL2,L'LABEL2
LABEL4   DS    XL(L'LABEL1+L'LABEL2+5)
LABEL5   EQU   LABEL+X'1C'+26+C'P',5
```

Restrictions on expressions

Spectrum Writer does not support complex expressions within an Assembler statement. It interprets only non–nested operations that are performed strictly in left–to–right order. Thus, the following expression *is not supported* because it involves a nested operation:

```
LABEL    EQU   A+(B*C)
```

Spectrum Writer prints a warning message when an expression like the one above is encountered.

You *may* however use one level of parentheses around an entire expression. Thus, the following expression is accepted:

```
LABEL    EQU   (X+Y-Z)
```

The following expression *is not supported* because it implicitly requires that the second operation (C*D) be performed before the first operation (B+C).

```
LABEL    EQU   B+C*D
```

Spectrum Writer prints a warning message when an expression like the one above is encountered. You can simplify such expressions, if desired, so that Spectrum Writer can support your record layout. For example, the above statement could be simplified by breaking it into 2 statements:

```
TEMP     EQU   C*D
LABEL    EQU   B+TEMP
```

The above statement *are* acceptable to Spectrum Writer.

Multiple Operands

Spectrum Writer does not support DS or DC statements with multiple operands. For example, neither of the following statements is supported:

```
TABLE    DC    AL4(1,2,3,4)
MESSAGE  DC    H'5',C'HELLO'
```

However, DC and DS statements with *repetition factors* are supported. Thus, the following statement is acceptable to Spectrum Writer:

```
TABLE    DS    4AL4
```

Handling EQUs

When Spectrum Writer encounters an EQU statement that contains a label, it defines a field based on the statement's operands. (If the EQU statement has no label, the EQU statement is ignored.) The first operand of the EQU statement must be a self–defining expression. The value of this expression is used as the displacement for the field. If the EQU statement has no length operand, a length of 1 is assumed. If the EQU statement has no data type operand, character data is assumed. The "default location" is *not changed* as a result of an EQU statement. Consider the following two EQU statements:

```
LASTNAME EQU   NAME+10,15
R15      EQU   15
```

The above example would result in two fields being defined. The LASTNAME field would begin 10 bytes after the start of the NAME field (which must have been previously defined). It is a character field that is 15 bytes long. The second field, R15, would be a 1–byte character field beginning at displacement 15 in the record.

Handling DSECTs

When Spectrum Writer encounters a DSECT statement, it does two things. Firstly, it resets the default location to the value it had at the start of the Assembler code. That would be column 1 if no other fields had been defined earlier for the file. Or, it would be the value specified in any STARTCOL or STARTDISP parm in the ASM statement. Secondly, if the DSECT statement has a label, Spectrum Writer defines a 1-byte character field whose name is the DSECT name.

Unique field names

Spectrum Writer requires unique field names for each field within a file. Therefore, if you copy multiple record layouts and the same field name is used more than once, Spectrum Writer makes the second field name unique by appending a "tiebreaker" to it. The tiebreaker has the format "#nnn". For example, if the Assembler code you use contains two fields with the name DATE, Spectrum Writer would use DATE for the first item and DATE#001 for the second item. A message is printed in the control listing whenever Spectrum Writer modifies a name in this way to make it unique.

Chapter 7. Working with Databases

Chapter Table of Contents

Chapter 7. Working with Databases 391

Using Spectrum Writer with DB2 Databases 392
- Using DB2 Data in Reports 393
- Using DB2 Data in PC Programs 395
- What Fields Are in Your DB2 Table? 397
- Using the WHERE Parm 397
- Using the ORDERBY Parm 399
- Using Multiple DB2 Tables 400
- Using Data from Three DB2 Tables 403
- WHERE Parm Syntax 405
- Customizing Your DB2 Fields 407
- Saving DB2 File Definitions 408
- DB2 Setup 409
- DB2 Restrictions 410

Chapter 7. Working with Databases

This version of Spectrum Writer MVS supports the following databases:

- DB2

Using Spectrum Writer with DB2 Databases

Spectrum Writer's DB2 Option lets you use DB2 data with Spectrum Writer exactly like you use other mainframe data. That means you can:

- produce attractive custom reports from DB2 tables in just minutes.
- turn DB2 data into PC files designed especially for PC spreadsheet, database and graphics programs.
- turn DB2 data into any custom file format you need for use on mainframes, Unix machines, database servers, etc.
- use DB2 data to create Web reports.

Spectrum Writer's DB2 Option has these features:

- no data dictionary is required when using DB2 data. You just use the standard DB2 names for your DB2 tables, views, and columns. *This means you can start using Spectrum Writer with all of your DB2 tables right away*.
- you can combine data from up to 15 different DB2 tables to create a single report or PC file.
- you can even mix DB2 data with data from non–DB2 files. For example, you might have a tape file as the primary input to a Spectrum Writer job. Using data from that file, you could read additional data from VSAM files and/or DB2 tables. Or, you could use a DB2 table as your primary input and use data from it to read from additional DB2 tables or VSAM files. The possibilities are endless.

It's easy to use DB2 data with Spectrum Writer. You use the same control statements that you already know, with just a few differences. In fact, the only statements affected by the DB2 Option are these:

- the OPTIONS statement
- the INPUT statement
- the READ statement (not required)
- the FILE statement (not required)

For most reports and PC files, you'll only use the OPTIONS and INPUT statements.

> **JCL Note:** When using DB2 tables with Spectrum Writer, be sure that the STEPLIB DD in the execution JCL points to the load module where DB2's run–time modules are located. An example of a DB2 run–time module is DSNTIAR.

In the following sections, we assume that you are already familiar with using Spectrum Writer to request reports and output files. These sections explain the few differences that you need to know in order to use DB2 data in Spectrum Writer.

Using DB2 Data in Reports

Let's begin by looking at an actual Spectrum Writer report that uses DB2 data. Notice the sample report in **Figure 63**. Two of the control statements in this example contain DB2–related information. They are the OPTIONS statement and the INPUT statement.

First notice the OPTIONS statement. You'll see that we used the DB2SUBSYS option. This option tells Spectrum Writer which DB2 subsystem to access. Many shops have multiple DB2 subsystems. For example, a shop might have a test subsystem and a production subsystem. This option tells Spectrum Writer which subsystem to access for a particular run.

In our example, we specified a DB2 subsystem named "DB2T." That's the test subsystem in our "imaginary" company.

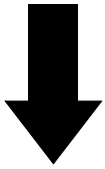
The DB2SUBSYS option is *required* when using DB2 data in a run. Remember to specify this option before your INPUT statement.

Next notice the INPUT statement. There are two names used in the INPUT statement:

- PROJECT, which is a user–assigned "Spectrum Writer name" for this input file. You can put any name here that you like. This name is not known to DB2 at all. In most runs, this name will never be referred to again. (However, in runs that use multiple input files, as you'll see later, "PROJECT" would be used to refer specifically to this input file.)
- DSN8230.PROJ, which is of course the actual name of the DB2 table. You can name a DB2 table *or* a DB2 view in this parm. By the way, DSN8230.PROJ is the name of a real "sample table" that is supplied by IBM with your DB2 system. Therefore, you can run this same job in your own shop for practice, if you like. This table contains information about various projects in an imaginary company.

These Control Statements:

```
OPTION:  DB2SUBSYS('DB2T')
INPUT:   PROJECT
         DB2NAME('DSN8230.PROJ')
TITLE:  'LISTING OF PROJECT DB2 TABLE'
COLUMNS: PROJNO
         PROJNAME
         DEPTNO
         RESPEMP
         PRSTDATE
         PRSTAFF
```



Produce this Report:

LISTING OF PROJECT DB2 TABLE

PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTDATE	PRSTAFF
AD3100	ADMIN SERVICES	D01	000010	01/01/82	6.50
AD3110	GENERAL AD SYSTEMS	D21	000070	01/01/82	6.00
AD3111	PAYROLL PROGRAMMING	D21	000230	01/01/82	2.00
AD3112	PERSONNEL PROGRAMMG	D21	000250	01/01/82	1.00
AD3113	ACCOUNT.PROGRAMMING	D21	000270	01/01/82	2.00
IF1000	QUERY SERVICES	C01	000030	01/01/82	2.00
IF2000	USER EDUCATION	C01	000030	01/01/82	1.00
MA2100	WELD LINE AUTOMATION	D01	000010	01/01/82	12.00
MA2110	W L PROGRAMMING	D11	000060	01/01/82	9.00
MA2111	W L PROGRAM DESIGN	D11	000220	01/01/82	2.00
MA2112	W L ROBOT DESIGN	D11	000150	01/01/82	3.00
MA2113	W L PROD CONT PROGS	D11	000160	02/15/82	3.00
OP1000	OPERATION SUPPORT	E01	000050	01/01/82	6.00
OP1010	OPERATION	E11	000090	01/01/82	5.00
OP2000	GEN SYSTEMS SERVICES	E01	000050	01/01/82	5.00
OP2010	SYSTEMS SUPPORT	E21	000100	01/01/82	4.00
OP2011	SCP SYSTEMS SUPPORT	E21	000320	01/01/82	1.00
OP2012	APPLICATIONS SUPPORT	E21	000330	01/01/82	1.00
OP2013	DB/DC SUPPORT	E21	000340	01/01/82	1.00
PL2100	WELD LINE PLANNING	B01	000020	01/01/82	1.00
*** GRAND TOTAL (20 ITEMS)					73.50

Figure 63. A Spectrum Writer DB2 report

Note: The sample IBM tables used in the following examples are named according to the particular release level of DB2. Thus, under Release 3.1 of DB2, for example, the Project table is named DSN8310.PROJ, rather than DSN8230.PROJ.

The INPUT statement does two things.

- it associates an actual DB2 table with a user–friendly Spectrum Writer "file name." (This association is not permanent— it lasts only during the one Spectrum Writer run.)
- it makes that DB2 table the primary input for your Spectrum Writer run.

These are the only required parms for an INPUT statement for a DB2 table. Subsequent sections of this chapter discuss other, optional, DB2-related parms for the INPUT statement. (The complete syntax for the INPUT statement appears on page 542.)

> **Terminology:** For the sake of consistency, we'll refer to the **DB2 table** named in an INPUT statement as an "input file," even though technically speaking it is not a "file". Similarly, we'll refer to DB2 **columns** as "DB2 fields" in this manual.

After your INPUT statement, you can use any of the other Spectrum Writer statements in any way you like. Refer to the DB2 fields by using their standard, unqualified DB2 names. Spectrum Writer will automatically recognize these DB2 names. For example, in the COLUMNS statement in **Figure 63** (page 394), we referred to the following DB2 fields from the project table: PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE and PRSTAFF.

You can also use the DB2 fields in the SORT statement, COMPUTE statements, INCLUDEIF statements, BREAK statements, and all the other Spectrum Writer statements. Just use the DB2 fields in exactly the same way as you would use the fields from a non–DB2 input file.

That's all there is to using DB2 data with Spectrum Writer! Here's a review of the differences from non–DB2 Spectrum Writer requests:

- no data definition of your DB2 file is necessary (that is, no FILE or FIELD statements are required)
- no Spectrum Writer Copy Library is required
- use an OPTIONS statement with the DB2SUBSYS parm (to identify the DB2 subsystem to use)
- use the DB2NAME parm in your INPUT statement (to identify the primary DB2 table to use)

> **Note:** Spectrum Writer supports character, numeric, date and time fields from DB2 tables. DB2 "timestamps" are treated as 26–byte character fields by Spectrum Writer. DB2 "graphic strings" and "floating point" numbers are not supported.

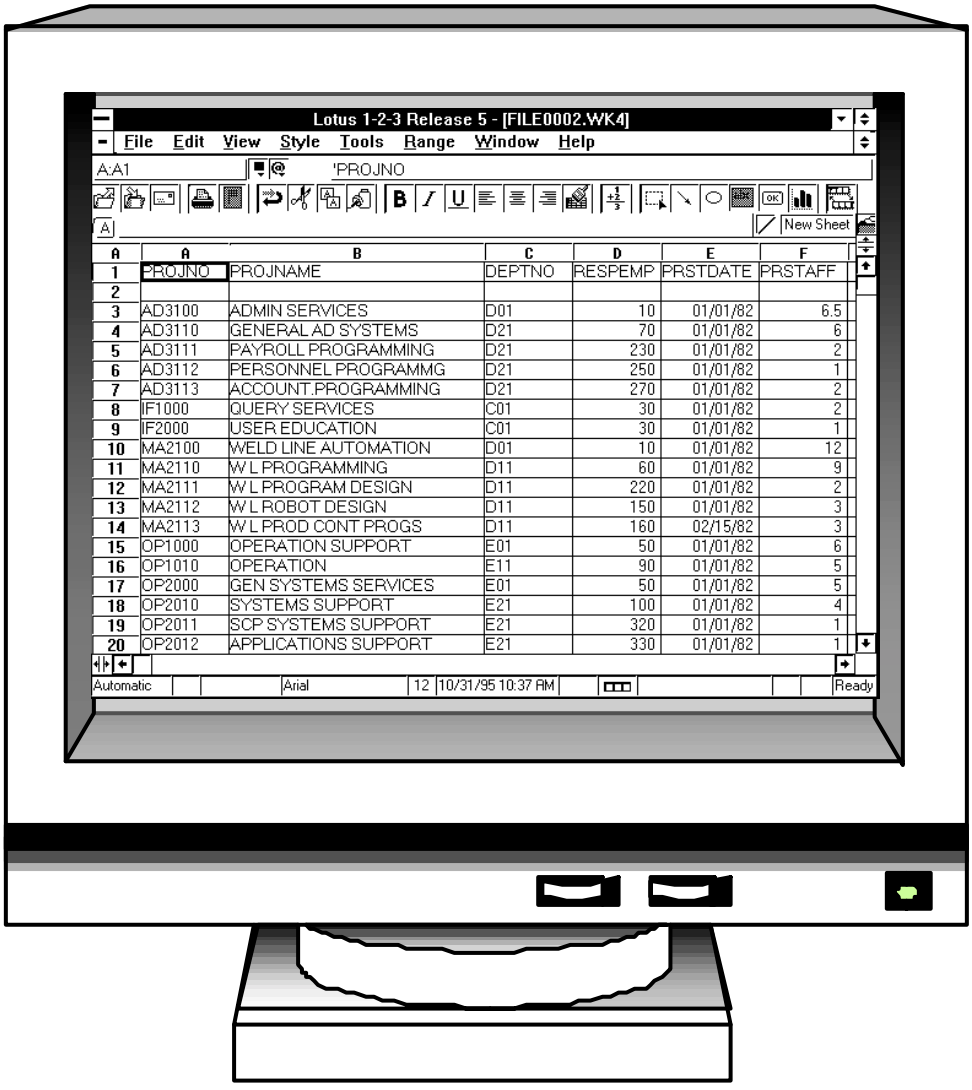
Using DB2 Data in PC Programs

We've just seen how easy it is to use DB2 data in custom reports with Spectrum Writer. It's just as easy to turn your DB2 data into PC files with Spectrum Writer. Simply add the appropriate PC option to the OPTIONS statement. An example of using DB2 data in a Lotus 1–2–3 spreadsheet is shown in **Figure 64**.This example shows the same "project table" data being used in a Lotus 1–2–3 spreadsheet.

These Control Statements:

```
OPTION:  LOTUS    DB2SUBSYS('DB2T')
INPUT:   PROJECT  DB2NAME('DSN8230.PROJ')
COLUMNS: PROJNO
         PROJNAME
         DEPTNO
         RESPEMP
         PRSTDATE
         PRSTAFF
```

Produce this Lotus 1-2-3 Spreadsheet:



	A	B	C	D	E	F
1	PROJNO	PROJNAME	DEPTNO	RESPEMP	PRSTDATE	PRSTAFF
2						
3	AD3100	ADMIN SERVICES	D01	10	01/01/82	6.5
4	AD3110	GENERAL AD SYSTEMS	D21	70	01/01/82	6
5	AD3111	PAYROLL PROGRAMMING	D21	230	01/01/82	2
6	AD3112	PERSONNEL PROGRAMMG	D21	250	01/01/82	1
7	AD3113	ACCOUNT.PROGRAMMING	D21	270	01/01/82	2
8	IF1000	QUERY SERVICES	C01	30	01/01/82	2
9	IF2000	USER EDUCATION	C01	30	01/01/82	1
10	MA2100	WELD LINE AUTOMATION	D01	10	01/01/82	12
11	MA2110	W L PROGRAMMING	D11	60	01/01/82	9
12	MA2111	W L PROGRAM DESIGN	D11	220	01/01/82	2
13	MA2112	W L ROBOT DESIGN	D11	150	01/01/82	3
14	MA2113	W L PROD CONT PROGS	D11	160	02/15/82	3
15	OP1000	OPERATION SUPPORT	E01	50	01/01/82	6
16	OP1010	OPERATION	E11	90	01/01/82	5
17	OP2000	GEN SYSTEMS SERVICES	E01	50	01/01/82	5
18	OP2010	SYSTEMS SUPPORT	E21	100	01/01/82	4
19	OP2011	SCP SYSTEMS SUPPORT	E21	320	01/01/82	1
20	OP2012	APPLICATIONS SUPPORT	E21	330	01/01/82	1

Figure 64. Using DB2 data in a Lotus 1-2-3 spreadsheet

What Fields Are in Your DB2 Table?

You may not remember the names of all of the fields defined for your DB2 table. Spectrum Writer will list the DB2 fields available in your DB2 file for you. Just use the SHOWFLDS(YES) parm in your INPUT statement:

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       SHOWFLDS(YES)
```

The above statement causes a list to be printed showing each DB2 field available from the DSN8230.PROJ table. This list appears in the Spectrum Writer control statement listing. The list also indicates the data type (character, numeric, date or time) of each of the DB2 fields.

The SHOWFLDS parm can also be used in the READ statement.

Using the WHERE Parm

Here's how Spectrum Writer interacted with the DB2 subsystem in order to produce the report in **Figure 63** (page 394). Spectrum Writer first opened a "cursor" with DB2 that "selected" the DB2 fields needed to produce the report. It then "fetched" from DB2 all the rows for that cursor. Since no INCLUDEIF statement was used, Spectrum Writer included in the report *all* the rows that were returned by DB2.

Now let's consider a more advanced report. What if we want to include only the records for department D21 in our report. Of course, the standard way to do that with Spectrum Writer is to use an INCLUDEIF statement, like this:

```
INCLUDEIF: DEPTNO = 'D21'
```

And that method works just fine! If you use this statement, Spectrum Writer would again fetch *all* rows from the DB2 table. Spectrum Writer would then examine the DEPTNO field in each row and include in the report only those rows where the DEPTNO field contained "D21".

But when using DB2 data as your input, there is another way to accomplish the same thing. You can let DB2 do the record selection rather than Spectrum Writer. To do this, use a WHERE parm in the INPUT statement:

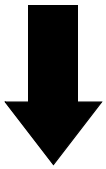
```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       WHERE(DEPTNO = 'D21')
```

The WHERE parm in the INPUT statement serves the same function as the WHERE clause in a DB2 "SELECT" statement. It tells DB2 which rows we want from the DB2 table. If your INPUT statement contains a WHERE parm, Spectrum Writer will include it as a WHERE clause in the SELECT statement that it builds for DB2. (If your INPUT statement does not have a WHERE parm, the SELECT statement will not have a WHERE clause, and DB2 will return *all* rows from the DB2 table.)

In the example above, the WHERE parm causes DB2 to return to Spectrum Writer only those rows from the project table whose DEPTNO field equals "D21". If you used this WHERE parm, you would not need an INCLUDEIF statement. You would want Spectrum Writer to include all the rows that DB2 returned to it.

These Control Statements:

```
OPTION:  DB2SUBSYS('DB2T')
INPUT:   PROJECT  DB2NAME('DSN8230.PROJ')
         WHERE(DEPTNO = 'D21')
         ORDERBY(PROJNAME)
TITLE:  'PROJECTS FOR DEPARTMENT D21'
COLUMNS: PROJNO
         PROJNAME
         DEPTNO
         RESPEMP
         PRSTDATE
         PRSTAFF
```



Produce this Report:

```
                    PROJECTS FOR DEPARTMENT D21

PROJNO  ________PROJNAME_________ DEPTNO RESPEMP PRSTDATE   PRSTAFF

AD3113 ACCOUNT.PROGRAMMING       D21   000270  01/01/82          2.00
AD3110 GENERAL AD SYSTEMS        D21   000070  01/01/82          6.00
AD3111 PAYROLL PROGRAMMING       D21   000230  01/01/82          2.00
AD3112 PERSONNEL PROGRAMMG       D21   000250  01/01/82          1.00

*** GRAND TOTAL (4 ITEMS)                                      11.00
```

Remarks:

- we could have achieved the same result by omitting the WHERE and ORDERBY parms, and adding these statements:

```
INCLUDEIF: DEPTNO = 'D21'
SORT:      PROJNAME
```

Figure 65. Using the WHERE and ORDERBY parms

As far as the final report goes, using the WHERE parm yields identical results to using the INCLUDEIF statement. Feel free to use whichever method you're most comfortable with. The example in **Figure 65** (page 398) uses a WHERE parm in the INPUT statement.

> **Performance Note:** Which one of these methods is more efficient? There is no "right" answer for all cases. It depends on various factors, including the percentage of records that are included in the report. For long–running jobs, where performance is an important consideration, you may want to try running the job each way and choose the method that works best in your particular case.

You can also use a combination of the WHERE parm *and* the INCLUDEIF statement. If you do, DB2 will pass to Spectrum Writer all rows that meet the WHERE conditions. Of those rows, Spectrum Writer will then include in the report only the ones that meet the INCLUDEIF statement conditions.

See "WHERE Parm Syntax" (page 405) for further details about the syntax allowed in the WHERE parm.

Using the ORDERBY Parm

Another optional parm in the INPUT statement is the ORDERBY parm. (Note that this parm must be spelled with no imbedded space.)

The ORDERBY parm in Spectrum Writer serves the same function as the ORDER BY clause in a DB2 "SELECT" statement. It tells DB2 what order to pass the rows to Spectrum Writer in. If your INPUT statement contains an ORDERBY parm, Spectrum Writer will include it as an ORDER BY clause in the SELECT statement that it builds for DB2. (If your INPUT statement does not have a ORDERBY parm, the SELECT statement will not have an ORDER BY clause. Then DB2 will pass Spectrum Writer the rows in an "arbitrary" order.)

Use this parm if you want DB2 to pass its rows to Spectrum Writer in a certain order. You may wish to use this parm rather than using a SORT statement. When no SORT statement is used, Spectrum Writer outputs the data in the same order that DB2 passes it to Spectrum Writer in.

The example in **Figure 65** (page 398) uses an ORDERBY parm in the INPUT statement.

Within the ORDERBY parm, you may list one or more DB2 fields, along with the optional keywords ASC and DESC (for "ascending" and "descending.") Here are two examples of INPUT statements that use the ORDERBY parm:

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       ORDERBY(DEPTNO, PROJNAME)
```

The above example would cause DB2 to return the rows from the project table to Spectrum Writer in department number order, with "ties" being further sorted in project name order.

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       WHERE(DEPTNO = 'D21')
       ORDERBY(PROJNAME DESC)
```

The above statement would cause the rows from the project table to be returned to Spectrum Writer in *descending* project name order. As you can see, you are allowed to use

both the WHERE and ORDERBY parms, if you wish. Their order in the INPUT statement is not important.

Note: If you want one or more control breaks in your report, you should use the SORT statement (rather than the ORDERBY parm). That is because Spectrum Writer only allows control breaks on fields that are in a SORT statement.

Note: You can use both an ORDERBY parm and a SORT statement, though this would rarely be useful. DB2 would pass the rows from the DB2 table to Spectrum Writer in the order specified in the ORDERBY parm. Spectrum Writer would then sort the final report according to the SORT statement.

Using Multiple DB2 Tables

Sometimes the DB2 table in your INPUT statement will not contain all the data you need for a report or a PC file. In that case, you can use one or more READ statements to obtain data from additional DB2 tables.

Let's begin by reviewing how the READ statement works with VSAM files. The file named in the INPUT statement is called the "primary input file." Spectrum Writer always reads this primary input file sequentially. Then, each time a record is read from the primary file, Spectrum Writer reads one additional record from each VSAM file named in a READ statement. The READKEY parm (in the READ statement) tells Spectrum Writer what key to use when performing the read. The key is usually a field from the primary input file.

You can also use READ statements with DB2 tables. Each READ statement will cause one row of data to be read from a DB2 table (or multiple rows if the MULTI parm is used). Instead of using a READKEY parm, use the WHERE parm to identify which row(s) you want to read. (Please refer to "Using the WHERE Parm" on page 397. The WHERE parm's syntax is discussed in "WHERE Parm Syntax" on page 405.)

Let's start with the DB2 report on page 394 to illustrate the use of the READ statement. That report shows data from the "project" DB2 table. One of the items in the project table is called RESPEMP. This is the employee number of the project's "responsible employee." Now suppose we want to include the employee's actual *name* in our report. The employee name is not kept in the project table. But it is kept in a different DB2 table — the employee table.

We can use the following statements to get data from both the project and the employee tables for use in our report.

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')

READ:  EMPLOYEE
       DB2NAME('DSN8230.EMP')
       WHERE(EMPNO = RESPEMP)
```

Notice that the READ statement, like the INPUT statement, begins with a Spectrum Writer file name. It also has the DB2NAME parm. And, unlike the INPUT statement, the WHERE parm is *required* in a READ statement.

Here's how Spectrum Writer will process the above statements. The primary input to the report is the project DB2 table. So, Spectrum Writer will retrieve all rows from the DB2 project table. For each row from the project table, Spectrum Writer will now also fetch a single row from the employee table. The row from the employee table will be the row whose EMPNO field equals the RESPEMP field from the project table.

As a result of these two statements, you now have access to *every DB2 field* in both the project and the employee DB2 tables. You can use those DB2 fields in your COLUMNS statement, SORT statement, COMPUTE statements, and so on. This simple way of linking multiple DB2 table is one of Spectrum Writer's most powerful features. All it takes is a single READ statement.

The report in **Figure 66** (page 402) illustrates this example. Our report now includes LASTNAME, which is a column from the employee DB2 table. This report shows the last name of the employee responsible for each project.

You can also use the ORDERBY parm in the READ statement. As mentioned, by default Spectrum Writer fetches only a single row from a READ file (for each row retrieved from the INPUT file). It is possible that the WHERE clause will not uniquely identify a single row in the READ file. In that case, you can use the ORDERBY parm to determine which row DB2 will return *first* to Spectrum Writer. For example, if there were more than one employee with the same employee number in the employee table, you might specify:

```
READ: EMPLOYEE
      DB2NAME('DSN8230.EMP')
      WHERE(EMPNO = RESPEMP)
      ORDERBY(LASTNAME)
```

The above statement specifies that DB2 should return rows from the employee table in LASTNAME order. Therefore, if multiple rows existed for a certain employee number, DB2 would return the row whose LASTNAME came first alphabetically. If no ORDERBY parm is specified and multiple rows meet the WHERE condition, DB2 will return the rows in an "arbitrary" order. When processing READ statements, Spectrum Writer always uses the *first* row returned by DB2.

> **Note:** For simplicity's sake, in this discussion we implied that Spectrum Writer *always* reads a row from each READ file. In some cases, Spectrum Writer may be able to detect that data from an auxiliary input table will not actually be needed in the run and, to improve performance, will not perform the read.

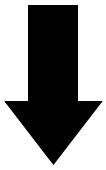
The complete READ statement syntax is shown on page 578.

One-to-Many Table Matching

If you want to use *all of the rows* that meet the WHERE parm conditions, add the MULTI parm to your READ statement. When the READ statement has the MULTI parm, Spectrum Writer creates and processes "logical input records" by matching the primary input file row with *each* qualifying row from the auxiliary input file. For more information on how the MULTI parm works, see "How to Perform "One–to–Many" Reads" on page 232.

These Control Statements:

```
OPTION: DB2SUBSYS('DB2T')

INPUT:  PROJECT   DB2NAME('DSN8230.PROJ')

READ:   EMPLOYEE  DB2NAME('DSN8230.EMP')
        WHERE(EMPNO = RESPEMP)

TITLE:  'LISTING OF PROJECT DB2 TABLE'
COLUMNS: PROJNO
         PROJNAME
         DEPTNO
         RESPEMP
         LASTNAME
         PRSTDATE
         PRSTAFF
```



Produce this Report:

LISTING OF PROJECT DB2 TABLE

PROJNO	PROJNAME	DEPTNO	RESPEMP	LASTNAME	PRSTDATE	PRSTAFF
AD3100	ADMIN SERVICES	D01	000010	HAAS	01/01/82	6.50
AD3110	GENERAL AD SYSTEMS	D21	000070	PULASKI	01/01/82	6.00
AD3111	PAYROLL PROGRAMMING	D21	000230	JEFFERSON	01/01/82	2.00
AD3112	PERSONNEL PROGRAMMG	D21	000250	SMITH	01/01/82	1.00
AD3113	ACCOUNT.PROGRAMMING	D21	000270	PEREZ	01/01/82	2.00
IF1000	QUERY SERVICES	C01	000030	KWAN	01/01/82	2.00
IF2000	USER EDUCATION	C01	000030	KWAN	01/01/82	1.00
MA2100	WELD LINE AUTOMATION	D01	000010	HAAS	01/01/82	12.00
MA2110	W L PROGRAMMING	D11	000060	STERN	01/01/82	9.00
MA2111	W L PROGRAM DESIGN	D11	000220	LUTZ	01/01/82	2.00
MA2112	W L ROBOT DESIGN	D11	000150	ADAMSON	01/01/82	3.00
MA2113	W L PROD CONT PROGS	D11	000160	PIANKA	02/15/82	3.00
OP1000	OPERATION SUPPORT	E01	000050	GEYER	01/01/82	6.00
OP1010	OPERATION	E11	000090	HENDERSON	01/01/82	5.00
OP2000	GEN SYSTEMS SERVICES	E01	000050	GEYER	01/01/82	5.00
OP2010	SYSTEMS SUPPORT	E21	000100	SPENSER	01/01/82	4.00
OP2011	SCP SYSTEMS SUPPORT	E21	000320	MEHTA	01/01/82	1.00
OP2012	APPLICATIONS SUPPORT	E21	000330	LEE	01/01/82	1.00
OP2013	DB/DC SUPPORT	E21	000340	GOUNOT	01/01/82	1.00
PL2100	WELD LINE PLANNING	B01	000020	THOMPSON	01/01/82	1.00
*** GRAND TOTAL (20 ITEMS)						73.50

Figure 66. A report that uses data from two different DB2 tables

Using Data from Three DB2 Tables

In the previous example, we showed how to use a READ statement to obtain data from a second DB2 table. But you're not limited to using only two DB2 tables at a time. Spectrum Writer allows you to use up to 15 different DB2 tables in a single run.

In this section, we'll show another example of using multiple DB2 tables in a single run. This time, we'll use two READ statements. That will give us access to the data from three DB2 tables altogether.

Let's pick up with the report we just produced in **Figure 66** (page 402). That report contains data from the project DB2 table. It also shows the "responsible employee's" last name, which comes from the employee DB2 table. Now suppose we want to show the *department name* for each project (not just the department number). Another DB2 table, called the department table, contains the names of each department. We'll read a row from that table in order to get the department name.

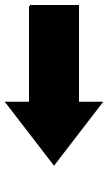
```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')

READ:  EMPLOYEE
       DB2NAME('DSN8230.EMP')
       WHERE(EMPNO = RESPEMP)

READ:  DEPARTMENT
       DB2NAME('DSN8230.DEPT')
       WHERE(DEPARTMENT.DEPTNO = PROJECT.DEPTNO)
```

Notice the READ statement on the previous page. In its WHERE parm we had to use record name prefixes to uniquely identify the DEPTNO fields. If we had written DEPTNO by itself, it would have resulted in an "ambiguous field name" error. That's because a field named DEPTNO exists in the project table *and* in the department table. We prefixed each occurrence of DEPTNO with a record name to eliminate the ambiguity. The WHERE parm correctly identifies the row that we want to read from the department file. It is the row whose own DEPTNO field equals the DEPTNO field from the project table. (The use of record names is discussed further in "WHERE Parm Syntax" on page 405.)

The report in **Figure 67** uses the three statements above.

These Control Statements:

```
OPTION: DB2SUBSYS('DB2T')
INPUT:  PROJECT     DB2NAME('DSN8230.PROJ')

READ:   EMPLOYEE    DB2NAME('DSN8230.EMP')
        WHERE(EMPNO = RESPEMP)

READ:   DEPARTMENT  DB2NAME('DSN8230.DEPT')
        WHERE(DEPARTMENT.DEPTNO = PROJECT.DEPTNO)

TITLE: 'LISTING OF PROJECT DB2 TABLE'
COLUMNS: PROJNO
         PROJNAME
         DEPARTMENT.DEPTNO
         DEPTNAME
         RESPEMP
         LASTNAME
         PRSTDATE
         PRSTAFF
```



Produce this Report:

```
                                LISTING OF PROJECT DB2 TABLE

                          DEPARTMENT
PROJNO  PROJNAME          DEPTNO    DEPTNAME              RESPEMP  LASTNAME  PRSTDATE  PRSTAFF

AD3100 ADMIN SERVICES          D01  DEVELOPMENT CENTER    000010  HAAS      01/01/82     6.50
AD3110 GENERAL AD SYSTEMS      D21  ADMINISTRATION SYSTEMS 000070  PULASKI   01/01/82     6.00
AD3111 PAYROLL PROGRAMMING     D21  ADMINISTRATION SYSTEMS 000230  JEFFERSON 01/01/82     2.00
AD3112 PERSONNEL PROGRAMMG     D21  ADMINISTRATION SYSTEMS 000250  SMITH     01/01/82     1.00
AD3113 ACCOUNT.PROGRAMMING     D21  ADMINISTRATION SYSTEMS 000270  PEREZ     01/01/82     2.00
IF1000 QUERY SERVICES          C01  INFORMATION CENTER    000030  KWAN      01/01/82     2.00
IF2000 USER EDUCATION          C01  INFORMATION CENTER    000030  KWAN      01/01/82     1.00
MA2100 WELD LINE AUTOMATION    D01  DEVELOPMENT CENTER    000010  H AAS     01/01/82     2.00
MA2110 W L PROGRAMMING         D11  MANUFACTURING SYSTEMS 000060  STERN     01/01/82     9.00
MA2111 W L PROGRAM DESIGN      D11  MANUFACTURING SYSTEMS 000220  LUTZ      01/01/82     2.00
MA2112 W L ROBOT DESIGN        D11  MANUFACTURING SYSTEMS 000150  ADAMSON   01/01/82     3.00
MA2113 W L PROD CONT PROGS     D11  MANUFACTURING SYSTEMS 000160  PIANKA    02/15/82     3.00
OP1000 OPERATION SUPPORT       E01  SUPPORT SERVICES      000050  GEYER     01/01/82     6.00
OP1010 OPERATION               E11  OPERATIONS            000090  HENDERSON 01/01/82     5.00
OP2000 GEN SYSTEMS SERVICES    E01  SUPPORT SERVICE       000050  GEYER     01/01/82     5.00
OP2010 SYSTEMS SUPPORT         E21  SOFTWARE SUPPORT      000100  SPENSER   01/01/82     4.00
OP2011 SCP SYSTEMS SUPPORT     E21  SOFTWARE SUPPORT      000320  MEHTA     01/01/82     1.00
OP2012 APPLICATIONS SUPPORT    E21  SOFTWARE SUPPORT      000330  LEE       01/01/82     1.00
OP2013 DB/DC SUPPORT           E21  SOFTWARE SUPPORT      000340  GOUNOT    01/01/82     1.00
PL2100 WELD LINE PLANNING      B01  PLANNING              000020  THOMPSON  01/01/82     1.00


*** GRAND TOTAL (    20 ITEMS)                                                          73.50
```

Figure 67. A report that uses data from three different DB2 tables

WHERE Parm Syntax

The syntax allowed within the WHERE parm is similar to, but not identical to, the DB2 syntax for a WHERE clause (in the DB2 "SELECT" statement). This section discusses the differences from the DB2 syntax.

The main differences in syntax concern:

- **Record Name Prefixes:** Spectrum Writer allows you to prefix any field name in the WHERE parm with a Spectrum Writer record name (to eliminate possible ambiguity)
- **Date and Time Literals:** you may use either Spectrum Writer's own date and time literals, or DB2's date and time literals

In a DB2 WHERE clause, each operand in a comparison can be any of the following:

- the name of a **DB2 column** in the table
- the name of a **"host variable"** (in DB2 terminology)
- a **literal value**

Spectrum Writer also supports all 3 kinds of operands in the WHERE parm. Here is a short discussion of each type of operand.

DB2 columns

Your comparisons can refer to any DB2 column in the "current" DB2 table. (That is, the DB2 table named in the DB2NAME parm of the same statement.) For example:

```
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(DEPTNO = 'D21')
```

In the WHERE parm above, DEPTNO is the name of a DB2 column within the DSN8230.PROJ table. This WHERE parm would select all rows from the project table where the DEPTNO field is equal to the literal value 'D21'.

In this example, the Spectrum Writer WHERE parm syntax is identical to the DB2 WHERE clause's syntax. But a problem can arise if the DB2 column name is not unique. This happens when an earlier input file contains a field by the same name. It can also happen if you create a COMPUTE field with the same name as a DB2 column.

Let's assume that our primary input file also has a field named DEPTNO in it. In that case, the WHERE parm above would result in an "ambiguous field name" error. Spectrum Writer wouldn't know whether you were referring to the DEPTNO field in the primary input file, or the DEPTNO field in the current (PROJECT) DB2 table.

To avoid such ambiguity, Spectrum Writer allows you to prefix any field name within the WHERE parm with a record name. (For more information on record names, see "How to Name the Input File Records" on page 228. Briefly, each input record has a record name. This record name can be specified explicitly with the RECNAME parm of the INPUT and READ statements. If no RECNAME is specified, the record name is the same as the file name.) To

tell Spectrum Writer that we mean the DEPTNO field from the "current" DB2 table, we would write:

```
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(PROJECT.DEPTNO = 'D21')
```

In the above statement, we used the record name of the "current" table (PROJECT) to prefix the DB2 field name. Now Spectrum Writer knows that the DEPTNO operand refers to the DB2 column within the project table itself, and not to the DEPTNO field from the primary input file.

Note: Don't confuse Spectrum Writer's record name prefix with a DB2 qualifier. DB2 qualifiers are not necessary and are *not allowed* within Spectrum Writer's WHERE parm.

Note: Some COMPUTE fields are not associated with any input record, and therefore cannot be prefixed with a record name. If you have problems with ambiguous field names due to such a COMPUTE field, the solution may be to choose a different name for your COMPUTE field.

Host Variables

When a field name in a WHERE parm refers to a field that is *not* in the current DB2 table, that field must be passed to DB2 as a "host variable." Spectrum Writer takes care of this for you automatically. It substitutes a "host variable marker" in the WHERE clause that is passed to DB2. Consider the following statements:

```
COMPUTE: TEST-DEPT = TEST-LETTER + '21'
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(DEPTNO = TEST-DEPT)
```

In this example, we have created a COMPUTE field named TEST–DEPT. In the WHERE parm, DEPTNO is compared to this COMPUTE field. In this case, Spectrum Writer would recognize that TEST–DEPT is not a field within the project DB2 table. So, it substitutes a host variable marker for TEST–DEPT before passing the WHERE clause to DB2. Doing this provides DB2 access to Spectrum Writer's internal value for the COMPUTE field (TEST–DEPT).

Once again, if a host variable name is not unique, you may prefix it with a record name to make it unique.

There is an example of a host variable in the report in **Figure 67** (page 404). Notice the READ statement for the employee DB2 table. It looks like this:

```
READ: EMPLOYEE
      DB2NAME('DSN8230.EMP)
      WHERE(EMPNO = RESPEMP)
```

EMPNO is a field within the current (employee) table. But Spectrum Writer treats RESPEMP as a host variable, since it is not a field within the employee table. (RESPEMP is a field from an earlier DB2 table— the project table.)

Note: Do *not* use a colon (:) to indicate a "host variable" within the WHERE parm (as you would when writing SQL code). As explained above, Spectrum Writer examines each field name in your WHERE parm and determines whether it is the name of a DB2 column within the current table or not. Spectrum Writer automatically takes care of passing host variables to DB2 for you.

Literals

Your WHERE parm expression can contain any valid DB2 literal. In addition, you are allowed to use Spectrum Writer's own literal formats. For example, if you wanted to, you could use a date literal in DB2's ISO date format, like this:

```
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(PRSTDATE = '1993-01-31')
```

Or, you could use a Spectrum Writer date literal, like this:

```
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(PRSTDATE = 1/31/1993)
```

Either format will yield the same result. When you use DB2 format literals, Spectrum Writer's passes them in the WHERE clause to DB2 unchanged. When you use a Spectrum Writer literal, Spectrum Writer passes it as a "host variable" to DB2.

Note that for character and numeric literals, the formats are the same for DB2 and for Spectrum Writer. So your choice in choosing literals applies only to date and time literals.

Note: Floating point literals are not allowed.

For simplicity, the examples in this discussion have shown only a single test in the WHERE parm. However, you are allowed to specify as many tests as you like in your WHERE parm. For example:

```
READ: PROJECT
      DB2NAME('DSN8230.PROJ)
      WHERE(PRSTDATE <= 1/31/1993 AND (DEPTNO = 'D21' OR DEPTNO = 'E11'))
```

Customizing Your DB2 Fields

As mentioned earlier, no FILE or FIELD statements are needed to define the fields in a DB2 input file. Spectrum Writer recognizes the actual DB2 column names that are defined for your DB2 table.

Since FIELD statements are not supported for DB2 fields, how do you permanently define such things as:

- the column headings to use for a field
- the display format to use for a field
- whether or not a numeric field should be totalled in reports

You can use COMPUTE statements to perform such customization. Use a COMPUTE statement that simply assigns the value of a DB2 field to the COMPUTE field. The COMPUTE statement syntax supports column headings, display formats and the ACCUM/NOACCUM parms (which determine whether a field is totalled or not).

For example, let's pretend that our project DB2 table contains a column named PROJTEL, which is a telephone number stored in DB2's "integer" format. By default Spectrum Writer would treat it as a regular numeric field, which means it would be formatted with commas,

it would be totalled, etc. Of course, for a particular run you could change these defaults directly in your COLUMNS statement, like this:

```
COLUMNS: PROJTEL(PIC'(999) 999-9999', NOACCUM)
```

In the above statement we specified an override display format (a "picture"), to make the numeric value look like a telephone number. And we specified NOACCUM to prevent the column from being totalled at the end of the report.

But if you will be using a field in many different reports, it would be easier to specify the display format and the NOACCUM parm just once and then forget about them. Do that by using a COMPUTE statement, like this:

```
COMPUTE: TELEPHONE(PIC'(999) 999-9999', NOACCUM) = PROJTEL
```

Now, whenever the field TELEPHONE is used in a report, it will be formatted appropriately, and will not be totalled. You can use the same method to define *column headings* for a DB2 field:

```
COMPUTE: TELEPHONE(PIC'(999) 999-9999', NOACCUM, 'TEL#') = PROJTEL
```

Now TELEPHONE will have TEL# as its default column heading in reports and PC files.

Saving DB2 File Definitions

The previous section explained how to use COMPUTE statements to customize your DB2 fields. A convenient way to handle these COMPUTE fields is to store them in your Spectrum Writer Copy Library. (See "Keeping Your File Definitions in a Copy Library" on page 360 for detailed information on using the copy library.)

Briefly, here's what to do. Create a member in the copy library for the DB2 file you want to define. In that member, put a FILE statement that specifies the desired filename and its DB2 name. Then add one COMPUTE statement for each DB2 field that you wish to customize. You might also want to include COMPUTE statement for any commonly used *computations* involving the DB2 fields. Do not put any FIELD statements in this member. FIELD statements are not allowed for DB2 files.

For example, for the project DB2 table you might create a member named PROJECT in the copy library. It might contain these statements:

```
FILE:    PROJECT  DB2NAME('DSN8230.PROJ')
COMPUTE: TELEPHONE(PIC'(999) 999-9999', NOACCUM, 'TEL#') = PROJTEL
COMPUTE: NUMBER('PROJECT NUMBER')                        = PROJNO
COMPUTE: NAME('PROJECT NAME')                            = PROJNAME
COMPUTE: SHORT-PROJ-NAME                                 = #SUBSTR(PROJNAME,1,5)
COMPUTE: YEARLY-STAFF(PIC'ZZZ9')                         = PRSTAFF * 52
```

Now we could request reports or PC files from the project DB2 table as easily as this:

```
INPUT: PROJECT
COLUMNS: NUMBER  SHORT-PROJ-NAME  TELEPHONE  PRSTAFF  YEARLY-STAFF
```

Upon seeing the INPUT statement for PROJECT, Spectrum Writer would process the FILE and COMPUTE statements from the PROJECT member in the copy library. Since the FILE statement contains the DB2NAME parm for PROJECT, the INPUT statement doesn't need it.

The COLUMNS (and any other) statements can now refer to either the actual DB2 field name, or the COMPUTE fields that we defined. Using the COMPUTE field names results in the column headings and display formats that were specified for those fields.

This method makes DB2 files look and work just the same as non–DB2 files from your end–users point of view. A programmer can do the small amount of setup required. Then end–users can use DB2 data in Spectrum Writer without necessarily even knowing it comes from a DB2 table.

DB2 Setup

Before you use Spectrum Writer with DB2 data for the first time, some simple DB2 setup is required. (You will also need to perform this setup each time you install a new release level of Spectrum Writer.) In most shops, this DB2 setup is performed by a Database Administrator. The setup consists of these two steps:

- a new DB2 "plan" must be created and "bound." This plan identifies Spectrum Writer to your DB2 system.
- authority to execute this plan must then be "granted" to your users.

Note: If Spectrum Writer will be used on multiple DB2 subsystems, these steps should be performed on each of those subsystems.

1. **Creating the DB2 Plan**

   The first step is to create a new DB2 plan. The plan name should be "SPECTnnn", where nnn is the release level of Spectrum Writer. For example, the plan name for release 3.0.0 of Spectrum Writer is SPECT300. That is the plan name that Spectrum Writer assumes you will use.

   Note: It is possible to use a different plan name if that is necessary for some reason. But you will then have to tell Spectrum Writer the name of your plan in every job you run. That is done with the **DB2PLAN** option:

   ```
   OPTION: DB2PLAN('OURNAME')
   ```

   If you use "SPECTnnn" as your plan name, you will not need to use the above statement.

   After creating the SPECTnnn plan, you must "bind" two Spectrum Writer "DBRM" modules into that plan. You can perform the bind with ISPF, or any other way your shops prefers.

   Note: The DBRM modules were included with your original installation files.

2. **Granting DB2 Execute Authority**

   After you have created and bound the DB2 plan, you must grant "execute authority" for that plan. Generally you will grant execute authority for this plan to PUBLIC. That allows anyone in your shop to execute Spectrum Writer. But it does not mean that every user can now access every DB2 table in the shop! Each user's access will still be limited to those DB2 tables that they have been granted access to. Granting them execute authority on "SPECTnnn" simply allows them to execute the Spectrum Writer program with its DB2 Option.

Here's how a user's access is determined. Each Spectrum Writer job has a DB2 "authorization ID" that is (or is related to) the jobname used for the run. If a Spectrum Writer job tries to access a DB2 table which is not permitted for that jobname, DB2 will return an error message to Spectrum Writer. Spectrum Writer will not be able to access that particular table, and will print an error message to that effect. If the jobname does have authority to read the DB2 table, Spectrum Writer will then access the DB2 data and complete the run normally.

DB2 Restrictions

DB2 has certain restrictions which Spectrum Writer must observe. In particular, you should keep the following restriction in mind:

- DB2 allows a maximum precision of 15 digits in numeric operands. Any decimal digits also count toward this maximum of 15 digits. (Spectrum Writer allows a precision of 31 digits.) This means, for example, that any Spectrum Writer COMPUTE field that you refer to in a WHERE clause must never have a value smaller than –999999999999999 or greater than +999999999999999. And, if the field contains decimal digits, the allowed range of values is reduced even further.

Chapter 8. Operating System Considerations

Chapter Table of Contents

Chapter 8. Operating System Considerations 411

OS/390 Operating System Considerations 412
Execution JCL for Reports — OS/390 412
DD statements used by Spectrum Writer 414
Execution JCL for PC and Mainframe Files — OS/390 415
Spectrum Writer PROC — OS/390 417
Output File Options — OS/390 417
Considerations for Runs with Multiple Outputs — OS/390 419
Setting Up File Definitions — OS/390 420
Copy Library DD — OS/390 423
Input File DDs — OS/390 423
Specifying Shop–Wide Options — OS/390 424
Completion Codes — OS/390 425
VSE Operating System Considerations 425
Execution JCL for Reports — VSE 427
Execution JCL for PC and Mainframe Files — VSE 429
Output File Options — VSE 431
Input File DLBL/TLBLs — VSE 432
The Control Statement Listing — VSE 433
The EXEC Statement’s SIZE Parm — VSE 433
Specifying Sort Work Files — VSE 434
Considerations for Runs with Multiple Outputs — VSE 435
Setting Up File Definitions — VSE 437
Completion Codes — VSE 439

Chapter 8. Operating System Considerations

This chapter discusses various topics that are related to the specific operating system under which Spectrum Writer is executed. It is intended primarily for programmers who are setting up the job control language (JCL) for Spectrum Writer jobs.

The following operating systems are discussed:

- OS/390 (below)
- VSE (page 425)

OS/390 Operating System Considerations

The following sections discuss operating environment considerations for executing **Spectrum Writer MVS**. Spectrum Writer MVS runs under the MVS/SP, MVS/XA, MVS/ESA, OS/390 and z/OS operating systems. The following topics are presented:

- sample execution JCL for custom reports (page 412)
- sample execution JCL for output files, including PC files and mainframe files (page 415)
- a sample Spectrum Writer PROC (page 417)
- specifying the access method and LRECL to use for Spectrum Writer's output records (page 417)
- special considerations for runs that produce more than one report or output file (page 419)
- setting up file definitions in a Copy Library (page 420)
- the Copy Library DD statement (page 423)
- the input file DD statements (page 423)
- the DD statement available for start–up options (page 424)
- the jobstep completion codes (page 425)

Execution JCL for Reports — OS/390

This section explains:

- the JCL needed to produce Spectrum Writer reports

Chapter 2, "How to Request a Report" explained how to use Spectrum Writer's control statements to request custom reports. The JCL needed to produce such a report is very simple. **Figure 68** shows sample JCL for producing a Spectrum Writer report.

This JCL:

```
//SPECTWTR JOB 'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,    PRODUCE SPECTRUM WRITER REPORT
//               REGION=2048K
//STEPLIB  DD  DSN=SPECTWTR.LOADLIB,DISP=SHR LOADLIB TO USE
//SWCOPY   DD  DSN=SPECTWTR.COPYLIB,DISP=SHR COPY LIBRARY
//SWOUTPUT DD  SYSOUT=*                      REPORT OUTPUT
//SWLIST   DD  SYSOUT=*                      CONTROL LISTING
//SYSOUT   DD  SYSOUT=*                      SORT STATISTICS
//SYSUDUMP DD  SYSOUT=*                      DUMP OUTPUT
//SORTWK01 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))  SORT WORK FILE
//SORTWK02 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))  SORT WORK FILE
//SORTWK03 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))  SORT WORK FILE
//SALEFILE DD  DSN=PROD.SALES.DATA,DISP=SHR  SALES FILE
//SYSIN    DD  *                             CONTROL STATEMENTS
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
//
```

Produces this Report:

```
TUE  05/16/95    8:25 AM     DATA FROM SALES-FILE                    PAGE     1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT          TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL          101.38         6.09
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE              137.00         8.22
EAST   MORRISON   03/29/95 15:30:22 STAR MARKET              44.35         2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY           29.65         1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI            14.99         0.90
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL             234.45        14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES            9.98         0.60
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE              135.75         8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY               9.98         0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY              10.25         0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN                121.76         7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN                 10.25         0.62
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING           500.00        30.00
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER             23.87         1.43

*** GRAND TOTAL (14 ITEMS)                                1,383.66        83.05
```

Remarks:

- the Spectrum Writer control statements in SPECTWTR.COPYLIB(SALES) would automatically be processed by Spectrum Writer during this run. Appendix F, "Files Used in Examples" (page 648) shows the statements in that member.
- the SALEFILE DD is necessary since SALES–FILE is used as an input in the report. The FILE statement for SALES–FILE specifies SALEFILE as the DDNAME to use.

Figure 68. Sample Spectrum WriterSpectrum Writer JCL for reports - OS/390

The JCL to produce reports from a particular input file only needs to be set up once. Once it's written, you can use the same JCL to produce as many different reports from that file as you like. Only the Spectrum Writer control statements (SYSIN) will be different in each run.

DD statements used by Spectrum Writer

Here is a description of the DD statements used by Spectrum Writer.

DD Statements Used by Spectrum Writer		
DDNAME	**REQUIRED**	**USED FOR**
SYSIN	Yes	Control statements describing the desired report or PC file
SWLIST	Yes	Spectrum Writer writes the control statement listing, error messages, and end–of–job statistics here.
SWOUTPUT	Yes	Spectrum Writer writes the actual report or output file here.
SWOUT002 **SWOUT003** **...**	No	Spectrum Writer writes the second, third, etc. report or output file to these DD statements.
SWCOPY	No	Points to the Spectrum Writer Copy Library
SWOPTION	No	Used for installation–wide options. Points to a dataset containing Spectrum Writer control statements which should be processed before the control statements in SYSIN.
SYSOUT	Yes	Sort program statistics. (Not required if a sort will not be performed during the run.)
SORTWK01 **SORTWK02** **SORTWK03 ...**	Yes	Sort work files. (Not required if a sort will not be performed during the run, or if these files are dynamically allocated at your shop.)
SRT2WK01 **SRT2WK02** **SRT2WK03 ...** **SRT3WK01** **SRT3WK02** **SRT3WK03 ...** **...** **SR10WK01** **SR10WK02** **SR10WK03 ...**	No	Sort work files to use when sorting the second, third, tenth, etc. report or output file in a run.

DD Statements Used by Spectrum Writer		
DDNAME	**REQUIRED**	**USED FOR**
STEPLIB	Yes	The load library where the SPECTWTR load module (and any exit program modules) are located. If DB2 tables will be used, this should also point to the library where the DB2 run–time modules (DSNTIAR, for example) are located. (Not required if these modules are located in a default steplib library.)
XXXXXXXX	Yes	One DD for each input file that will be used during the run. The DDNAME to use is specified in the DDNAME parm of the FILE statement that defines the file.

Execution JCL for PC and Mainframe Files — OS/390

This section explains:

- the JCL needed to produce Spectrum Writer output files

Chapter 3, "How to Request a PC File" explained how to use Spectrum Writer's control statements to request PC files. Chapter 4, "Beyond the Basics" included a section on creating mainframe output files (page 280).

The only JCL difference when creating PC (and mainframe) files is in the SWOUTPUT DD. Rather than routing the output to SYSOUT, you will normally want to write the output records to a dataset. That way the dataset can be downloaded to a PC or used by another mainframe program.

Figure 69 shows sample JCL for writing a PC file to disk.

You may specify any LRECL (and valid BLKSIZE) that you want in the SWOUTPUT DD. Pick a record length that will be big enough to hold all of the fields that you want to write to the output file.

Since output files do not need the "carriage control character" found in report output lines, you will specify a RECFM of F or FB (not FBA).

For more information on available options for the output file, see "Output File Options — OS/390" (page 417).

This JCL:

```
//SPECTWTR JOB  'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,    PRODUCE SPECTRUM WRITER PC FILE
//               REGION=2048K
//STEPLIB  DD  DSN=SPECTWTR.LOADLIB,DISP=SHR      LOADLIB TO USE
//SWCOPY   DD  DSN=SPECTWTR.COPYLIB,DISP=SHR      COPY LIBRARY
//SWOUTPUT DD  DSN=MY.PC.FILE,DISP=(NEW,CATLG),   PC OUTPUT FILE
//             UNIT=SYSDA,SPACE=(CYL,1),
//             DCB=(RECFM=FB,LRECL=250,BLKSIZE=2500)
//SWLIST   DD  SYSOUT=*                           CONTROL LISTING
//SYSOUT   DD  SYSOUT=*                           SORT STATISTICS
//SYSUDUMP DD  SYSOUT=*                           DUMP OUTPUT
//SORTWK01 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))       SORT WORK FILE
//SORTWK02 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))       SORT WORK FILE
//SORTWK03 DD  UNIT=SYSDA,SPACE=(CYL,(5,1))       SORT WORK FILE
//SALEFILE DD  DSN=PROD.SALES.DATA,DISP=SHR       SALES FILE
//SYSIN    DD  *                                  CONTROL STATEMENTS
OPTIONS: PC
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
//
```

Produces this Output File:

```
" ","EMPL","SALES","SALES"," "," "," "
"REGION","NAME","DATE","TIME","CUSTOMER","AMOUNT","TAX"
" "," "," "," "," "," "," "
"SOUTH","JOHNSON   ","03/12/95","10:25:00","ACE ELECTRICAL ",      101.38,      6.09
"WEST ","BAKER     ","03/26/95","12:09:09","JACKS CAFE     ",      137.00,      8.22
"EAST ","MORRISON  ","03/29/95","15:30:22","STAR MARKET    ",       44.35,      2.66
"EAST ","MORRISON  ","03/30/95","19:05:41","A1 PHOTOGRAPHY ",       29.65,      1.78
"EAST ","SIMPSON   ","04/01/95","08:17:57","EUROPEAN DELI  ",       14.99,      0.90
"NORTH","JOHNSON   ","04/01/95","17:02:47","VILLA HOTEL    ",      234.45,     14.07
"NORTH","JOHNSON   ","04/05/95","14:33:10","MARYS ANTIQUES ",        9.98,      0.60
"WEST ","BAKER     ","04/12/95","14:31:12","JACKS CAFE     ",      135.75,      8.15
"WEST ","THOMAS    ","04/14/95","15:41:38","YOGURT CITY    ",        9.98,      0.60
"NORTH","JONES     ","04/15/95","07:58:32","EZ GROCERY     ",       10.25,      0.62
"NORTH","JONES     ","04/15/95","08:01:59","TOY TOWN       ",      121.76,      7.31
"NORTH","JONES     ","04/15/95","13:52:41","TOY TOWN       ",       10.25,      0.62
"SOUTH","JOHNSON   ","04/16/95","11:48:33","ACME BUILDING  ",      500.00,     30.00
"EAST ","SIMPSON   ","04/30/95","15:30:21","J & S LUMBER   ",       23.87,      1.43
```

Remarks:

- only the SWOUTPUT DD statement is different from the JCL used to produce a report (page 413)

Figure 69. Sample Spectrum Writer JCL for PC and Mainframe files — OS/390

Spectrum Writer PROC — OS/390

You may wish to create a PROC for Spectrum Writer. That makes it much easier to set up new Spectrum Writer jobstreams. A PROC also makes it easier for non–technical users to run Spectrum Writer jobs. Here is a an example of how such a PROC might look:

```
//SPECTWTR PROC COPYLIB='NULLFILE'
//SPECTWTR EXEC PGM=SPECTWTR,                    PRODUCE SPECTRUM WRITER REPORT
//              REGION=2048K
//STEPLIB  DD   DSN=SPECTWTR.LOADLIB,DISP=SHR    LOADLIB TO USE
//SWCOPY   DD   DSN=&&COPYLIB.,DISP=SHR          COPY LIBRARY
//SWLIST   DD   SYSOUT=*                         CONTROL LISTING
//SWOUTPUT DD   SYSOUT=*                         REPORT OUTPUT
//SYSOUT   DD   SYSOUT=*                         SORT STATISTICS
//SYSUDUMP DD   SYSOUT=*                         DUMP OUTPUT
//SORTWK01 DD   UNIT=SYSDA,SPACE=(CYL,5)         SORT WORK SPACE
//SORTWK02 DD   UNIT=SYSDA,SPACE=(CYL,5)         SORT WORK SPACE
//SORTWK03 DD   UNIT=SYSDA,SPACE=(CYL,5)         SORT WORK SPACE
//         PEND
```

Once the above PROC is created, you could now request a report with just the following simple JCL:

```
//SPECTWTR JOB  'REQUESTOR'
//STEP     EXEC SPECTWTR,COPYLIB='SPECTWTR.COPYLIB'
//SALEFILE DD   DSN=PROD.SALES.DATA,DISP=SHR
INPUT: SALES-FILE
COLUMNS: REGION  EMPL-NAME  SALES-DATE  CUSTOMER
//
```

Output File Options –– OS/390

This section explains:

- the default **access method** Spectrum Writer uses to write its output records, and how to override it
- the output records' default **record length** (LRECL), and how to override it

By default, Spectrum Writer writes its output (whether a report or an output file) to the SWOUTPUT DD using QSAM I/O. This is appropriate for writing to SYSOUTs (printer output) as well as for writing output files to standard disk and tape datasets.

If you prefer, you can write your output to an existing ESDS VSAM file. One reason to do that is to make the output file available to CICS transactions (which can only access VSAM files.) To write your report or output file to a VSAM dataset, specify the following option in your control statements:

```
OPTIONS: OUTTYPE(VSAM)
```

Most standard line printers can print only 132 characters of data per line. However, many laser printers support "forms" that allow you to print longer print lines. And when creating PC or mainframe files as output, you may want records that are hundreds — or even thousands — of bytes long in order to hold all the desired data.

Spectrum Writer supports output records up to approximately 16K bytes long. Here is how Spectrum Writer determines what record length (LRECL) to use in for the output of a particular run.

LRECL for QSAM Output Files

When writing QSAM output (Spectrum Writer's default) the LRECL used is:

1. the LRECL specified in the DCB parm of the SWOUTPUT DD in the JCL, if any, or
2. the LRECL specified in a file's label, when writing to an existing dataset, or
3. the OUTLRECL value (from an OPTIONS statement), if any, or
4. 133

In other words, if you are printing a report (SWOUTPUT is routed to SYSOUT) and you do not specify a LRECL either in the JCL or the control statements, Spectrum Writer creates 133–byte records. This allows for a standard 132–byte print line, plus a 1–byte "carriage control character." In such runs, if you specify more fields in the COLUMNS statement than will fit in 132 bytes, Spectrum Writer will print a message telling you that it is truncating one or more fields.

If you want a report that is wider than 133 bytes, you can specify your own LRECL. Do this in either the JCL or in the Spectrum Writer control statements. To specify the LRECL in the JCL, just use the DCB=LRECL=nnnnn parm, like this:

```
//SWOUTPUT DD SYSOUT=*,DCB=LRECL=201
```

The above DD statement tells Spectrum Writer to allow up to 200 characters in the report (again reserving one byte for the carriage control character). Spectrum Writer would only truncate columns that extended beyond column 200. (Of course, in order to print such a report your *printer* must also support 201–byte print lines.)

To specify the LRECL in the control statements, use a statement like this:

```
OPTIONS: OUTLRECL(201)
```

The above example accomplishes the same thing as specifying 201 in the LRECL parm in the JCL. If you specify this option, you do not need to specify the DCB=LRECL parm in your JCL.

> **Note:** To print wide reports on your laser printer, the laser printer may require some "setup" information. This will tell the printer, for example, to use a *condensed font* so that more characters can fit on the page. You may be able to use the PRTSETUP parm of the OPTIONS statement to send this setup string to your printer. Here is an example of using the PRTSETUP option (the actual setup string will be different for each shop):
>
> ```
> OPTIONS: PRTSETUP('+$$$DJDE$ JDE=40,FORMAT=L66200,DATA=(0,200),END;')
> ```

When creating QSAM output files, Spectrum Writer again defaults to 133 byte records (if it has no other LRECL information). In the case of output files, all 133 bytes are available for data, since no carriage control character is written for output files.)

However, if you write your file to an existing dataset, Spectrum Writer will automatically determine the LRECL of that dataset and let you create records up to that size (before printing truncation warning messages).

When writing to a new dataset, you can specify the desired LRECL in either the DCB=LRECL parm of the JCL, or with the OUTLRECL option in your control statements. For example, to create a 300–byte PC file, you might use this JCL statement:

```
//SWOUTPUT DD DSN=EXCEL.FILE,DISP=(NEW,CATLG),
//            DCB=(LRECL=300,BLKSIZE=3000,RECFM=FB),
//            SPACE=(CYL,5),UNIT=SYSDA
```

In the above example, Spectrum Writer would only truncate fields that extended beyond column 300.

LRECL for VSAM Output Files

For VSAM output files, the LRECL used is:

1. the OUTLRECL value from an OPTIONS statement (if it is valid for the VSAM file's definition), if any, or
2. 133 (if it is valid for the VSAM file's definition), or
3. the maximum RECORDSIZE value from the VSAM file's definition

VSAM files are assigned an average record length and a maximum record length when they are first defined. As long as your OUTLRECL value is no longer than the maximum record length defined for the VSAM file, Spectrum Writer will use that LRECL as the size of its output records. If no OUTLRECL option is specified, Spectrum Writer again defaults to writing 133–byte records. However, if the VSAM dataset was defined with a maximum record size less than 133, then Spectrum Writer defaults to writing records the size of the maximum record size defined for the file.

Considerations for Runs with Multiple Outputs — OS/390

When producing multiple outputs in a single run, there are some additional JCL considerations. Specifically you will need to:

- add a DD statement for each additional output
- possibly add sort work file DD statements for the additional outputs
- possibly take measures to increase the available storage

DD Statements for Additional Output Files

When requesting multiple reports, the execution JCL must have one DD statement for each report produced. Following are the default names of the DD statements that Spectrum Writer writes to:

- SWOUTPUT (for the first report)
- SWOUT002 (for the second report)
- SWOUT003 (for the third report), and so on

Add the appropriate DD statement(s) to your JCL, depending on how many reports you will be creating.

> **Note:** If desired, you can override the DD name used by Spectrum Writer for an output. You can also specify, for each output, the type of file to write (QSAM or VSAM) and — for new datasets — the LRECL to use. Specify your overrides with the

OUTDDN, OUTTYPE and/or OUTLRECL options (in an OPTIONS statement). The overrides will apply only to the report that is currently being defined.

DD Statements for Additional Sort Work Files

When producing multiple outputs, you may need to add DD statements to your JCL for additional sort work files. Spectrum Writer starts a separate Sort subtask for each (sorted) report in a run. When a sort cannot be performed entirely in memory, the Sort program must use temporary sort work files. These sort work files may *not* be shared — each sort must have its own separate work files. At some shops, sort work files are dynamically allocated by the Sort program. If your shop uses such dynamic allocation, you do not need to add any additional DDs to your JCL. Otherwise, you should add new sort work DD statements for each additional report. The sort work file DDs must be named as follows:

- SORTWK01, SORTWK02, SORTWK03, and so on (for the first report)
- SRT2WK01, SRT2WK02, SRT2WK03, and so on (for the second report)
- SRT3WK01, SRT3WK02, SRT3WK03, and so on (for the third report)

  ...

- SR10WK01, SR10WK02, SR10WK03, and so on (for the tenth report)
- SR11WK01, SR11WK02, SR11WK03, and so on (for the eleventh report)
- and so on

Note: If you have a conflict using these DDNAMEs, you can use the SORTDD option (in an OPTIONS statement) to specify a different 4-byte prefix to use (see page 574).

Storage Considerations with Multiple Outputs

Each additional report in a run requires additional storage. Mostly this storage is needed for the Sort program. If your job fails due to lack of storage, consider the following:

- increase the value of the REGION= parm in your JCL's JOB or EXEC statement.
- reduce the amount of storage used by each Sort. By default, Spectrum Writer allows 256K of storage for each Sort. You can change this default with the SORTSIZE option (in an OPTIONS statement):

  ```
  OPTION: SORTSIZE(128)
  ```

  The above statement causes the Sort program (for the current report) to use just 128K of storage. Note that the SORTSIZE option applies only to the output currently being defined. Thus to reduce the amount of storage used in all sorts, you should add a SORTSIZE parm (in an OPTIONS statement) to the control statements for each report.

Setting Up File Definitions — OS/390

Before running Spectrum Writer, some one–time setup is required. This setup consists of creating a Spectrum Writer Copy Library PDS, and then storing descriptions of your company's files in it. This is necessary before Spectrum Writer can produce reports or PC files from your company's data. (See "Using the Spectrum Writer Copy Library" on page 363 for a discussion of how the copy library is used.)

The following steps are needed to set up your Spectrum Writer Copy Library:

Step 1.
Allocate a new PDS to be used as your Spectrum Writer Copy Library. The purpose of this PDS is to store definition statements about the files in your shop. The PDS's LRECL should be 80 bytes. The blocksize may be any multiple of 80. The amount of space required will depend on how many files you expect to define to Spectrum Writer. (A Spectrum Writer file definition requires approximately the same amount of space as a Cobol record layout for the same file.) If you have no idea what size to allocate, try allocating 20 tracks, with 20 directory blocks.

If you prefer, you can use an existing 80–byte PDS (such as a Cobol copy library, etc.) However, it is *recommended* that a new PDS be created to serve exclusively as the Spectrum Writer Copy Library.

Step 2.
Create a new member in the copy library for the first file that you want to define to Spectrum Writer. For example, if you want to define your company's payroll file, you might create a new member named PAYROLL. Within this member, put a FILE statement defining the payroll file. For example, if the payroll file is a simple sequential file, you might enter the following:

```
FILE: PAYROLL  DDNAME(PAYROLL)  LRECL(1500)
```

The above statement defines a sequential file that will be referred to as "PAYROLL" in Spectrum Writer control statements. The DDNAME associated with this file will also be PAYROLL. Be sure to specify an LRECL value that's as big as the biggest record in your file. In our PAYROLL example, we specified 1500 as the largest record length. For more information on the FILE statement, see "How to Define a File" (page 328.)

Next, put one FIELD statement for each field in the payroll file. (For more information on the FIELD statement, see "How to Define a Field" on page 333.) For example, if the first two fields in the payroll file are a 10–byte last name and a 15–byte first name, you would enter the following:

```
FIELD: LAST-NAME  LENGTH(10)
FIELD: FIRST-NAME LENGTH(15)
```

It isn't required that you define *all* of the fields in the file to start with. If the file contains fields that you don't care about using with Spectrum Writer, you do not need to define those fields. If you skip over some fields, just use the COLUMN parm in the next FIELD statement to tell Spectrum Writer what column that field begins in.

When you are finished, the copy library member should contain a single FILE statement, followed by any number of FIELD statements. (Appendix F, "Files Used in Examples" on page 648 shows some examples of copy library members.) Save this copy library member when you are done.

> **Note:** If you have a Cobol or Assembler record layout for the file you are defining, you can use Spectrum Writer to convert that layout into FIELD statements for you. Or, you can even produce a report directly from the record layout, without using FIELD statements at all. Both of these options are described in "Using Cobol and Assembler Record Layouts" (page 369). To begin with, though, we suggest you define one or two small files manually (as described above) to get a clear idea of

how Spectrum Writer works. That will make it easier for you to later see how Spectrum Writer's Cobol and Assembler interpreter fits into the picture.

Step 3.

Put an alias entry for your file in the SWALIAS member, if necessary. This step is not required as long as you chose an 8–byte (or smaller) file name in Step 2, and used that same name as the member name in your PDS. That is just what we did in our PAYROLL example in Step 2 above. We used "PAYROLL" both for the file name (in the FILE statement) and for the member name in the copy library. So no alias entry would be needed in this example.

Here's the purpose of alias entries. Whenever Spectrum Writer processes an INPUT (or READ) statement, it tries to automatically copy the input file's definition statements from the copy library. To do that, it must determine which member of the copy library contains the definition statements for the file named in the INPUT statement. An alias entry relates the file name in the INPUT statement (which can be up to 70 characters long) to the 8–byte member name where that file's definition is stored. When the two names are the same, no alias is needed. But when you use longer file names, you'll need to create an alias entry. Put the entry in a special member named SWALIAS in your copy library. The alias entry has this format:

```
FILENAME = MEMBER
```

For example, let's say we wanted to call our payroll file HEADQUARTERS–PAYROLL. That name is too big to use as the member name in the copy library. So, you would pick a shorter member name to keep the file definition statements in — say HQPAYROL. Then just add an alias entry like this to the SWALIAS member:

```
HEADQUARTERS-PAYROLL = HQPAYROL
```

The above line tells Spectrum Writer that the file definition statements for the HEADQUARTERS–PAYROLL file are stored in the member named HQPAYROL. "HEADQUARTERS–PAYROLL" is the name that users will use for the file in Spectrum Writer control statements (such as the INPUT statement). It's also the name you will use in the FILE statement when defining the file. "HQPAYROL" will only be used internally by Spectrum Writer as the member name for reading the definition statements from the copy library. Appendix F, "Files Used in Examples" (page 648) also shows an example of the SWALIAS member in a copy library.

The alias lines may appear in any order within the SWALIAS member.

Step 4.

Repeat steps 2 and 3 for each file that you wish to define to Spectrum Writer.

Step 5.

In your execution JCL, make sure the SWCOPY DD points to this copy library that you just set up. It contains the file definition statements and the SWALIAS member.

Your Spectrum Writer Copy Library is now ready. You can now request all the custom PC files and reports you want from the files that you defined.

Copy Library DD — OS/390

We saw in the previous section how a copy library is used to store the definition statements for your company's files.

Use the SWCOPY DD in your execution JCL to point to the PDS that Spectrum Writer should use as the copy library during the run. Of course, you may want different runs to use different copy libraries. (Perhaps different departments in your company will want to maintain and use their own copy libraries.) Just point the SWCOPY DD to the appropriate PDS in each run.

You can also use the copy library to store any other frequently used set of control statements. Use the COPY statement to include statements from copy library members in your report requests.

For example, you might store a number of commonly used COMPUTE statements in the copy library. Or, if you frequently run reports that use multiple input files, you could store the INPUT statement, any COMPUTE statements needed to create the read keys, and the READ statements all as one member of the copy library. That way the end–users would not need to remember how to link all of the input files. They could just begin their report request with a COPY statement that does all of that for them.

Input File DDs — OS/390

This section explains:

- how to write the DD statements for the input files

In order for Spectrum Writer to produce a report (or output file), it must "open" and "read" from the input file specified in the INPUT control statement. If the report uses auxiliary input files (specified in READ statements), Spectrum Writer must also open and read from these files.

Make sure that the JCL used to run a Spectrum Writer report contains one DD statement for each input file used in the report.

How does Spectrum Writer know which DD to use when reading these files? The file named in an INPUT or READ statement must have been previously defined to Spectrum Writer with a FILE statement. The DDNAME parm in the FILE statement tells what DD to use when reading the file. (The FILE statement is normally kept in the Spectrum Writer Copy Library.)

An *override* DDNAME parm can also be specified directly in the INPUT or READ statement. When this happens, Spectrum Writer uses the override DDNAME, rather than the one from the FILE statement.

Random reads to VSAM files can be relatively slow. VSAM maintains two types of buffers — data and index — while processing input files. When a required data record or index record is already in one of VSAM's buffers, VSAM can use the buffer copy instead of having to perform actual disk I/O, thus improving performance. If your report will be reading a large number of records from a VSAM input file, you may want to increase the number of buffers that VSAM maintains. This will increase the likelihood that VSAM will find a needed

record already in one of its buffers. You can increase the number of data buffers (BUFND) and/or index buffers (BUFNI) in either of two ways:

1. in the execution JCL, use the AMP=('AMORG,BUFNI=nn,BUFND=nn') parm in the DD statement, or
2. in the INPUT or READ statement, use the BUFNI(nn) and BUFND(nn) parms.

For IBM's recommended BUFNI and BUFND values, see page 658.

CICS Users Note: One of VSAM's weaknesses is in its ability to maintain file integrity for a VSAM file that is being accessed from multiple regions. For example, if CICS has a VSAM file open for update at the same time that Spectrum Writer is reading that file, there is a possibility that Spectrum Writer will not see all of the records that are "in the file". The reason for this is that when updates are made to a VSAM file under CICS, CICS may not immediately write those updates out to the physical file; instead, it may maintain the updated records within its buffers to be written at a later time (sometimes *days* later if activity for a file is very slow). Since Spectrum Writer is running in another region, it does not have access to the updates within CICS's buffers— only to the records that have actually been written to the VSAM file. Thus, VSAM may not pass to Spectrum Writer all of the records that an online CICS user would "see" in the same file. The safest way to avoid this problem is to **issue a CEMT CLOSE** to the VSAM file (from CICS) before running any batch job (including Spectrum Writer) that will read that file.

Specifying Shop–Wide Options –– OS/390

There may be some options that your shop will want to use in *every* report. For example, you may want to always print 80 lines per page (rather than Spectrum Writer's default of 60). That is specified with an OPTIONS statement:

```
OPTIONS: PAGELEN(80)
```

Or, many international users may prefer to always see dates formatted in DD–MM–YY format. They might want this statement in all of their runs:

```
OPTIONS: FORMAT(DD-MM-YY)
```

Or, if your shop prints to a laser printer that can skip to new sheets of paper, you may want to specify a PRTSHEET parm. (This parm allows control breaks to skip to a new *sheet* of paper, rather than merely a new side of the page.) For example:

```
OPTIONS: PRTSHEET('+$$$DJDE$ SIDE=NUFRONT,END;')
```

You *could* type these statements at the beginning of every report requested at your shop. But there is an easier way. Store these (and any other similar statements) in a data set. (Most shops use a member of the Spectrum Writer Copy Library for this purpose, but you can also use a flat file.) Then, use the SWOPTION DD to point to this data set. For example:

```
//SWOPTION DD DSN=SPECTWTR.COPYLIB(SWOPTION),DISP=SHR
```

When a SWOPTION DD statement is present in the JCL, Spectrum Writer processes the statements contained in that data set *before* processing the SYSIN statements.

The use of the SWOPTION DD is entirely optional. You are not required to have such a DD.

Completion Codes –– OS/390

Upon completion, Spectrum Writer exits back to the operating system with one of the following completion codes:

SPECTRUM WRITER OS/390 COMPLETION CODES	
COMPLETION CODE	MEANING
0	No errors or warning messages issued. Spectrum Writer produced its output normally. (Some informatory messages may have been printed.)
4	Only warning messages were issued. Spectrum Writer produced its output as well as it could.
8	Error messages were issued. No output (or only a partial output) was produced.
16	Security error. Spectrum Writer has expired or some other error was detected in the authorization codes. No output was produced.

Note: The ONIOERROR parm (available in the INPUT, READ and OPTIONS statements) can be used to increase the severity of an I/O error during a run. For more information, see under the appropriate statement name in Chapter 10, "Control Statement Syntax".

Note: you can use the EMPTYCC option (page 562) to specify a special completion code to use for runs that are "empty" (that is, no records passed the inclusion tests). Or use the NOTEMPTYCC option (page 568) to special a special completion code to use for runs that do include one or more record.

VSE Operating System Considerations

The following sections discuss the JCL needed to execute **Spectrum Writer VSE**. Spectrum Writer VSE runs under DOS/VSE, VSE/SP and VSE/ESA. The following topics are presented:

- sample execution JCL for custom reports (page 427)
- sample execution JCL for output files, including PC files and mainframe files (page 429)
- specifying the type and record size of the output file (page 431)
- special considerations for runs that produce more than one report or output file (page 435)
- setting up file definitions in a Copy Library (page 437)
- the DLBL/TLBL statements required for input files (page 432)
- routing the control statement listing (page 433)

- specifying the SIZE parm in the EXEC JCL statement (page 433)
- using sort work files (page 434)
- the jobstep completion codes (page 439)

Execution JCL for Reports — VSE

This section explains:

- the JCL needed to produce Spectrum Writer reports

Chapter 2, "How to Request a Report" explained how to use Spectrum Writer's control statements to request custom reports. The JCL needed to produce such a report is very simple. **Figure 70** shows sample JCL for producing a Spectrum Writer report.

The JCL to produce reports from a particular input file only needs to be set up once. Once the JCL has been prepared, you can use it to produce as many different reports from that file as you like. Only the Spectrum Writer control statements (SYSIPT) will be different in each run.

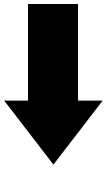
Here is a list of the logical unit assignments used by Spectrum Writer:

SPECTRUM WRITER VSE LOGICAL UNIT ASSIGNMENTS	
LOGICAL UNIT	DESCRIPTION
SYSIPT	the Spectrum Writer control statements are read from SYSIPT
SYS010	a "control listing" is written to this logical unit. It includes a listing of your Spectrum Writer control statements, any warning or error messages, and the end–of–run statistics.
SYS011	the report (or output file) produced by the run. This assignment can be changed with the OUTATTR option (page 431).
SYS012 **SYS013** **...**	the second, third, etc. report or output file produced in the run. These assignments can be changed with the OUTATTR option (page 431).

Note: To ensure that your report output is completely separate from the control listing messages and statistics, be sure to assign SYS010 and SYS011 to *different* virtual printers.

This JCL:

```
// JOB    SPECTWTR
// ASSGN  SYS010,SYSLST                CONTROL STATEMENT LISTING
// ASSGN  SYS011,006                   REPORT OUTPUT
// LIBDEF PHASE,SEARCH=LIB.SPECTWTR
// DLBL   SALEFIL,'SALES.MASTER.FILE'   INPUT FILE
// EXTENT SYS015,,,,6764,1000
// EXEC   SPECTWTR,SIZE=(SPECTWTR,300K)
OPTION:  SUBLIB('LIB.SPECTWTR')
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
/*
/&
```



Produces this Report:

```
TUE  05/16/95    8:25 AM     DATA FROM SALES-FILE                    PAGE     1

          EMPL     SALES    SALES
REGION    NAME     DATE     TIME     CUSTOMER          AMOUNT          TAX

SOUTH  JOHNSON    03/12/95 10:25:00 ACE ELECTRICAL          101.38         6.09
WEST   BAKER      03/26/95 12:09:09 JACKS CAFE              137.00         8.22
EAST   MORRISON   03/29/95 15:30:22 STAR MARKET              44.35         2.66
EAST   MORRISON   03/30/95 19:05:41 A1 PHOTOGRAPHY           29.65         1.78
EAST   SIMPSON    04/01/95 08:17:57 EUROPEAN DELI            14.99         0.90
NORTH  JOHNSON    04/01/95 17:02:47 VILLA HOTEL             234.45        14.07
NORTH  JOHNSON    04/05/95 14:33:10 MARYS ANTIQUES            9.98         0.60
WEST   BAKER      04/12/95 14:31:12 JACKS CAFE              135.75         8.15
WEST   THOMAS     04/14/95 15:41:38 YOGURT CITY               9.98         0.60
NORTH  JONES      04/15/95 07:58:32 EZ GROCERY              10.25         0.62
NORTH  JONES      04/15/95 08:01:59 TOY TOWN                121.76         7.31
NORTH  JONES      04/15/95 13:52:41 TOY TOWN                 10.25         0.62
SOUTH  JOHNSON    04/16/95 11:48:33 ACME BUILDING           500.00        30.00
EAST   SIMPSON    04/30/95 15:30:21 J & S LUMBER             23.87         1.43

*** GRAND TOTAL (14 ITEMS)                                1,383.66        83.05
```

Remarks:

- the Spectrum Writer control statements in member SALES.SPECTWTR of LIB.SPECTWTR would automatically be processed by Spectrum Writer during this run. Appendix F, "Files Used in Examples" (page 648) shows the statements in that member.
- the SALEFIL DLBL is necessary since SALES–FILE is used as an input in the report. The FILE statement for SALES–FILE specifies SALEFIL as the DLBL to use.

Figure 70. Sample Spectrum Writer JCL for reports — VSE

Execution JCL for PC and Mainframe Files — VSE

This section explains:

- the JCL needed to produce Spectrum Writer output files

Chapter 3, "How to Request a PC File" explained how to use Spectrum Writer's control statements to request PC files. Chapter 4, "Beyond the Basics" included a section on creating mainframe output files (page 280).

The only JCL difference when creating PC (or mainframe) files concerns where the output will be written. By default, Spectrum Writer writes output file records to the "printer" at SYS011 (just as it writes report lines).

If you want to download PC files from the POWER queue, this default may be just fine for you. In that case, use the same JCL for PC files as for reports (page 428).

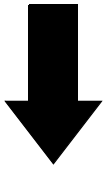
However, you may prefer to write output files to actual datasets, rather than the POWER queue. That way the dataset can be downloaded to a PC, or used by another mainframe job.

Figure 71 shows sample JCL for creating a PC file and writing it to a disk file. In this example, we used the OUTATTR parm to tell Spectrum Writer to write to a disk file rather than to a printer. We also added an appropriate DLBL statement for the output file to the JCL.

The OUTATTR option can also be used to specify the desired record size of your output file. You can also use it to write your output file to a VSAM file or a tape. The OUTATTR parm is discussed in more detail on page 431.

This JCL:

```
// JOB    SPECTWTR
// ASSGN  SYS010,SYSLST                  CONTROL STATEMENT LISTING
// LIBDEF PHASE,SEARCH=LIB.SPECTWTR
// DLBL   SALEFIL,'SALES.MASTER.FILE'    INPUT FILE
// EXTENT SYS015,,,,6764,1000
// DLBL   SWOUT,'PC.FILE'                OUTPUT FILE
// EXTENT SYS015,,,,5288,100
// EXEC   SPECTWTR,SIZE=(SPECTWTR,300K)
OPTION:  SUBLIB('LIB.SPECTWTR')
         PC
         OUTATTR(DASD,'SWOUT',250,2500)
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
/*
/&
```



Produces this Output File:

```
" ","EMPL","SALES","SALES"," "," "," "
"REGION","NAME","DATE","TIME","CUSTOMER","AMOUNT","TAX"
" "," "," "," "," "," "," "
"SOUTH","JOHNSON   ","03/12/95","10:25:00","ACE ELECTRICAL ",      101.38,      6.09
"WEST ","BAKER     ","03/26/95","12:09:09","JACKS CAFE     ",      137.00,      8.22
"EAST ","MORRISON  ","03/29/95","15:30:22","STAR MARKET    ",       44.35,      2.66
"EAST ","MORRISON  ","03/30/95","19:05:41","A1 PHOTOGRAPHY ",       29.65,      1.78
"EAST ","SIMPSON   ","04/01/95","08:17:57","EUROPEAN DELI  ",       14.99,      0.90
"NORTH","JOHNSON   ","04/01/95","17:02:47","VILLA HOTEL    ",      234.45,     14.07
"NORTH","JOHNSON   ","04/05/95","14:33:10","MARYS ANTIQUES ",        9.98,      0.60
"WEST ","BAKER     ","04/12/95","14:31:12","JACKS CAFE     ",      135.75,      8.15
"WEST ","THOMAS    ","04/14/95","15:41:38","YOGURT CITY    ",        9.98,      0.60
"NORTH","JONES     ","04/15/95","07:58:32","EZ GROCERY     ",       10.25,      0.62
"NORTH","JONES     ","04/15/95","08:01:59","TOY TOWN       ",      121.76,      7.31
"NORTH","JONES     ","04/15/95","13:52:41","TOY TOWN       ",       10.25,      0.62
"SOUTH","JOHNSON   ","04/16/95","11:48:33","ACME BUILDING  ",      500.00,     30.00
"EAST ","SIMPSON   ","04/30/95","15:30:21","J & S LUMBER   ",       23.87,      1.43
```

Figure 71. Sample Spectrum Writer JCL for PC and Mainframe files — VSE

Output File Options –– VSE

This section explains:

- the default **access method** Spectrum Writer uses to write its output records, and how to override it
- the output record's default **record size**, and how to override it
- how to use the **OUTATTR parm** (of the OPTIONS statement)

The OUTATTR ("Output Attributes") option lets you give Spectrum Writer explicit information about how and where to write its output. If no OUTATTR option is specified, Spectrum Writer makes these default assumptions:

- the output is going to a printer–type device. (Of course, in most cases the "printer" will actually be a POWER spool file.)
- the "printer" is at logical unit SYS011
- each record will be 133 bytes long (including a 1–byte carriage control character)

If you are creating reports, this default should work just fine for you. Your JCL will simply assign SYS011 to SYSLST or some other "printer" device.

Still, if you like you could use OUTATTR to specify a different SYSnnn or a different record size. For example:

```
OPTIONS: OUTATTR(PRT,SYS007,120)
```

The above statement tells Spectrum Writer to write the output file to a "printer" device at SYS007. The records should be 120 bytes long.

> **Note:** For report output, the first byte in each record is a "carriage control character." So in the example above, only 119 bytes would be available for the report data itself. For PC or mainframe file output (or when using the NOCC option) no control character is written. In that case, the entire length of the record is available for data.

When creating PC or mainframe files, you may prefer to write them to disk or tape, rather than to the POWER queue. And you may want a record size bigger (or smaller) than 133 bytes. To change the defaults, just use Spectrum Writer's OUTATTR option. This option lets you specify:

- the type of device to write to (choose from a printer, a DASD file (that is, a SAM file on disk), a VSAM file, or a tape file).
- the logical unit to write to. (Used with printer and tape files only.)
- the length of each output record. You can choose any record size you like (up to approximately 16K). Specify a record size that is big enough to hold all the data you plan to write.

Figure 71 (page 430) shows sample JCL for writing a PC file to a SAM file on disk. In that example, the following OUTATTR parm is used:

```
OPTIONS: OUTATTR(DASD,'SWOUT',250,2500)
```

The DASD parm in the above statement tells Spectrum Writer to write its output to a SAM disk file. The file is defined in the JCL by a DLBL statement named SWOUT. The records will be 250 bytes long, and the block size will be 2500.

Note: When writing to a disk or tape file, you can omit the ASSGN statement for SYS011 in your JCL.

You may use any record size (and valid block size) in the OUTATTR parm that you want. Pick a record size that will be big enough to hold all of the data you will be writing to the output file. If you do not specify a record size, Spectrum Writer assumes a default record size of 133 bytes.

You can also use the OUTATTR option to have Spectrum Writer write its output to a VSAM file. One reason to do this is so that CICS can access the output. You may want to use CICS to download the data to a PC. Here is an example of writing to a VSAM file:

```
OPTIONS: OUTATTR(VSAM,'OUTVSAM',450)
```

The above statement tells Spectrum Writer to write the output file to a VSAM file. (The VSAM file must have been defined ahead of time, and it must be defined as an ESDS file.) The DLBL for the VSAM file in the JCL will be named OUTVSAM. The records will be 450 bytes long. Note that block sizes are not used for VSAM files.

Finally, here is an example of writing Spectrum Writer's output to a tape file:

```
OPTIONS: OUTATTR(TAPE,'OUTFILE',SYS009,200,12000)
```

The above statement tells Spectrum Writer to write the output file to a tape mounted on logical unit SYS009. The TLBL for the output file in the JCL will be named OUTFILE. The records will be 200 bytes long, and the block size will be 12000.

Note: The complete syntax of the OUTATTR option is shown on page 555.

Input File DLBL/TLBLs –– VSE

This section explains:

- how to write the DLBL or TLBL JCL statements for your job's input files

In order for Spectrum Writer to produce a report (or output file), it must "open" and "read" from the input file specified in the INPUT statement. If the run uses auxiliary input files (specified in READ statements), Spectrum Writer must also open and read from those files.

How does Spectrum Writer know which DLBL or TLBL name to use when reading these files? The file named in an INPUT or READ statement must have been previously defined to Spectrum Writer with a FILE statement. The ATTR parm in the FILE statement specifies which DLBL (or TLBL) Spectrum Writer should use when reading the file. The ATTR parm also tells Spectrum Writer other important information about the file, such as its record size and block size.

Note: The FILE statement is normally kept in the Spectrum Writer Copy Library.

Note: The syntax of the FILE statement is shown on page 531.

An *override* ATTR parm can also be specified directly in the INPUT or READ statement. When this happens, Spectrum Writer uses the override DLBL (or TLBL) name, rather than the one from the FILE statement.

Make sure that your Spectrum Writer JCL contains one DLBL (or TLBL) statement for each input file needed to produce your report or output file. (An EXTENT JCL statement may also be needed for each DLBL statement.)

Random reads to VSAM files can be relatively slow. VSAM maintains two types of buffers — data and index — while processing input files. When a required data record or index record is already in one of VSAM's buffers, VSAM can use the buffer copy instead of having to perform actual disk I/O, thus improving performance. If your report will be reading a large number of records from a VSAM input file, you may want to increase the number of buffers that VSAM maintains. This may increase the likelihood that VSAM will find a needed record already in one of its buffers. You can increase the number of data buffers (BUFND) and/or index buffers (BUFNI) in either of two ways:

1. in the execution JCL, or
2. in the INPUT or READ statement, using the BUFNI(nn) and BUFND(nn) parms.

For IBM's recommended BUFNI and BUFND values, see page 658.

CICS Users Note: One of VSAM's weaknesses is in its ability to maintain file integrity for a VSAM file that is being accessed from multiple partitions. For example, if CICS has a VSAM file open for update at the same time that Spectrum Writer is reading that file, there is a possibility that Spectrum Writer will not see all of the records that are "in the file". The reason for this is that when updates are made to a VSAM file under CICS, CICS may not immediately write those updates out to the physical file; instead, it may maintain the updated records within its buffers to be written at a later time (sometimes *days* later if activity for a file is very slow). Since Spectrum Writer is running in another partition, it does not have access to the updates within CICS's buffers— only to the records that have actually been written to the VSAM file. Thus, VSAM may not pass to Spectrum Writer all of the records that an online CICS user would "see" in the same file. The safest way to avoid this problem is to **issue a CEMT CLOSE** to the VSAM file (from CICS) before running any

The Control Statement Listing — VSE

The control statement listing (which lists your control statements and any diagnostic messages, as well as end–of–run statistics) is always written to the printer–type device at SYS010. The record size is always 133 bytes, including a 1–byte carriage control character.

You should "assign" SYS010 to a *different* printer device than the one that SYS011 (the actual report) is assigned to. This prevents your control listing from being intermixed with your report output.

The EXEC Statement's SIZE Parm –– VSE

Spectrum Writer makes extensive use of the GETVIS portion of its partition. Therefore, you should provide a larger than normal GETVIS area by using the SIZE parm in your EXEC statement.

Spectrum Writer uses the GETVIS portion of the partition for these things:

- its own control blocks, used to process your request
- VSAM's control blocks
- any User Exits (written by your shop to perform custom processing) are also loaded into GETVIS storage.

The program area of the partition is used for the following:

- the Spectrum Writer phase itself (about 250K)
- the Librarian program (if you will be using Spectrum Writer's copy library feature)
- the Sort program (if you request that your report or output file be sorted)

The Librarian and Sort programs are used at different times, so they can use the same area in memory. Generally, reserving 300K for the Sort and/or Librarian programs is sufficient.

Therefore, we recommend using the following EXEC statement in your JCL:

```
// EXEC SPECTWTR,SIZE=(SPECTWTR,300K)
```

Of course, special considerations may cause you to want to experiment with the SIZE parm in your applications.

Specifying Sort Work Files — VSE

Most Spectrum Writer jobs will involve a sort. This is required in order to put your report or output file into the order specified by the SORT control statement. Spectrum Writer calls your shop's standard sort program to perform the sort. By default, the sort program is told to perform the sort entirely in memory. For large reports or output files, it may not be possible to perform the sort in memory— external sort work files will be needed.

In that case, you should do two things:

1. Provide one or more SORTWKn DLBL/EXTENT statements in your JCL. For example, you could add JCL statements similar to the following in order to provide the sort program with 2 work files:

   ```
   // DLBL   SORTWK1
   // EXTENT SYS016,,,,4124,1000
   // DLBL   SORTWK2
   // EXTENT SYS016,,,,3098,1000
   ```

2. Use the SORTWORKNUM option to tell Spectrum Writer how many sort work files are available for the sort program to use. For example if you added the two DLBL/EXTENT statements above, you would specify:

   ```
   OPTIONS: SORTWORKNUM(2)
   ```

Considerations for Runs with Multiple Outputs — VSE

When producing multiple outputs in a single run, some there are additional JCL considerations. Specifically you will need to:

- provide a logical unit or DLBL for each additional output
- possibly add DLBL statements for sort work files for the additional outputs
- possibly take measures to increase the available storage

Additional Output Attributes

By default, Spectrum Writer writes all reports to printer-type logical units. Following are the default logical units that Spectrum Writer writes to:

- SYS011 (for the first report)
- SYS012 (for the second report)
- SYS013 (for the third report), and so on

If you wish to write a report to a different logical unit, or to a SAM or VSAM file, use the OUTATTR option (in an OPTIONS statement).

The OUTATTR option describes the output attributes for the report currently being defined. For example, to write three reports to DLBLs named BYREG, BYCUST and BYAMT, you could use these control statements:

```
INPUT: SALES-FILE
OPTION: OUTATTR(SAM,'BYREG',100,1000)
TITLE: 'SALES BY REGION'
COLUMNS: REGION CUSTOMER AMOUNT TAX
SORT: REGION
BREAK: REGION

NEWOUT:
OPTION: OUTATTR(SAM,'BYCUST,150,1500)
TITLE: 'SALES BY CUSTOMER'
COLUMNS: CUSTOMER REGION AMOUNT TAX  EMPL-NAME
SORT: CUSTOMER
BREAK: CUSTOMER

NEWOUT:
OPTION: OUTATTR(SAM,'BYAMT',80,800)
TITLE: 'SALES SORTED BY DESCENDING AMOUNT'
COLUMNS: AMOUNT CUSTOMER
SORT: AMOUNT(D)
```

Additional Sort Work File DLBL Statements

When producing multiple outputs, you may need to add DLBL (and EXTENT) statements to your JCL for additional sort work files. Spectrum Writer starts a separate Sort subtask for each (sorted) report in a run. When a sort cannot be performed entirely in memory, the Sort program must use temporary sort work files. These sort work files may *not* be shared — each sort must have its own separate work files. At some shops, sort work files are dynamically allocated by the Sort program. If your shop uses such dynamic allocation, you may not need to add any additional DLBLs to your JCL. Otherwise, you should add new sort

work DLBL statements for each additional report. The sort work file DLBLs must be named as followed:

- SORTWK1, SORTWK2, SORTWK3, ... (for the first report)
- SRT2WK1, SRT2WK2, SRT2WK3, ... (for the second report)
- SRT3WK1, SRT3WK2, SRT3WK3, ... (for the third report)

  ...

- SR10WK1, SR10WK2, SR10WK3, ... (for the tenth report)
- SR11WK1, SR11WK2, SR11WK3, ... (for the eleventh report)
- and so on

Note: If you have a conflict using these DLBLs, you can use the SORTDD option (in an OPTIONS statement) to specify other 4-byte prefixes to use (see page 574).

Note: Many Sort programs assume, by default, that a single sort work file is available in the execution JCL. To explicitly tell the Sort program how many sort work files are provided for in the JCL, use the SORTWORKNUM option (in a Spectrum Writer OPTIONS statement). Spectrum Writer will then pass this information to the Sort program. For example:

```
OPTION: SORTWORKNUM(3)
```

The above statement tells Spectrum Writer that three sort work files are defined in the JCL for the report being defined. Note that the SORTWORKNUM option applies only to the report currently being defined. Thus, if applicable, you should include a SORTWORKNUM option among the control statements for *each* report.

Note: Even in shops where the sort work files are dynamically allocated, the SORTWORKNUM option may still be required in order to trigger the dynamic allocation.

Storage Considerations with Multiple Outputs

Each additional report in a run requires additional storage. Mostly this storage is used by the Sort program. If your job fails due to lack of storage, consider the following:

- increase the amount of reserved storage using the EXEC statement's SIZE parm. The storage used by the Sort program comes from this part of the partition. Thus, if your run produces three reports, each one using the default 256K for its sort, then 768K will be required just for the Sort programs. Round this up for other users of storage:

  ```
  // EXEC SPECTWTR,SIZE=(SPECTWTR,900K)
  ```

- reduce the amount of storage used by each Sort, By default, Spectrum Writer allows 256K for each Sort. You can change this default with the SORTSIZE option (in an OPTIONS statement):

  ```
  OPTION: SORTSIZE(128)
  ```

  The above statement causes the Sort program (for the current report) to use only 128K of storage. Note that the SORTSIZE option applies only to the output currently being defined. Thus to reduce the amount of storage used in all sorts,

you should add a SORTSIZE parm (in an OPTIONS statement) to the control statements for each report.

- try running the job in a larger partition

Setting Up File Definitions — VSE

This section explains:

- how to set up a Librarian sublibrary to serve as the **Spectrum Writer Copy Library**
- how to use the **SUBLIB option** to tell Spectrum Writer the name of the copy library

Before running Spectrum Writer, some one–time setup is required. This setup involves storing descriptions of your company's files in the Spectrum Writer Copy Library. This is necessary before Spectrum Writer can produce reports or PC files from your company's data. (See "Using the Spectrum Writer Copy Library" on page 363 for a discussion of how the copy library is used.)

The following steps are needed to set up your Spectrum Writer Copy Library:

Step 1.
Pick a Librarian sublibrary to use as your Spectrum Writer Copy Library. We recommend that you create a *new* sublibrary to be used exclusively for this purpose. However, you can use any Librarian sublibrary as your Spectrum Writer Copy Library.

Some shops may want to use *multiple* copy libraries with Spectrum Writer. (Perhaps one for each department in the company.) It is fine to do that. You will tell Spectrum Writer via a control statement the name of the copy library to use in each run.

Step 2.
Create a member in the copy library for the first file that you want to define to Spectrum Writer. The member *name* can be anything that you like. The member *type* should be SPECTWTR. For example, to define your company's payroll file, you might create a new member named PAYROLL.SPECTWTR.

This member should contain a FILE statement defining certain attributes of the file. For example, you might have the following:

```
FILE: PAYROLL  ATTR(DASD,'PAY',80,4000)
```

The above statement defines a DASD SAM file that will be referred to as "PAYROLL" in Spectrum Writer control statements. The DLBL name associated with this file will be PAY. The records are 80 bytes long, and the blocks are 4000 bytes long. (For more information on writing FILE statements, see page 331.)

Next, the member should contain one FIELD statement for each field in the payroll file. (For more information on writing FIELD statements, see page 333.) For example, if the first 2

fields in the payroll file were a 15–byte last name and a 10–byte first name, you might enter the following:

```
FIELD: LAST-NAME  LENGTH(15)
FIELD: FIRST-NAME LENGTH(10)
```

You do not need to define *all* of the fields in the file to start with. If the file contains fields that you don't care about using with Spectrum Writer, you do not need to define those fields. If you skip over some fields, just use the COLUMN parm in the next FIELD statement to tell Spectrum Writer what column that field begins in.

When you are finished, the copy library member should contain a single FILE statement, followed by any number of FIELD statements. (Appendix F, "Files Used in Examples" on page 648 shows examples of copy library members and their file definition statements.)

> **Note:** If you have a Cobol or Assembler record layout for the file you are defining, you can use Spectrum Writer to convert that layout into FIELD statements for you. Or, you can even produce a report directly from the record layout, without using FIELD statements at all. Both of these options are described in "Using Cobol and Assembler Record Layouts" (page 369). To begin with, though, we suggest you define one or two small files manually (as described above) to get a clear idea of how Spectrum Writer works. That will make it easier for you to later see how Spectrum Writer's Cobol and Assembler interpreter fits into the picture.

Step 3.

Put an alias entry for your file in the SWALIAS.SPECTWTR member, if necessary. This step is *not required* as long as you chose an 8–byte (or smaller) file name in Step 2 and used that same name as the member name in your copy library. That's just what we did in our PAYROLL example in Step 2 above. We used "PAYROLL" both for the file name (in the FILE statement) and as the member name in the copy library. So no alias entry would be needed in that example.

Here's the purpose of alias entries. Whenever Spectrum Writer processes an INPUT (or READ) statement, it tries to automatically copy the input file's definition statements from the copy library. To do that, it must determine which member of the copy library contains the definition statements for the file named in the INPUT statement. An alias entry relates the file name in the INPUT statement (which can be up to 70 characters long) to the 8–byte member name where that file's definition is stored. When the two names are the same, no alias is needed. But when you use longer file names, you'll need to create an alias entry. Put the entry in a special member named SWALIAS.SPECTWTR in your copy library. The alias entry has this format:

```
FILENAME = MEMBER
```

For example, let's say we wanted to call our payroll file HEADQUARTERS–PAYROLL. That name is too big to use as the member name in the copy library. So, you would pick a shorter member name to keep the file definition statements in — say HQPAYROL. Then just add an alias entry like this to the SWALIAS.SPECTWTR member:

```
HEADQUARTERS-PAYROLL = HQPAYROL
```

The above line tells Spectrum Writer that the file definition statements for the HEADQUARTERS–PAYROLL file are stored in the member named HQPAYROL.SPECTWTR. "HEADQUARTERS–PAYROLL" is the name that users will use for the file in Spectrum Writer control statements (such as the INPUT statement). It's also the name you will use in the FILE

statement when defining the file. "HQPAYROL" will only be used internally by Spectrum Writer as the member name for reading the definition statements from the copy library. Appendix F, "Files Used in Examples" (page 648) shows an example of a SWALIAS member in a copy library.

Note that only the member name (not the member type) is specified in an alias entry All copy library members should have a member type of SPECTWTR. (See the MEMTYPE option on page 566 for information on changing the default member type.)

The alias lines may appear in any order within the SWALIAS member.

Step 4.
Repeat steps 2 and 3 for each file that you wish to define to Spectrum Writer.

Step 5.
In your Spectrum Writer control statements, always begin with an OPTIONS: SUBLIB statement. This will tell Spectrum Writer the name of this copy library that you just set up. For example, if you named your copy library LIB.SPECTWTR, you would use the following statement:

```
OPTIONS: SUBLIB('LIB.SPECTWTR')
```

Your Spectrum Writer Copy Library is now ready. You can now request all the custom reports and output files that you want from the files that you have defined.

Completion Codes –– VSE

Upon completion, Spectrum Writer exits back to the operating system with one of the following completion codes:

SPECTRUM WRITER VSE COMPLETION CODES	
COMPLETION CODE	**MEANING**
0	No errors or warning messages issued. Spectrum Writer produced its output normally. (Some informatory messages may have been printed.)
4	Only warning messages were issued. Spectrum Writer produced its output as well as it could.
8	Error messages were issued. No output (or only a partial output) was produced.
16	Security error. Spectrum Writer has expired or some other error was detected in the authorization codes. No output was produced.

Note: The ONIOERROR parm (available in the INPUT, READ and OPTIONS statements) can be used to increase the severity of an I/O error during a run. For more information, see under the appropriate statement name in Chapter 10, "Control Statement Syntax".

Note: you can use the EMPTYCC option (page 562) to specify a special completion code to use for runs that are "empty" (that is, no records passed the inclusion tests). Or use the NOTEMPTYCC option (page 568) to special a special completion code to use for runs that do include one or more record.

Part 2. Reference Manual

Chapter 9. General Syntax Rules

Chapter Table of Contents

Chapter 9. General Syntax Rules 441

Control Statements 443
- What Is a Control Statement? 443
- How to Write Control Statements 443
- How to Continue a Control Statement Onto Multiple Lines 444
- The Order of Control Statements 444
- How to Put Comments in Your Control Statements 445
- How to Put Page Breaks in the Control Listing 446

Names of Files, Fields, and Records 446
- Rules for Assigning Names 446
- How to Make Field Names Unique 447

How to Write Literals 448
- The Five Types of Data 448
- Character Literals 448
- Numeric Literals 449
- Date Literals 449
- Time Literals 450
- Bit Literals 450
- When Do You Need Quotes Around a Number? 450

PICTURE Display Formats 451
- Examples of PICTUREs 452
- Showing Scaled Numbers with PICTUREs 453
- How PICTUREs Work 455
- Time PICTUREs 458

Conditional Expressions 459
- How to Specify a Relation Condition 460
- Comparing Character Operands of Different Lengths 462
- Comparing Fields of Different Data Types 463
- Conditions Involving Explicit Literals 464
- How to Specify a Bit Field Condition 465
- How to Specify Multiple Conditions 465
- Conditional Expressions That Use AND 465
- Conditional Expressions That Use OR 466
- Conditional Expressions That Use Both AND and OR 467
- How to Shorten Long Expressions 468
- How to Negate Conditions 469

Examples of Conditional Expressions 470
Computational Expressions 472
Operands in Computational Expressions 473
Operators in Computational Expressions 473
Order of Operations 474
Examples of Computational Expressions 474

Chapter 9. General Syntax Rules

This chapter describes:

- the general syntax rules that apply to all control statements.

Control Statements

What Is a Control Statement?

Control statements are the means by which you describe a desired report or PC file to Spectrum Writer. Each control statement describes some aspect of the desired report or PC file. You can request a report with as few as two control statements. Or, you might use dozens of statements to request a very complicated report. A PC file can be requested with as few as three control statements.

How to Write Control Statements

You will probably type your control statements into a dataset using an editor. Each line in your dataset will be 80 columns long. Each dataset line does not necessarily correspond to one control statement. A single control statement may be typed onto multiple lines.

As mentioned, the lines in your dataset will each be 80 columns long. However, Spectrum Writer only looks at the first 72 columns of each line. (This is because some editors store information of their own in the last 8 columns of each line.) Be sure not to type any part of a control statement past column 72, because Spectrum Writer will ignore that part.

Every control statement begins with a statement name. The statement name must begin in the very first column of a line, and must be immediately followed by a colon. Here are examples of how several common control statements begin:

```
INPUT:
TITLE:
COLUMNS:
```

What follows the statement name depends on the particular statement. The complete syntax for each control statement is found in Chapter 10, "Control Statement Syntax."

After the statement name, the rest of each control statement is "free format." That means that you are not required to put the field names or keywords in any specific column— you can type them wherever you like in the line (up to column 72). You may use as many blanks around the words in your statement as you like (to make the statement easier to read). You may also use commas to separate words if you like. In general, Spectrum Writer treats

commas like blanks. The following four control statements are all equivalent, even though they are spaced differently:

```
COLUMNS: LAST-NAME FIRST-NAME TOTAL-SALES
COLUMNS: LAST-NAME, FIRST-NAME, TOTAL-SALES
COLUMNS:        LAST-NAME       FIRST-NAME TOTAL-SALES
COLUMNS: LAST-NAME
         FIRST-NAME   TOTAL-SALES
```

Notice that the last example above used *two lines* for the COLUMNS statement. You may use as many lines as you want for a single control statement.

How to Continue a Control Statement Onto Multiple Lines

Sometimes a control statement will contain so much information that it will have to be split onto multiple lines. Other times, you may want to spread a control statement onto multiple lines just to make it easier to read (and perhaps easier to modify later).

The only rule about "continuation lines" is that *they must begin with a blank in the first column*. That is how Spectrum Writer can tell whether a line is a continuation of the preceding statement, or the beginning of a new statement. Lines with a *non–blank* in column 1 are new statements. Lines with a *blank* in column 1 are continuations of the preceding statement.

Where should you split a statement onto a separate line? Generally, you can end a line anywhere that a space is allowed in the statement, and then continue on the next line. This means that you *cannot* split a statement in the middle of a field name or a keyword. Split a statement between such words, where spaces would be allowed.

You may, however, split a statement in the middle of a character literal. This is necessary, for instance, if you have a very long literal for a TITLE statement. To continue a character literal onto a new line, simply type the literal right up through column 72 of the first line, and then resume typing in column 2 of the next line. (Remember that column 1 of the second line must be left blank, since it is a continuation line.) If a third line is required, do the same thing: type through column 72 of the second line and resume in column 2 of the third line, and so on.

Here is an example of a TITLE statement that has a long literal text split across two lines. (The scale shows the column numbers of the lines).

```
1...5...10...15...20...25...30...35...40...45...50...55...60...65...70...75...80
TITLE: 'LIST OF CUSTOMERS FOR THE NEW, ADVANCED, MINIATURIZED, SOLID STA
 TE, ZERO WAIT STATE PAPER CLIP'
```

The Order of Control Statements

There is no rigid order required for the control statements. The general rule is that any file name or field name referred to in a control statement must *already* have been defined (in a preceding control statement). For example, a COLUMNS statement that names a computed field cannot appear *before* the COMPUTE statement that defines that field.

Although there is no requirement as to specific control statement order, the following suggested order is a logical way to organize most requests:

1. Start with any OPTIONS statements needed. Some options *must* appear before any other control statements, so it's a good idea to group all OPTIONS statements together at the beginning of your request.
2. Put the INPUT statement next. Spectrum Writer must know the input file name early, so that it will know which field names to allow in subsequent statements.
3. If your request will use READ statements, they should appear next. Again, this lets Spectrum Writer know what additional field names are available for use in subsequent statements. If your READ statement uses a computed field as it key, place the necessary COMPUTE statement(s) just ahead of the READ statement.
4. Next comes any COMPUTE statements needed to define additional fields you will be using in your request.
5. The TITLE, COLUMNS, SORT, and FOOTNOTE statements may now follow in any order. BREAK statements, if used, must follow the SORT statement.

The following sample request follows the above guidelines:

```
OPTIONS: SUMMARY
INPUT:   SALES-FILE
COMPUTE: SPECIAL-KEY = "9" +#SUBSTR(EMPL-NUM,2,2)
READ:    EMPL-FILE  READKEY(SPECIAL-KEY)
COMPUTE: DISCOUNT = AMOUNT * 0.05
TITLE:   'SALES REPORT'
COLUMNS: SALES-DATE  CUSTOMER  AMOUNT  DISCOUNT  LAST-NAME
SORT:    CUSTOMER
BREAK:   CUSTOMER TOTAL('CUSTOMER TOTAL') SPACE(PAGE)
```

How to Put Comments in Your Control Statements

Often it is helpful to include comments among your control statements. Comments are ignored by Spectrum Writer but provide good documentation to other people looking at your control statements. There are two ways to include comments in your control statements.

1. use an entire **comment line**, by putting an asterisk (*) in column 1 of the line
2. or, **embed comments** in other control statements, by surrounding your comment with the symbols /* and */

Any line that begins with an asterisk (*) in column 1 is considered a comment line. The entire line will be ignored by Spectrum Writer. Comment lines may appear anywhere among the control statements.

Here is an example of how to use comment lines:

```
*******************************************************
*                                                     *
*      THIS REPORT PRODUCES AN EMPLOYEE DIRECTORY     *
*                                                     *
*******************************************************
INPUT: EMPL-FILE
COLUMNS: LAST-NAME  FIRST-NAME  TOTAL-SALES
```

You may also *embed* comments within control statements. Use a slash and asterisk pair (/*) to indicate the beginning of your comment and use an asterisk and slash pair (*/) to indicate

the end of your comment. Everything between these symbols will be ignored by Spectrum Writer. You are allowed to begin and end your comment on different lines.

Here are some examples of imbedded comments:

```
INPUT:   EMPL-FILE  /* THIS IS THE EMPLOYEE MASTER FILE */
COLUMNS: LAST-NAME  FIRST-NAME  /* LAST YEARS SALES */  TOTAL-SALES
SORT:    TOTAL-SALES(DESC)      /* SORT LARGEST SALE FIRST  */
         LAST-NAME              /* THEN SORT BY LAST NAME   */
```

Warning: *Do not* begin or end an imbedded comment in a *comment line* (one beginning with an asterisk in column 1). Comment lines are *completely* ignored, including any /* or */ symbols within them.

Also, do not use columns 1 and 2 of any line for the /* or the */ symbols. Column 1 is reserved for statement names and asterisks only.

How to Put Page Breaks in the Control Listing

There is one special comment line that you can use to control the paging of the control listing report. A comment line beginning with the word "*PAGE" will cause the control listing to skip to a new page. This is useful when you are listing many control statements and would like to separate them into logical groups. Here is an example of using the "*PAGE" comment line:

```
COPY:    MSTRDEF  LIST(YES)
*PAGE
INPUT:   MASTER-FILE
COLUMNS: NAME  DATE  ADDRESS
```

In the control listing, the INPUT and COLUMNS statements would appear on a new page, separate from the statements copied by the COPY statement.

Names of Files, Fields, and Records

Rules for Assigning Names

You may make up your own names for the files, fields, and records you will be working with. (These names are assigned in the FILE, FIELD, COMPUTE, INPUT, and READ statements.) The only requirements for the names you assign are:

- each character in the name *must be* one of the following
  - an alphabetic character
  - a numeric character
  - a dash (–)
  - an underscore character (_)
  - an ampersand (@)
  - a dollar sign ($)
  - a pound sign (#)

- the first character of the name *may not* be a numeric character or a dash (–)
- the total length of the name must fit on a single line. Names may not be split across lines.

  Note: We recommend that you do not name your fields beginning with the pound sign (#). This is to avoid confusion with Spectrum Writer's built–in fields and functions, which all begin with a pound sign. For example, Spectrum Writer's built–in field that contains the current system date is named #TODAY.

Some examples of valid names are:

```
EMPL-NUM
HIRE-DATE
X
PRIMARY-SUBSCRIBERS-SOCIAL-SECURITY-NUMBER
SALARY
A12345-67890
EMPLOYEE_NAME
SUBSCRIPTION#
```

How to Make Field Names Unique

When you are producing reports that use multiple files as input, it is possible that a field with the *same name* may exist in more than one input file. For example, you may be using both the EMPL–FILE and the SALES–FILE as inputs to a report. There happens to be a field named EMPL–NUM in *both* of these files.

When this situation occurs, you can indicate which of the two fields you mean by using a record name to "qualify" the field name. (By default, a file's record name is the same as the file name.) A qualified name consists of a record name, followed by a period, followed by a field name (with no spaces in between). For example, to list the EMPL–NUM field from the EMPL–FILE, you would use this statement:

```
COLUMNS: EMPL-FILE.EMPL-NUM
```

And, to list to the EMPL–NUM field from the SALES–FILE, you would use this statement:

```
COLUMNS: SALES-FILE.EMPL-NUM
```

If you just used EMPL–NUM by itself in the COLUMNS statements above, you would get an error message indicating that the field name was not unique.

Record names are also discussed in "How to Name the Input File Records" (page 228).

Note: We mentioned earlier that a field name may not be split across multiple lines. If a field name is qualified, the prefix, the period, and the field name itself must *all* fit on a single line. For this reason, it is better not to make your field names extremely long.

How to Write Literals

A "literal" is a *constant* value. In other words, its value does not depend on the contents of any input record. Literals are used in many of Spectrum Writer's control statements.

There are five types of literals, corresponding to the five types of data recognized by Spectrum Writer. Before going into the syntax of literals, let's review the five types of data.

The Five Types of Data

All data processed by Spectrum Writer falls into one of five general data types. This applies to data contained in fields as well as to literal values. The five types of data are:

- character
- numeric
- date
- time
- bit

Spectrum Writer knows what kind of data exists in a particular field from the TYPE parm specified in its FIELD statement. Spectrum Writer knows what kind of data a literal value contains from its format (discussed below). It is important to know an item's data type for the following reasons:

- in a conditional expression, you may only compare two items if they are of the same type.
- in a computational expression, all operands must be of the same type (with one exception discussed later). Also, the operations allowed will depend on the data type of the operands.
- in print expressions, any display format parms must be appropriate for the data type of the field involved.

Character Literals

Character literals are always enclosed in either single quotation marks (apostrophes) or double quotation marks (' or "). You can use whichever character you like. Whichever of these characters you choose, be sure to begin and end the literal with the same character. If you need to include that same character (the single or double quotation mark) *within* the literal, you may do so by entering *two* of the characters together. Character literals may be up to 256 characters long. (See page 444 for instructions on writing literals that don't fit on a single line.) Here are some examples of character literals used in TITLE statements:

```
TITLE:  'END OF YEAR REPORT'
TITLE:  "LAST QUARTER'S EARNINGS"
TITLE:  'MANAGER''S STATUS REPORT'
```

Another way to specify character literals is to use their **hexadecimal** representation. This is useful when you wish to enter a special character which has no associated key on the

keyboard, such as certain graphics characters, or the LOW–VALUE and HIGH–VALUE literals used in Cobol. A hexadecimal literal begins with an "X", immediately followed by the hexadecimal value enclosed in quotation marks. (Again, you can use either single or double quotation marks.) Remember that only the digits 0 through 9, and the letters A though F are allowed in hexadecimal literals. Here are some examples of hexadecimal literals used in various control statements:

```
OPTIONS:   COLSEP(X'05')
COMPUTE:   LOW-VALUES = X'00000000'
TITLE:     X"4040C1"
INCLUDEIF: EMPL-NUM = X'FFFFFF'
```

Since each byte contains 2 hex digits, your hexadecimal literals should normally contain an even number of hex digits. Spectrum Writer pads hexadecimal literals that do not contain an even number of digits by adding a trailing hex "0".

Numeric Literals

Numeric literals should *not* be enclosed in quotation marks. A numeric literal may contain only the numeric digits 0 though 9, a decimal point, and a sign character (+ or –). If a sign character is used, it must be the first character in the literal. Commas are *not* allowed in numeric literals. A numeric literal may contain a maximum of 31 digits. Here are some examples of numeric literals used in various control statements:

```
COMPUTE:   INTEREST =  .125
COMPUTE:   FACTOR   =  -1
INCLUDEIF: AVERAGE  >  1.5234
INCLUDEIF: TOTAL-SALES < 100000
```

Date Literals

Date literals also should *not* be enclosed in quotation marks. Specify date literals in either MM/DD/YYYY or MM/DD/YY format. We recommend using MM/DD/YYYY literals to reduce the possibility of erroneous results. Leading zeros in the month and day are optional. If you do use MM/DD/YY literals, the 2–digit years are assumed to fall between 1950 and 2049. That is because the default CENTURY option value is 50 (page 559). However, you may specify your own CENTURY option if you prefer a different century cutoff year.

Date literals must specify a date between January 1, 1900 and December 31, 2099 (inclusive). Here are some examples of date literals used in various control statements:

```
COMPUTE:   START-DATE  =  12/31/1989
COMPUTE:   END-DATE    =  7/4/98
INCLUDEIF: HIRE-DATE   <  2/15/04
INCLUDEIF: HIRE-DATE   <  04/15/1999
INCLUDEIF: HIRE-DATE   <  1/1/2001
```

Note: Date literals must always be written using slashes (/) as the delimiter. The DATEDELIM option, if used, applies only to how dates are formatted in the output — it does not affect the way date literals must be written.

Note: If you prefer, you can choose to write *all* date literals in DD/MM/YYYY (or DD/MM/YY) format. Just place the DDMMYYLIT option (in an OPTIONS statement) at the beginning of your control statements. For example:

```
OPTIONS:   DDMMYYLIT
INCLUDEIF: HIRE-DATE < 15/4/1999
COMPUTE:   START-DATE = 31/12/89
```

Time Literals

Time literals also should *not* be enclosed in quotation marks. Specify time literals in either HH:MM:SS or HH:MM format. A leading zero in the hour portion of the time is optional. Time literals in HH:MM:SS format may also contain decimal parts of seconds— HH:MM:SS.SSS. Time literals must specify a time between 00:00:00 and 23:59:59. Here are some examples of time literals used in various control statements:

```
COMPUTE:   START-TIME     =    8:30
COMPUTE:   END-TIME       =   17:00:00
INCLUDEIF: SALES-TIME     >=  12:00  AND  <= 12:00:05
INCLUDEIF: TIME-ON-PHONE  <   00:00:01.5
```

Note: Time literals must always be written using colons (:) as the delimiter. The TIMEDELIM option, if used, applies only to how times are formatted in the output—it does not affect the way time literals must be written.

Bit Literals

There are no true bit literals in Spectrum Writer. However, there are two built–in functions which perform the same role. Literals are generally used in two ways:

- within a comparison, in a conditional expression
- as an operand in a computational expression

Within conditional expressions, no comparisons are allowed with bit fields. A bit field name is a condition all by itself. Therefore, no bit literal is required for comparisons. (For more information on this, see "Conditional Expressions" on page 459.)

Within the COMPUTE statement, you may use the built–in functions #ON and #OFF as the equivalent of bit literals. Since these are *functions* (which simply return the constant values ON or OFF), they are not technically literals. Here is a sample control statement that uses these built–in functions:

```
COMPUTE: NEW-EMPLOYEE =  WHEN(HIRE-DATE > 1/1/1990)  ASSIGN(#ON)
                         ELSE                        ASSIGN(#OFF)
```

When Do You Need Quotes Around a Number?

In most cases, matching data types comes naturally. Most people wouldn't try to compare a date field (like HIRE–DATE) with a character field (like LAST–NAME).

But, there is one area where mistakes in mixing data types are commonly made. That is when it comes to distinguishing between *character* fields that contain numeric characters,

and true *numeric* fields. For example, consider the EMPL–NUM field in the EMPL–FILE (defined in Appendix F, "Files Used in Examples" on page 648). Since this field contains an employee *number*, it is easy to think of it as a numeric field. But in reality it is defined as a *character* field. (It just happens to contain only "numeric" characters.) This means that when a comparison is made to it, a *character* literal must be used— not a *numeric* literal. For example, the following statement is valid:

```
INCLUDEIF: EMPL-NUM = '037'
```

The above statement would select all records for employee number 037. The character literal '037' (in quotes) is compatible with the *character* field EMPL–NUM. However, consider the following statement:

```
INCLUDE: EMPL-NUM = 037
```

The above statement is in error! It is attempting to compare a *character* field (EMPL–NUM) with the *numeric* literal 037 (without quotes).

A similar error might be made when trying to display EMPL–NUM in the report. Consider the following statement:

```
COLUMNS: EMPL-NUM(PIC'ZZ9')
```

The above statement is also invalid! It attempts to use a *numeric* display format (a PICTURE) to format a *character* field.

Of course, since the EMPL–NUM field in the records always contains a numeric character, we could have defined EMPL–NUM as a numeric field (by using TYPE(NUM) in the FIELD statement). Then, we *could* have used numeric literals and numeric display formats with the field. Had we defined EMPL–NUM as a numeric field, we would also want to specify the NOACCUM parm, to prevent the EMPL–NUM column from being totalled in reports.

So, when do you need quotation marks around numbers? Whenever the number is being used as a *character literal*, rather than a numeric literal.

> **Note:** To determine if a particular field has been defined as a character or a numeric field, add the SHOWFLDS(YES) parm to your INPUT **(**or READ**)** statement. This parm causes a listing of all of the fields defined for the file to appear in your control statement listing. The data type of each field (character or numeric) also appears in this listing.

> **Note:** For more discussion on character versus numeric fields, see "Should You Define a Field as Character or Numeric?" on page 339.

PICTURE Display Formats

A PICTURE is a special display format that describes how a numeric value should be displayed in a report. The PICTURE display format consists of the word PICTURE (or an abbreviation, such as PIC) immediately followed by text enclosed in either apostrophes or

quotation marks. (Do not put a space before the apostrophe or quotation mark.) For example:

```
PICTURE'text'
PIC'text'
```

The characters making up the text give a "picture" of how the formatted result should look. The PICTURE specifies such thing as:

- the **size** of the formatted output (that is, how many characters it will occupy in a print line)
- whether **leading zeros** should be displayed or suppressed
- whether **commas** (or some other character) will be used to separate the thousands, the millions, etc.
- whether a **floating dollar sign** should appear in the result
- where the **minus sign** should appear, for negative numbers
- where (and whether) a **plus sign** should be displayed for positive numbers
- how many **decimal digits** should print
- any **literal characters** that should be included in the formatted result
- whether **automatic scaling** of the number is wanted (so that the number is shown "in thousands," "in millions," etc.)

Examples of PICTUREs

Spectrum Writer's PICTUREs are very similar to COBOL's PICTURE clause, in case you are familiar with that. If you haven't worked with PICTUREs before, the best way to learn about them is probably to look at some examples. The following examples show the format produced by various PICTUREs. Pick a result that is similar to what you want, and use that PICTURE as a guide. Adjust the number of digit symbols in your PICTURE according to the size of the numbers that you will be printing.

In the table below, a sample positive value (1,234.56) and a sample negative value (-98,765.4) are used to demonstrate each PICTURE.

EXAMPLES OF PICTUREs

PICTURE	FORMATTED POSITIVE VALUE	FORMATTED NEGATIVE VALUE
PIC'999999999'	000001235	****S****
PIC'999999.9'	001234.6	****S***
PIC'999999.99'	001234.56	****S****
PIC'999999V99'	00123456	****S***
PIC'ZZZZZ9.99'	1234.56	-98765.40
PIC'ZZZZZ9V99'	123456	-9876540
PIC'ZZZ,ZZ9.99'	1,234.56	-98,765.40

EXAMPLES OF PICTUREs (CONTINUED)		
PICTURE	FORMATTED POSITIVE VALUE	FORMATTED NEGATIVE VALUE
PIC'---,--9.99'	1,234.56	-98,765.40
PIC'+++,++9.99'	+1,234.56	-98,765.40
PIC'ZZZ,ZZ9.99-'	1,234.56	98,765.40-
PIC'ZZZ,ZZ9.99+'	1,234.56+	98,765.40-
PIC'$$,$$$,$$9.99'	$1,234.56	-$98,765.40
PIC'ZZZ.ZZ9V,99'	1.234,56	-98.765,40
PIC'ZZZ ZZ9V,99'	1 234,56	-98 765,40
PIC'ZZZ.ZZ9V,99 DM'	1.234,56 DM	-98.765,40 DM
PIC'ZZZZZ9.99%'	1234.56%	-98765.40%

Note: The first several examples above resulted in size error indicators (***S***) for the negative value. That is because the PICTURE did not have a place where the minus sign could be displayed. Since leading zero suppression was *not* used, there were no leading blanks in which to place a minus sign. If your numbers will include negative values, do *not* use all 9's in your PICTURE. Add at least one leading Z or – to the PICTURE.

Below are two additional examples that illustrate special purpose PICTUREs. Notice that when literal text is used heavily, you should normally use "9" as your digit symbol. If you want to display a literal character *before* the first numeric digit (as in the telephone number example below), you *must* use "9" for all of your digit symbols
.

ADDITIONAL PICTURE EXAMPLES		
PICTURE	UNFORMATTED VALUE	FORMATTED VALUE
PIC'(999) 999-9999'	1234567890	(123) 456-7890
PIC'999-99-9999'	123456789	123-45-6789

PICTUREs can be used anywhere that a numeric display format is allowed. Following are a few examples of how PICTUREs can be used in various control statements:

```
COLUMNS: EMPL-NAME  TOTAL-SALES(PIC'ZZZ,ZZZ,ZZ9.99-')
TITLE:   'TELEPHONE DIRECTORY --'  TELEPHONE(PIC'(999) 999-9999')
BREAK:   REGION  FOOTING('TOTAL SALES FOR REGION:'
                         TOTAL-SALES(TOTAL,PIC'$$$,$$$,$$9'))
```

Showing Scaled Numbers with PICTUREs

You can also use a PICTURE to automatically scale the number being formatted. That is, to round the number to thousands, millions, etc. as necessary to make it fit within the PICTURE. The appropriate abbreviation (K, M, G, etc.) indicates what scale the number is shown in.

Scaled PICTUREs allow you to use less space in a report line while still showing approximate values for very large numbers. Look at these two columns of data:

```
                    SALES          SALES
                       26             26
                   48,712           49 K
               5,862,131,092      5,862 M
```

The first column, while showing the exact value of each number, uses up 13 bytes of the report line (even more if you have to allow room for Grand Totals). The second column shows scaled values for the same numbers and only uses 7 bytes. (And the Grand Total would also fit in 7 bytes.)

> **Note:** In order to use the "@" or "?" scaling symbols to request scaling in a PICTURE, you must have specified the **SCALEPIC option** in an (earlier) OPTION statement. Otherwise, those symbols are simply treated as literal text in the PICTURE.

Here is the COLUMNS statement used to format the above columns:

```
Example: OPTION:  SCALEPIC
         COLUMNS: SALES(13) SALES(PIC'Z,ZZ9 @')
```

The "@" in the PICTURE indicates that base-10 scaling (division by factors of 1000) is wanted for that column. (Base-10 scaling is normally used with business and financial data.) The "@" symbol also indicates just where to place the scale abbreviation (K, M, G, etc.).

Scaled PICTUREs can also include decimal digits, if you like:

```
Example: OPTION:  SCALEPIC
         COLUMNS: FILESIZE(13) FILESIZE(PIC'ZZ9.9 @')
```

The above statements result in the following columns.

```
                    SALES          SALES
                       26           26.0
                   48,712         48.7 K
               5,862,131,092       5.9 M
```

You can also request base-2 scaling (division by factors of 1024, or 2 to the 10th power). This type of scaling is often used with data related to computer systems. To specify base-2 scaling, use the "?" scaling symbol instead of "@". If you also add a literal "B" to the PICTURE, you will end up with the abbreviations KB, MB, GB, etc. in the column.

```
Example: OPTION:  SCALEPIC
         COLUMNS: SALES(13) SALES(PIC'Z,ZZ9 ?B')
```

The above statements result in the following columns.

```
                  FILESIZE       FILESIZE
                       26          26  B
                   48,712          48 KB
               5,862,131,092      5,591 MB
```

> **Note:** If the field you are scaling can contain **negative values**, *be sure* to begin the PICTURE with a minus sign, a space or an "extra" Z. If you fail to this, you won't get

a size error (***S***) as with regular PICTUREs. But the negative number will have to be scaled down to a potentially misleading degree. Sometimes all the way down to 0.

Take, for example, this PICTURE which has no extra byte for a minus sign: PIC'ZZ9@'. The number 100,000,000 would format normally as "100M". But the number –100,000,000 appears, surprisingly (at first glance), as " 0G". Spectrum Writer can't show "–100M" in the 4-byte PICTURE. So it has to scale the number down further to –0.1 billion. Rounding that to a whole number (to match the PICTURE) gives 0 billion. That does fit in the PICTURE (" 0G") but is not very useful and could be misleading. Using the correct PIC'–ZZ9@' would give the results you expect for both positive and negative numbers: " 100M" and "–100M".

Note: In most cases, you will want **at least 3 digit positions** in scaled PICTUREs. Otherwise, you can have a similar problem to the one described above (of having your number rounded down to meaninglessness) -- even when all values shown will be positive. Take for example, the following PICTURE with only 2 digits positions: PIC'Z9@'. The positive number 100,000,000 can only be shown as " 0G" in this small PICTURE.

Note: Spectrum Writer sometimes adds **additional leading digit positions** to user's PICTUREs (whether scaled or not). It does that when a long column heading or a column width parm makes a column wider than the width of the PICTURE itself. Usually, the ability to show extra digits is desirable. If, however, you don't want any extra leading digits added to your scaled PICTURE, left-pad your PICTURE with blanks to make it the same size as the column.

For example, if you want to see exactly PIC'Z,ZZ9@' in a column that will be 10 bytes long (because of a long column heading), change the PICTURE to PIC' Z,ZZ9@'.

How PICTUREs Work

This section explains in more detail exactly how PICTUREs are processed.

When a numeric value is being formatted according to a PICTURE, the following process takes place. The PICTURE is evaluated one character at a time, from left to right. Each character in the PICTURE is either:

- a symbol that represents one potential *digit* of the numeric value
- a *literal character* that, under certain conditions, will be moved into the result

The **character 9** in a PICTURE always represents a *digit* from the numeric value. It will be replaced by the appropriate digit of the number, *even if that digit is a leading zero.*

If you want to suppress leading zeros in your result, use one of the following characters to represent leading digits in your PICTURE: **Z**, **$**, **+** or **–**. When one of these characters appears in the PICTURE before the first 9, that character becomes the **leading zero suppression symbol** for the PICTURE. Each occurrence of that symbol will be replaced by the appropriate digit of the number *as long as that digit is not a leading zero*. If the digit is a leading zero, then a blank will appear in that position of the result.

Use the **$ character** for the leading digits in your PICTURE if you want a floating dollar sign to be placed just before the first significant digit in the result.

Use the **+ character** for the leading digits in your PICTURE if you want a floating sign to be placed just before the first significant digit in the result. A plus sign is used for positive numbers; a minus sign is used for negative numbers; no sign is used if the number is zero.

Use the **– character** for the leading digits in your PICTURE if you want a floating minus sign to be placed just before the first significant digit in the result (for negative values). Positive and zero values will have no sign character.

When the **letter Z** is used for the leading digits in your PICTURE, *and no trailing sign symbol appears in the PICTURE*, a floating minus sign is placed before the first significant digit in the result (for negative values).

Use a **+ character** as the *last byte* in your PICTURE if you want a trailing sign (either plus or minus) to be placed in that position of the result.

Use a **– character** as the *last byte* in your PICTURE if you only want a trailing minus sign to be placed in that position of the result (for negative values).

The **letter V** has a special meaning within a PICTURE. It shows where an "understood decimal point" is located. A PICTURE may contain only one V symbol. The V symbol does not take up a byte in the formatted output. (Thus, the result of PIC'99V9' would be just 3 bytes long, not 4.) If a V is present in the PICTURE, all decimal points (.) in the PICTURE are treated as literals and are not used in determining where the decimal digits appear in the result.

The **decimal point (.)** is treated specially within a PICTURE. If the PICTURE contains a V symbol, all decimal points within the PICTURE are just treated as literals. (Thus, the two decimal points in PIC'ZZZ.ZZZ.ZZ9V9' are treated as regular literals.) If no V symbol appears within the PICTURE, a single decimal point is allowed within the PICTURE. It shows where an "explicit decimal point" is to be located in the result.

The **at sign (@)** and **question mark (?)** are used in scaled pictures (page 453) to show where to put the abbreviation for the scale used (K, M, G, etc.). However, these characters only have this special meaning when the SCALEPIC option (page 573) is in effect.

All other characters are treated as *literals*. Literals are moved into the result just as they appear in the PICTURE, with one exception. Any literal that appears *before* the last zero suppression symbol in a PICTURE is blanked out if zero suppression is still in effect at that point. Such literals are only moved to the result if one or more non–zero digits have already been moved to the result. (Thus, the comma literals in PIC'ZZZ,ZZZ,ZZ9.99' are blanked out until after the first digit appears in the result.) Also, *trailing literals* are always moved to the result (even if no non-zero digits were moved.) Trailing literals are those that appear after all of the numeric positions in a PICTURE. They are usually currency indicators (PIC'ZZ9.99 USD') or percentage signs (PIC'ZZ9.9%').

> **Note:** In PICTUREs with *no* zero suppression symbols (such as PIC'(999) 999–9999'), all literals are moved to the result.

The following table summarizes the meaning of each character that can appear in a PICTURE.

> **Note:** A PICTURE may contain symbols representing no more than 31 digits. However, the entire PICTURE text (including literal characters) can be larger than 31 characters.

.

MEANING OF SYMBOLS WITHIN A PICTURE	
SYMBOL	**MEANING**
9	Replace this character with a digit from the numeric value, even if that digit is a leading zero.
Z	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a minus sign for negative numbers (unless the PICTURE contains an explicit trailing plus or minus sign).
$	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a dollar sign. For negative numbers, a minus sign will appear just before the floating dollar sign (unless the PICTURE contains an explicit trailing plus or minus sign).
–	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a minus sign for negative numbers.
+	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain: a plus sign for positive numbers; a minus sign for negative numbers; a blank if the number is zero.
–	*Minus sign, as the last character in a picture.* Specifies that a minus sign should appear in that position if the number is negative. Otherwise, a blank will appear in that position.
+	*Plus sign, as the last character in a picture.* Specifies that: a plus sign should appear in that position if the number is positive; a minus sign should appear in that position if the number is negative; a blank should appear in that position if the number is zero.
V	*Understood decimal point.* This character indicates where the understood decimal point exists within a picture. However, no actual decimal point will appear there. This PICTURE symbol does not affect the size of the formatted result. When this symbol is used, any decimal points (.) in the PICTURE are treated as literals.
.	*When used as an explicit decimal point.* When a PICTURE does not contain a V, this becomes the explicit decimal point. It is displayed as is, unless "leading zero suppression" is still in effect. In that case, a blank will appear in its place.

MEANING OF SYMBOLS WITHIN A PICTURE (CONTINUED)	
SYMBOL	**MEANING**
@	*Only if an earlier SCALEPIC option was processed.* Indicates that the numeric value should be scaled as necessary to fit within the PICTURE. Base-10 scaling (division by factors of 1000) is desired. The "@" symbol also indicates where to put the scale abbreviation (K, M, G, etc.).
?	*Only if an earlier SCALEPIC option was processed.* Indicates that the numeric value should be scaled as necessary to fit within the PICTURE. Base-2 scaling (division by factors of 1024) is desired. The "?" symbol also indicates where to put the scale abbreviation (K, M, G, etc.).
other	Any characters other than those listed above are considered *literal characters* within a picture. These characters will appear in the formatted result just as they are, unless "leading zero suppression" is still in effect. In that case, blanks will appear in their place. Trailing literals (any literal after the last digit position) are always formatted into the result.

Notes:

(1) the first Z, $, + or – character that appears in a picture becomes the "zero suppression symbol" for that picture. Once the zero suppression symbol has been determined for a picture, the other three characters in that set are just treated as literals.

Time PICTUREs

There is also a picture–type display format available for time fields. It is called a TPICTURE ("time picture"). It can also be abbreviated as TPIC and TP. TPICTUREs work similarly to the regular numeric PICTURE. They are a handy way to indicate the number of digits to reserve for the hours portion of very large time values, as well as the number of decimal digits to display. For example, consider the following statement:

```
COLUMNS: TIME-ON-PHONE(TPIC'ZZZ9:99:99.9')
```

The above statement uses a TPIC to specify how the TIME–ON–PHONE field should be displayed. It reserves 4 digits for the hours portion of the time value, and specifies leading zero suppression up until the last hour digit. The TPIC also specifies that 1 decimal digit is wanted in the formatted result. (The main reason for wanting to display more than 2 hour digits is when time *intervals* are being added up and the Grand Total value may be large.)

When formatting times using TPICs, Spectrum Writer treats the time value as a numeric value of the form ...HHHHMMSS.SSSS... That is, the numeric value has 2 digits of seconds, 2 digits of minutes, and an indefinite number of digits for hours. It also contains an indefinite number of decimal digits. The number of digit symbols in the TPIC (characters Z and 9) will determine how many hours digits and decimal digits (if any) are to be displayed.

Conditional Expressions

This section explains:

- how to write **conditional expressions**

Conditional expressions specify one or more conditions. Upon evaluation, a conditional expression will either be true or false. Conditional expressions are used in:

- the INCLUDEIF statement (to specify which records to include in the report)
- the WHEN parm of some COMPUTE statements (to specify when to assign a particular value to a field)

Topics covered in the following sections are:

- how to specify **relation type conditions**
- how to specify **bit field type conditions**
- how to specify **multiple conditions**, by using the keywords AND and OR
- how to **shorten** long conditional expressions
- how to **negate** conditions, using the NOT keyword

Note: Most of the examples used in this section involve fields from the sample EMPL-FILE, described in Appendix F, "Files Used in Examples" (page 648).

In general, a conditional expressions consists of any number of conditions, separated by the keywords AND and OR. You may also use parentheses around groups of conditions to indicate the order in which they should be evaluated. Parentheses may be "nested" to any level. Also, you may precede any condition, or parenthesized group of conditions, with the word keyword NOT, to "negate" the result.

An individual condition can take one of the following two forms:

- a **relation condition**
- a **bit field condition**

CONDITIONAL EXPRESSION SYNTAX

```
condition  [ AND/OR  condition ]  [ AND/OR  condition ]  ...
```

Notes:

- in addition, any number of *paired parentheses* may be used to specify the order of evaluation.
- any condition, or group of conditions in parentheses, may be preceded by the word NOT

Standard Spelling	Symbol Allowed
AND	&
OR	|
NOT	¬

How to Specify a Relation Condition

A relation condition compares the value of two operands, to see if a certain relationship exists between them. Here is an example of a relation condition:

```
TOTAL-SALES > 9000
```

The above condition is true if the value of the TOTAL–SALES field is greater than 9000.

A relation condition consists of two operands separated by a relation operator:

```
operand1  operator  operand2
```

Each **operand** can be either a field or a literal value. The operands can be any of the following types of data (but both operands must be of the *same* type):

- character
- numeric
- date
- time

Note: Bit operands are *not allowed* in relation conditions. A bit operand is a condition all by itself (see page 465).

The relation **operator** may be any of the following:

Relation Operators Allowed in Conditional Expressions	
Relation Operator	**Meaning**
=	"is equal to"
>	"is greater than"
<	"is less than"
>=	"is greater than or equal to"
<=	"is less than or equal to"
¬= or <>	"is not equal to"
¬<	"is not less than"
¬>	"is not greater than"
:	"contains" (for character operands only)
¬:	"does not contain" (for character operands only)

A relation condition is evaluated by comparing the values of the two operands. If the operands have the relation specified by the relation operator, then the condition is *true*. If the operands do not have the relation specified by the relation operator, then the condition is *false*.

Here is another example of a relation condition:

```
SALES-QTR1  >  SALES-QTR2
```

This condition is evaluated by comparing the contents of the SALES–QTR1 field with the contents of the SALES–QTR1 field. If SALES–QTR1 "is greater than" SALES–QTR2 the condition is true. Otherwise, the condition is *false*. For example, if the SALES–QTR1 field contained 4000, and the SALES–QTR2 field contained 3000, then the condition above would be *true*, because 4000 is greater than 3000. However, if the SALES–QTR1 field contained 4000 and the SALES–QTR2 field contained 6000, then the condition would be *false*, because 4000 is not greater than 6000.

Here is an INCLUDEIF statement that uses the condition shown above:

```
INCLUDEIF: SALES-QTR1  >  SALES-QTR2
```

The above statement specifies that only records where the SALES–QTR1 field is greater than the SALES–QTR2 field should be included in the report.

Remember that the operands being compared in a relation condition must be of the same general type of data. That is, numeric operands may only be compared to other numeric operands. Character operands may only be compared to other character operands. Date operands may only be compared to other date operands. And time operands may only be compared to other time operands. (For more information on this, see "Comparing Fields of Different Data Types" on page 463.)

Here is an example of a relation condition that involves date operands:

```
HIRE-DATE <= 1/1/1996
```

The above statement contains a relation condition involving a date field (HIRE–DATE) and a date literal (1/1/96). The condition is true if the HIRE–DATE field "is less than or equal to" January 1, 1996. The condition is false if the HIRE–DATE field contains any date after January 1, 1996.

Here is an example of a relation condition that involves time operands:

```
SALES-TIME > 17:15:48
```

The above statement contains a relation condition involving a time field (SALES–TIME) and a time literal (17:15:48). The condition is true if the SALES–TIME field "is greater than" 17:15:48 (5:15:48 PM). The condition is false if the SALES–TIME field contains a time less than or equal to 17:58:48.

Here is an example of a relation condition involving character data:

```
LAST-NAME = 'SMITH'
```

The above condition is true if the LAST–NAME field is equal to "SMITH".

The list of relation operators on page 461 includes two special operators that can only be used with character type operands. These are the **contains** (:) and the **does not contain** (¬:) operators. Operand1 is said to "contain" operand2 if all of the characters in operand2 appear together somewhere within operand1. Here is an example of a condition that uses the "contains" operator:

```
CUSTOMER : 'INC'
```

The above condition is true if, somewhere within the contents of the CUSTOMER field, the letters "INC" appear together. For example, the condition would be true if the CUSTOMER field in a record contained any of the following values:

- ACME INC
- ABC STORES, INCORPORATED
- BUILDERS INC. OF AMERICA

The same condition would **not be true** when the CUSTOMER field contained any of the following values:

- XYZ CORPORATION
- JOHN BROWN STORES, LTD.
- JONES & ASSOCIATES

Note: When using the "contains" and "not contains" relation operators, operand1 should be *at least as large* as operand2. Otherwise, operand2 could not possibly be contained within operand1.

Comparing Character Operands of Different Lengths

Consider the following conditional expression:

```
LAST-NAME = 'SMITH'
```

In this example a 15–byte character field (LAST–NAME) is compared with a character literal that is only 5 characters long ('SMITH'). When character operands of different lengths are compared, Spectrum Writer first adds enough trailing blanks to the shorter operand to make it the same size as the larger operand. Then the two operands, now of equal length, can be compared byte by byte. Thus, in the example above, Spectrum Writer is actually comparing the LAST–NAME field with a 15–byte character literal, as if the following had been written:

```
LAST-NAME = 'SMITH          '
```

(This addition of trailing blanks *does not actually modify* the value of either of the operands. The blanks are only added to a temporary copy of the operand.)

Comparing Fields of Different Data Types

As mentioned, the operands being compared in a relation condition must be of the same general type of data. That is, numeric operands may only be compared to other numeric operands. Character operands may only be compared to other character operands. Date operands may only be compared with other date operands. And time operands may only be compared with other time operands.

However, this does not mean that the fields being compared must have been defined with the identical *TYPE* parm in their FIELD statement. (The TYPE parm is discussed on page 529.) For example, a PACKED field may be compared to a BINARY field, since both PACKED and BINARY are *numeric* data types. And a MMDDYY type date field may be compared with a P–YYDDD (packed Julian) date field, or with any other kind of date field. Spectrum Writer automatically handles any data type conversion that is necessary.

Even if you find the need to compare operands of different general data types, you may still be able to do that. This can be accomplished by converting one of the operands to a data type compatible with the other operand. The following built–in functions are used to convert an operand from one data type to another. (Built–in functions are described in Appendix D, "Built-In Functions" on page 628)

DATA CONVERSION BUILT-IN FUNCTIONS	
BUILT-IN FUNCTION	**PURPOSE**
#MAKENUM	Converts a character, date or time operand to a numeric value.
#MAKEDATE	Converts a character or numeric operand to a date value.
#MAKETIME	Converts a character or numeric operand to a time value.
#FORMAT	Converts a date, time or numeric operand to a character value.

For example, even though EMPL–NUM is a character field, we can compare it to a numeric literal by first converting it to a numeric value:

```
COMPUTE: NUMERIC-EMPL-NUM = #MAKENUM(EMPL-NUM)
INCLUDEIF: NUMERIC-EMPL-NUM > 100
```

As another example, even though TIME–ON–PHONE is a time field, we can compare it to a numeric literal by first converting it to a numeric value (representing the number of seconds in the time value):

```
COMPUTE: NUMERIC-PHONE-TIME = #MAKENUM(TIME-ON-PHONE)
INCLUDEIF: NUMERIC-PHONE-TIME > 60
```

The above example converts TIME–ON–PHONE from a HH:MM:SS time value to a numeric value equal to the number of seconds in the time value. It then compares this number of seconds with the numeric literal 60.

Conditions Involving Explicit Literals

Normally, when comparing a field with a literal you do not need to know exactly how that field is stored in the input record. Spectrum Writer automatically performs any conversion necessary to make both the field and the literal compatible before comparing them.

As an example, assume that SALARY is a field stored in an input record as a 5–byte packed number. Normally, we would just compare this field to a numeric literal, like this:

```
INCLUDEIF: SALARY = 2345.99
```

When writing the above statement we did not need to know how SALARY was stored in the record. We use a normal numeric literal and let Spectrum Writer take care of the details necessary to make the comparison. The above statement would work whether SALARY was stored in packed, binary, display numeric or any other numeric format.

However, conditions that involve an **explicit hexadecimal literal** (one prefixed with an X) are handled differently. In these cases, no conversion is performed. The field's raw data — just as it is found in the input record — is compared with the literal. This means that when using explicit literals, you must know exactly how a field is stored in the record. You must know how many bytes the field occupies, as well its exact data type.

Consider the following condition that compares SALARY to an explicit hexadecimal literal:

```
INCLUDEIF: SALARY = X'000234599C'
```

This statement is equivalent to the previous statement that used a normal numeric literal. Since SALARY is stored in the input records as a 5–byte packed number, the explicit literal in the above condition also has to be 5 bytes long (10 hexadecimal digits). And the literal also has to be in valid packed format, with a "sign" in the second nibble of the last byte.

One common reason for writing conditions with explicit literals is to compare fields that may have **invalid data**. For example, assume that the input file has some records in it with hex zeros ("low values") in the SALARY field. We want to identify and list those records so that they can be corrected. Since hex zeros is not a valid packed value, there is no way to test for this condition using a normal numeric literal. Instead we have to compare the SALARY field to an explicit hexadecimal literal, like this:

```
INCLUDEIF: SALARY = X'0000000000'
```

As a similar example, assume that we know that some HIRE–DATE fields (in our sample EMPL–FILE) contain spaces rather than a valid character YYMMDD date. The only way to test

for this is to use an explicit literal. You could use either a character or hexadecimal explicit literal:

```
INCLUDEIF: HIRE-DATE = '      '
INCLUDEIF: HIRE-DATE = X'404040404040'
```

The above statements compare the 6–byte HIRE–DATE field to 6 spaces (hexadecimal 40).

How to Specify a Bit Field Condition

The relation condition (described in the preceding sections) is the most common type of condition. The other type of condition is a bit field condition. A bit field condition consists of nothing more than the name of a bit type field:

```
fieldname
```

The condition is considered true if the bit field has a value of "on." The condition is false if the bit field has a value of "off".

Here is an example of a bit field condition:

```
FULL-TIME
```

The above condition is true when the FULL–TIME bit field is "on" (contains a binary 1). The condition is false when the FULL–TIME field is "off" (contains a binary 0).

Here is an INCLUDEIF statement which uses the above bit field condition:

```
INCLUDEIF: FULL-TIME
```

The above statement specifies that only records whose FULL–TIME bit field is "on" should be included in the report.

How to Specify Multiple Conditions

All of the conditional expressions shown so far have contained only a *single condition* (either a relation condition or a bit field condition). Such expressions are called *simple* conditional expressions.

Spectrum Writer, however, allows you to have an *unlimited* number of conditions in a conditional expression. A conditional expression containing more than one condition is called a *complex* conditional expression. Complex conditional expressions consist of two or more conditions separated with the words AND or OR. Parentheses may also be used around groups of conditions to specify the order in which to evaluate the individual conditions.

The following sections explain how to write complex conditional expressions.

Conditional Expressions That Use AND

If all of the conditions in a complex expression are separated by the word AND, then the expression is true *only if all of the conditions are true.*

For example, consider the following expression which has two conditions separated by the word AND:

```
SALES-QTR1 > 3000  AND  HIRE-DATE < 1/1/1997
```

The above conditional expression is true if both of the two conditions are true. That is, the expression is true if the SALES–QTR1 value is greater than 3000 *and* the HIRE–DATE field is less than January 1, 1997.

You may mix relation conditions and bit field conditions in the same conditional expression, as in the following example:

```
SALES-QTR1 > 5000  AND  FULL-TIME
```

For the above conditional expression to be true, the SALES–QTR1 field must be greater than 5000 (a relation condition), and the FULL–TIME bit field must be "on" (a bit field condition).

A conditional expression can have as many conditions as you like. The following example has three conditions, all separated with the word AND:

```
LAST-NAME = 'SMITH'  AND  HIRE-DATE > 1/1/1980  AND  SALES-QTR1 > 10000
```

The above condition would be true if the LAST–NAME field is equal to "SMITH" and the HIRE-DATE field is greater than January 1, 1980 and the SALES–QTR1 field is greater than 10000.

Note: You may use the ampersand symbol (&) in place of the word AND in conditional expressions. For example, the conditional expression shown above could also be written like this:

```
LAST-NAME = 'SMITH'  &  HIRE-DATE > 1/1/1980  &  SALES-QTR1 > 10000
```

Conditional Expressions That Use OR

If all of the conditions in a complex expression are separated by the word OR, then the expression is true *as long as at least one of the conditions is true*.

Consider a conditional expression using the same two conditions as shown in an earlier example, but separated this time with the word OR instead of AND.

```
SALES-QTR1 > 3000  OR  HIRE-DATE < 1/1/1997
```

The conditional expression is now true if *either* the SALES–QTR1 field is greater than 3000, *or* if the HIRE–DATE field is less than January 1, 1997.

You may mix relation conditions and bit field conditions in the same conditional expression, as in the following example:

```
SALES-QTR1 > 5000  OR  FULL-TIME
```

For the above conditional expression to be true, either the SALES–QTR1 field must be greater than 5000 (a relation condition), or the FULL–TIME bit field must be "on" (a bit field condition).

A conditional expression can have as many conditions as you like. The following example has three conditions, all separated with the word OR:

```
LAST-NAME = 'SMITH'  OR  LAST-NAME = 'JONES'  OR  SALES-QTR1 > 10000
```

The above condition would be true if the LAST–NAME field was equal to either "SMITH" or "JONES", or if the SALES–QTR1 field was greater than 10000.

Note: You may use the vertical bar (|) in place of the word OR in conditional expressions. For example, the conditional expression shown above could also be written like this:

```
LAST-NAME = 'SMITH'     LAST-NAME = 'JONES'     SALES-QTR1 > 10000
```

Conditional Expressions That Use Both AND and OR

You may use both the word AND and the word OR in a single conditional expression. When this is done, parentheses are normally used to indicate the order in which the conditions should be evaluated. For example:

```
(LAST-NAME = 'JONES'  OR  LAST-NAME = 'SMITH')  AND  SALES-QTR1 > 5000
```

In the above expression, parentheses are used around the two conditions that are separated by the word OR. That indicates that these conditions should be evaluated first. If the LAST-NAME is equal to either "JONES" or "SMITH", then the parenthesized expression is true. Otherwise it is false. For the entire conditional expression to be true, this parenthesized result must be true; *and* the remaining condition (SALES–QTR1 > 5000) must be true. In other words, the parentheses cause the entire expression to be true if: the LAST–NAME is either "JONES" or "SMITH", and the SALES–QTR1 value is greater than 5000.

Now, consider what would happen if the parentheses are used around the AND conditions, like this:

```
LAST-NAME = 'JONES'  OR  (LAST-NAME = 'SMITH'  AND  SALES-QTR1 > 5000)
```

Again, the conditions enclosed in parentheses are evaluated first. In this case, the parenthesized expression is true only if LAST–NAME equals "SMITH" and SALES–QTR1 is greater than 5000. The entire expression is then true, if *either* the LAST–NAME equals "JONES", *or* if this parenthesized result is true. In other words, the above expressions is true if: the LAST–NAME equals "JONES", or if both of the following are true: the LAST–NAME equals "SMITH" and the SALES–QTR1 value is greater than 5000.

Note: If both the words AND and OR are used in an expression, and parentheses are *not used* to specify evaluation order, the conditions connected by AND will be evaluated before those connected by OR. However, it is always best to use parentheses in such expressions, so that there is no question or confusion about the order of evaluation.

How to Shorten Long Expressions

When one operand is being compared to more than one value in a conditional expression, you may write that expression in a shorter form. For example, consider the following:

```
LAST-NAME = 'JONES'  OR  LAST-NAME = 'SMITH'  OR  LAST-NAME = 'BROWN'
```

The expression above is true if the LAST–NAME field is equal to any of the three character literals ('JONES', 'SMITH', or 'BROWN'). Since all three relation conditions have the **same first operand**, you are allowed to omit that operand after specifying it the first time. You could specify the same conditional expression this way:

```
LAST-NAME = 'JONES'  OR =  'SMITH'  OR = 'BROWN'
```

Here are the rules for shortening expressions. You remember that the format of a relation condition is:

```
operand1  operator  operand2
```

> **Rule:** When two or more consecutive conditions have the same *operand1*, you may omit that operand after the first condition. Thus, whenever operand1 is not specified in a condition, the most recently specified operand1 will be used.

The conditional expression shown earlier contains three conditions, each separated with the word OR. Those three conditions are:

- LAST–NAME = 'JONES'
- = 'SMITH'
- = 'BROWN'

The first condition is written out fully, containing two operands and a relation operator.

The second condition contains no operand1. It just has an operator and operand2. Therefore, the most recently specified operand1 (LAST–NAME, from the previous condition) will be used as operand1 in the second condition.

The same thing applies to the third condition, which also lacks an operand1.

We can actually simplify the conditional expression even further. Since the second and third conditions also use the **same relation operator** as the first condition (namely, "="), we can omit that operator from those conditions as well:

```
LAST-NAME = 'JONES'  OR  'SMITH'  OR  'BROWN'
```

> **Rule:** When two or more consecutive conditions have the same *operand1 and the same relation operator*, you may omit those items after the first condition. Thus, whenever neither operand1 nor a relation operator is specified in a condition, the most recently specified operand1 and the most recently specified relation operator will be used.

Here is an example that combines the two forms of simplification:

```
SALES-QTR1 = 1000  OR  2000  OR  < 500
```

The above conditional expression contains three relation conditions, separated with the word OR. The three conditions are:

- SALES–QTR1 = 1000

- 2000
- < 500

The first condition is written out fully, containing two operands and a relation operator. The second condition does not contain an operand1 nor a relation operator, so SALES–QTR1 and "=" are assumed (from the previous condition). The third condition does not contain an operand1, but does contain a relation operator ("<"). So only operand1 (SALES–QTR1) is assumed. The above conditional expression is the same, then, as the following one:

```
SALES-QTR1 = 1000  OR  SALES-QTR1 = 2000  OR  SALES-QTR1 < 500
```

Here is one more example of a shortened conditional expression:

```
LAST-NAME ¬= 'SMITH'  AND  'JONES'  AND  'BROWN'
```

The above conditional expression is true if the LAST–NAME field is not equal to "SMITH" and is not equal to "JONES" and is not equal to "BROWN". In other words, the expression is true if the LAST–NAME contains anything other than those three names. The above statement is processed as if it were written like this:

```
LAST-NAME ¬= "SMITH"  AND  LAST-NAME ¬= "JONES"  AND  LAST-NAME ¬= "BROWN"
```

How to Negate Conditions

This section explains:

- how to use the word **NOT** (or ¬ symbol) in conditional expressions

You may precede any condition with the word NOT to negate the result of its evaluation.

For example, consider the following relation condition:

```
SALES-QTR1 > 2000
```

The above condition would be true if SALES–QTR1 contained 8000, since 8000 is greater than 2000. However, we could negate that condition like this:

```
NOT SALES-QTR1 > 2000
```

Now, the conditional expression would be *false* when SALES–QTR1 contained 8000. That is because the condition SALES–QTR1 > 2000 which is true, is *negated* by the preceding NOT.

You may also negate a *bit field* condition. For example:

```
NOT FULL-TIME
```

The above conditional expression is *true* when bit field condition is false, that is, when the FULL–TIME bit field is "off".

You may also negate a group of conditions in parentheses, as in this example:

```
NOT (SALES-QTR1 > 2000 AND HIRE-DATE < 1/1/1997)
```

The conditional expression above is now *true* whenever the complex condition within parentheses is *false*.

Note: You may use the not symbol (¬) in place of the word NOT in conditional expressions. For example, the preceding conditional expression could also be written like this:

```
¬ (SALES-QTR1 > 2000 AND HIRE-DATE < 1/1/1997)
```

Examples of Conditional Expressions

Case 1. This example compares the contents of **two numeric fields**.

```
INCLUDEIF: SALES-QTR2 > SALES-QTR1
```

The above statement would include all records where the SALES–QTR2 field was greater than the SALES–QTR1 field.

Case 2. This example compares the contents of a numeric field with a **numeric literal**.

```
INCLUDEIF: TOTAL-SALES < 1000
```

This example would include all records where the TOTAL–SALES field was less than 1000.

Case 3. Here is an example of comparing a **date field** with a date literal.

```
INCLUDEIF: HIRE-DATE < 6/1/1990
```

This example would include all records where the HIRE-DATE field was less than (earlier than) June 1, 1990.

Case 4. Here is an example of comparing a **time field** with a time literal.

```
INCLUDEIF: SALES-TIME >=  14:00:00
```

This example would include all records where the SALES–TIME field was greater than or equal to 14:00:00 (2 o'clock PM). Since the seconds in this example are zero, we could also write:

```
INCLUDEIF: SALES-TIME >=  14:00
```

Case 5. Here is an example of comparing a **character field** with a character literal.

```
INCLUDEIF: LAST-NAME = 'JONES'
```

This example would include all records where the LAST-NAME field equalled JONES (that is, the 5 letters JONES, followed by 10 blanks). Notice that character literals must be enclosed in either quotes or apostrophes. Numeric, date and time literals are not enclosed in quotes or apostrophes.

When character operands of different lengths are compared, Spectrum Writer temporarily pads the shorter operand with right–hand blanks before making the comparison.

Case 6. This example **scans a character field** to see if a certain text is contained anywhere within the field.

```
INCLUDEIF: CUSTOMER : 'CORP'
```

This example would select all records where the letters "CORP" appeared together anywhere within the CUSTOMER field. Records with customer names such as "ABC CORP", "CORPORATION OF AMERICA", and "ACME, INCORPORATED" would be selected using this example.

Case 7. This example bases the decision to include records on **more than one comparison**.

```
INCLUDEIF: SEX = 'F'  AND
           HIRE-DATE >= 1/1/1986  AND  <= 12/31/1986
```

This example would select all records where the SEX field contained "F", and the HIRE-DATE field was greater than or equal to January 1, 1986 and was less than or equal to December 31, 1986. In other words, the records included would be for all female employees hired sometime in 1986.

Case 8. Here is another example of **multiple comparisons**.

```
INCLUDEIF:  PRODUCT-CODE = '801'  OR  '802'  OR  >= '900'
```

This example would select all records where the PRODUCT–CODE field contained any of the following:

- 801
- 802
- any value greater than or equal to 900

Case 9. Here is another example that uses **multiple comparisons**.

```
INCLUDEIF: REGION = 'NORTH' AND
           (LAST-NAME = 'JONES' OR 'SMITH' OR 'BROWN')
```

This example would select all records where the REGION field was equal to "NORTH", and the LAST-NAME field was any one of the following:

- JONES
- SMITH
- BROWN

Case 10. This example checks whether a **bit field** is ON or OFF. The following statement will include only those records where the PART-TIME bit field is ON.

```
INCLUDEIF: PART-TIME
```

And the following statement would select all records where the PART-TIME bit field is OFF.

```
INCLUDEIF: NOT PART-TIME
```

Case 11. Here is an example of comparing the contents of a field to a **literal hexadecimal value**.

```
INCLUDEIF: DATE1 = X'000000'  OR  SALARY = X'FFFFFFFF'
```

When comparing a field to a hexadecimal literal, no data conversion is performed on the field at all. The comparison will be made against the data just as it exists in the input record. When a hexadecimal comparison is made to a field whose value is the result of a user data exit, the comparison will be made against the result passed to Spectrum Writer by the data exit. Hexadecimal comparisons are not allowed to "computed" fields (since they do not exist in a real input record).

As with regular character literals, when a hexadecimal literal is compared with a field of a different length, Spectrum Writer pads the shorter operand with right–hand *blanks* (not hex zeros) before making the comparison. This blank padding is done regardless of the data type of the field.

Computational Expressions

This section explains:

- how to write computational expressions

Computational expressions are used to specify a *value*. They are used in the COMPUTE statement to specify the value to assign to "compute" fields. A computational expression might be nothing more than a single field name (or literal). Or, it might be dozens of lines long and involve many mathematical operations. The syntax for a computational expression follows.

COMPUTATIONAL EXPRESSION SYNTAX

```
operand  [ operator operand ]  [ operator operand ]  ...
```

Notes:

- in addition, any number of *paired parentheses* may be used to specify the order of operations.

Only the first operand is required. You may specify as many additional operator/operand pairs as you like. In general, the data type of the first operand (character, numeric, date, time or bit) determines the data type of the entire expression. All subsequent operands must be of the same data type. Also, only the operators supported for that data type may be used in the expression.

Note: There is one exception to the rule that all operands in a computational expression must be of the same data type. For time computational expressions, the

operands may be either time values or numeric values. Numeric values are treated as being a number of seconds. Thus, the following COMPUTE statement adds 1 minute (60 seconds) to the time value in SALES–TIME:

```
COMPUTE: NEW-TIME = SALES-TIME + 60
```

Operands in Computational Expressions

An operand in a computational expression specifies a data value. An operand can be any of the following:

- a **literal** value. (See "How to Write Literals" on page 448.)
- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)
- a **built–in field** (a complete list of built–in fields is found in Appendix C, "Built-In Fields" on page 624).
- a **built–in function's result** (a complete list of built–in functions is found in Appendix D, "Built-In Functions" on page 628)

Operators in Computational Expressions

An operator in a computational expression specifies an operation to perform on the operands. The operators allowed in a particular expression will depend on the data type of the expression. For character, numeric, and time expressions, the following table shows the operators that are supported. (No operators are supported for date and bit expressions.)

Operators Allowed in Computational Expressions	
Character Operators	**Numeric and Time Operators**
+ (concatenation)	+ (addition)
	– (subtraction)
	* (multiplication)
	/ (division)

Note: Be sure to use one or more blanks both *before* and *after* the subtraction operator (**–**) in computational expressions. This is required because the same symbol is valid as a character within field names. The following:

```
ABC-XYZ
```

would be considered the name of a single field, named ABC–XYZ. However, the following:

```
ABC  -  XYZ
```

would be considered a subtraction operation, where field XYZ is subtracted from field ABC. For the other operators (**+**, ***** and */*), blanks are not required around the symbol, but are allowed.

Note: The standard numeric operations are also allowed in computational expressions for time values. When performing these operations, Spectrum Writer first converts each time value into a numeric value (equal to the total number of seconds in the time value). The operations are then performed on these numeric values. The final result is then converted back into a HH:MM:SS[.SSS...] time value.

Note: While no date operators are directly supported, it is still possible to perform certain manipulation of date fields. Use the #MAKENUM built–in function (page 636) to convert a date field to a numeric value. You can then add or subtract (days) to this numeric value. Then, use the #MAKEDATE built–in function (page 640) to convert the modified numeric value back to a date field. An example of this is shown on page 475.

Order of Operations

Operations within parentheses are performed first. If nested parentheses are encountered, the most deeply nested operations are performed first. When parentheses are not used, or for operations at the same level of parentheses, the order of operations is as follows:

- multiplications and divisions are performed first
- additions and subtractions are performed afterwards

Operations of equal priority are performed left to right.

Examples of Computational Expressions

Case 1. Here is an example of a COMPUTE statement with a **character** type computational expression:

```
COMPUTE: X = 'AAA' + 'BBB'
```

In the above example, the second operand ("BBB") is concatenated to (or, "appended to") the first operand "AAA". The new field X would contain the value "AAABBB".

Case 2. Following is an example of a **numeric** computational expression:

```
COMPUTE: YEARLY-SALES =
           SALES-QTR1  + SALES-QTR2  + SALES-QTR3  + SALES-QTR4
```

The above example computes the yearly sales total by adding the four quarterly sales fields together.

Case 3. Following is an example of using **parentheses** within a computational expression to indicate the order of operation:

```
COMPUTE: PERCENT-CHANGE(DIVTOTS) =
           ((SALES-QTR2 - SALES-QTR1) * 100) / SALES-QTR1
```

The above example computes the percentage change between the second quarter sales figure and the first quarter sales figure. The computational expression first subtracts SALES–QTR1 from SALES–QTR2, since that is the most deeply embedded operation. That difference is then multiplied by 100. The resulting product is then divided by SALES–QTR1, giving the percentage change.

> **Note:** The DIVTOTS parm tells Spectrum Writer not to simply total the values of this field for the Grand Totals line (or control break total lines). Totalling percentages often does not give a meaningful result. Instead, the DIVTOTS parm tells Spectrum Writer to "divide totals" — that is, divide the total value of the numerator by the total value of the denominator when printing total lines. For more information on the DIVTOTS parm, see "Computing True Percentages and Ratios at Control Breaks" (page 202).

Case 4. Following is an example of using a **numeric built–in function** in a computational expression:

```
COMPUTE: ABS-PERCENT-CHANGE = #ABS(PERCENT-CHANGE)
```

The above example uses the numeric built–in function #ABS ("absolute value"). The percentage change computed in the preceding case might be either a positive or a negative number. The #ABS function returns the absolute value (that is, the positive value) of its parm (the PERCENT–CHANGE field, in this example). The new field (ABS–PERCENT–CHANGE) now contains the percentage change as a positive value.

Case 5. You may **embed computational expressions within most built–in functions**. For example, we could have defined the ABS–PERCENT–CHANGE field all in one computational expression by using an imbedded expression within the #ABS function:

```
COMPUTE: ABS-PERCENT-CHANGE =
           #ABS(((SALES-QTR2 - SALES-QTR1) * 100) / SALES-QTR1)
```

Case 6. There are no operators supported for **date fields**. Therefore, computational expressions for these types of fields consists only of a single operand. For example:

```
COMPUTE: START-DATE = 1/1/1995
```

The above example simply assigns the literal date 1/1/1995 to the new field START–DATE.

Case 7. The single operand in a date expression may also be a date type *field*, or a **date type built–in function**. For example:

```
COMPUTE: DUE-DATE = #MAKEDATE(#MAKENUM(SALES-DATE) + 10)
```

The above example computes the DUE–DATE field by adding 10 days to the SALES–DATE. It does this by first converting the SALES–DATE field to a number, then adding 10 to that number, and finally converting this sum back into a date field.

The above COMPUTE statement could also be separated into three statements, perhaps making it easier to understand:

```
COMPUTE: NUM-SALES-DATE = #MAKENUM(SALES-DATE)
COMPUTE: NUM-DUE-DATE   = NUM-SALES-DATE + 10
COMPUTE: DUE-DATE       = #MAKEDATE(NUM-DUE-DATE)
```

Case 8. There are no operators supported for **bit** fields. Bit expressions can consist only of a single operand. That operand may be either another bit type field, or a bit type built–in function (such as #ON and #OFF). For example:

```
COMPUTE: TRUE-BIT = #ON
```

The above example defines a new bit type field named TRUE–BIT, whose value is ON.

Chapter 10. Control Statement Syntax

Chapter Table of Contents

Chapter 10. Control Statement Syntax 477

Syntax Notation 478
ASM Statement 479
BREAK Statement 481
COBOL Statement 493
COLUMNS Statement 498
COMPUTE Statement 506
COPY Statement 516
FIELD Statement 521
FILE Statement 531
FOOTNOTE Statement 538
INCLUDEIF Statement 540
INPUT Statement 542
NEWOUT Statement 554
OPTIONS Statement 555
READ Statement 578
SORT Statement 595
TITLE Statement 602

Chapter 10. Control Statement Syntax

This chapter contains the complete syntax information for each Spectrum Writer control statement. The statements appear in alphabetical order.

Syntax Notation

In the syntax boxes throughout this chapter, the following conventions are used.

CONVENTIONS USED IN SYNTAX BOXES	
STYLE	**MEANING**
lowercase	Items in lower case letters represent values to be supplied by the user.
uppercase	Items in UPPER CASE letters must be typed exactly as they appear. (However, valid abbreviations are also accepted.)
brackets	Items within [square brackets] are optional.
ellipsis	An ellipsis (...) indicates that the preceding item(s) may be repeated any number of times.
underline	Underlined items indicate the default value that will be used if no other value is specified.
slash	Slashes (/) indicate mutually exclusive items. One and only one of the items separated by slashes may be specified.

ASM Statement

PURPOSE

Specifies that an Assembler language record layout follows. Spectrum Writer processes the Assembler record layout and creates "internal" FIELD statements corresponding to the Assembler fields in the record layout. This lets you define the fields in a file by using an Assembler record layout, rather than writing FIELD statements.

Also use this statement to have Spectrum Writer convert an Assembler record layout into FIELD statements and write those FIELD statements to an output file.

Beginning immediately after the ASM statement (and any of its continuation lines) Spectrum Writer treats input lines as Assembler code. The Assembler code is assumed to end when the next Spectrum Writer control statement prefix is encountered. The only exception is that Spectrum Writer COPY statements may be imbedded in the Assembler code and do not end the scope of the ASM statement.

FEATURES

Use the ASM statement to:

- specify that an **Assembler record layout** follows
- specify whether to **print or write out FIELD statements** that correspond to the Assembler record layout
- specify **various options** that affect the way the Assembler code is processed

LEARNING MORE

The complete syntax of the ASM statement is shown in the following box. A description of the individual parms is found under the similar COBOL statement on page 493. In addition, the following parts of the manual relate to the ASM statement:

- the use of Assembler record layouts to define input files is discussed beginning on page 369

SYNTAX

ASM STATEMENT SYNTAX

```
ASM:    [ COLUMN[(ALL)]/DISP[(ALL)                                        ]
        [ FILE(filename/*)                                                ]
        [ MAXOCCURS(nnnnn/100)                                            ]
        [ NOSEQ                                                           ]
        [ OUTATTR(type,'dlbl/tlbl'[,SYSnnn][,80][,blksize])   (VSE only) ]
        [ OUTDDN(ddname)                                      (OS/390 only)]
        [ RELOC                                                           ]
        [ SHOWFLDS(YES/NO)                                                ]
        [ STARTCOL(nnnnn)/STARTDISP(nnnnn)                                ]
```

Standard Spelling	Alternate Spellings
COLUMN	COL
NO	N
YES	Y

No parms are required. The parms may appear in any order. For a description of the parms, see under the COBOL statement (page 493) which uses the same parms.

BREAK Statement

PURPOSE

Specifies that a control break should occur whenever the value of a certain field changes. Only sort fields may be used to create control breaks— that is, a field may be named in a BREAK statement only if it has also appeared in a preceding SORT statement.

The BREAK statement is also used to customize the **Grand Totals**.

For **summary reports** (where no individual detail lines are printed), the BREAK statement determines how the summary lines will look.

You may have more than one BREAK statement in a report. The use of multiple BREAK statements is discussed on page 204.

> **Note:** The *SORT statement* can also be used to request many control breaks. The SORT statement can specify: which fields to break on; the control break spacing to use; and, which, if any, of the statistical lines should print at a break. You must use the BREAK statement, however, if you want to print footing lines, heading lines, or customized statistical lines at a control break.

FEATURES

Use the BREAK statement to:

- specify control **break spacing** (whether to skip to a new page or print a number of blank lines at a control break)
- specify one or more customized **footing lines** to print at the end of a control group
- specify whether or not to print a **total line** at the end of a control group
- specify whether or not to print other **statistical lines** (such as averages, maximums, minimums) at the end of a control group
- customize the **text** used in the total and other statistical lines
- specify one or more customized **heading lines** to print at the beginning of a control group, and optionally at the top of all subsequent pages for that control group
- specify how the **Grand Total lines** should look
- **suppress total lines** at control breaks

LEARNING MORE

The complete syntax of the BREAK statement is shown on the following pages. In addition, the following parts of the manual relate to the BREAK statement:

- a lesson on using the BREAK statement in reports begins on page 65
- a lesson on using the BREAK statement in PC files begins on page 108
- advanced uses of the BREAK statement are discussed beginning on page 177
- using the BREAK statement to produce summary reports is discussed beginning on page 73
- using the BREAK statement to produce summary PC files is discussed beginning on page 113
- customizing the Grand Totals with a BREAK statement is discussed beginning on page 207

SYNTAX

BREAK STATEMENT SYNTAX

```
BREAK:      fieldname/#GRAND
        [   AVERAGE[(print-expression)]                                  ]
        [   FOOTING(print-expression) ...                                ]
        [   HEADING(print-expression) ...                                ]
        [   MAXIMUM[(print-expression)]                                  ]
        [   MINIMUM[(print-expression)]                                  ]
        [   NZAVERAGE[(print-expression)]                                ]
        [   NZMINIMUM[(print-expression)]                                ]
        [   REPEAT                                                       ]
        [   SPACE(n/PAGE/PAGE1/NEWSHEET/NEWSHEET1/ODDPAGE/ODDPAGE1)      ]
        [   TOTAL[(print-expression)]/NOTOTAL                            ]
```

Note: the syntax for the print-expressions is shown on page page 488.

Standard Spelling	Alternate Spellings
AVERAGE	AVER, AVG
BREAK	BRK
FOOTING	FOOT
HEADING	HEAD
MAXIMUM	MAX
MINIMUM	MIN
NOTOTAL	NOTOT, NOTOTALS, NOTOTS
NZAVERAGE	NZAVER, NZAVG
NZMINIMUM	NZMIN
PAGE	PG, P
SPACE	SPC
TOTAL	TOT, TOTALS, TOTS

The fieldname is *required* in a BREAK statement, and must be the *first* item after the statement prefix. All other parms are optional and can appear in any order on the BREAK statement.

fieldname/#GRAND

Identifies the control break field. Whenever the contents of this field changes, a control break will occur in the report or PC file. This field must have been specified as a sort field in a preceding sort statement.

You may also specify #GRAND rather than an actual field name. Using #GRAND allows you to specify control break options for the Grand Totals "control break" (page 207).

```
Example: BREAK:  REGION
```

The above example specifies that a control break should occur whenever the REGION field changes value. Since no other parms are specified, default processing will take place at the break: a line of region totals will print, followed by 2 blank lines.

```
Example: BREAK: #GRAND  AVERAGE
```

The above statement specifies that an average line is wanted at the Grand Totals "control break." The average line will print after the Grand Total line at the end of the report.

AVERAGE[(print–expression)]

Specifies that each numeric column's average value should print at the control break, and optionally can specify how the average line should look. The default is *not* to print averages at each break. If you simply specify the AVERAGE parm, a default average line will print at the control break. It will begin with the following text:

```
*** AVERAGE VALUE
```

After the above text, the average values themselves will print, lined up under the numeric columns of the report. If you would like the average line to begin with some other text, specify a print expression with the AVERAGE parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the AVERAGE parm is discussed on page 186.

```
Example: BREAK: REGION  AVERAGE
```

The above example causes a default average line to print whenever the REGION field changes value.

```
Example: BREAK: REGION  AVERAGE('AVERAGES FOR'  REGION)
```

The above example specifies that the average line should begin with the text "AVERAGES FOR xxxxx" (where xxxxx is the value of the REGION field).

FOOTING(print–expression)

Specifies a print line to print at the end of a control group. The print line may contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. You may have as many FOOTING parms as you like. The footing lines will print in the order in which they appear in this statement. The first footing line will print immediately after the last regular detail line in the control group and before the total line, if any. The syntax of the print expression for this parm is shown on page 488. The use of the FOOTING parm is discussed on page 188.

```
Example: BREAK: REGION  FOOTING('END OF REGION'  REGION)
```

The above example causes a line that reads "END OF REGION xxxxx" to print whenever the REGION field changes (where xxxxx is the value of the REGION field).

```
Example: BREAK: REGION NOTOTALS
           FOOTING('TOTAL AMOUNT=' AMOUNT(TOTAL) 'AVERAGE AMOUNT=' AMOUNT(AVERAGE))
```

The above example prints a single line that shows the AMOUNT field's total value and average value for the control group. (The standard total line is suppressed with the NOTOTALS parm.)

HEADING(print–expression)

Specifies a print line to print at the beginning of a control group. The print line may contain any combination of literal text and data from input files. You may have as many HEADING parms as you like. The heading lines will print in the order in which they appear in this statement. The syntax of the print expression for this parm is shown on page 488. The use of the HEADING parm is discussed on page 200. Specifying the REPEAT parm (in the BREAK statement) causes all of the HEADING lines to also be repeated at the top of all subsequent pages for that control group. These heading lines print after the column headings.

```
Example: BREAK: REGION  HEADING('REGION'  REGION  'FOLLOWS')
```

The above example causes a line that reads REGION xxxxx FOLLOWS to print whenever a new REGION is about to start printing (where xxxxx is the value of the REGION field).

MAXIMUM[(print–expression)]

Specifies that each numeric column's maximum value should print at the control break, and optionally can specify how the maximum line should look. The default is *not* to print maximums at each break. If you simply specify the MAXIMUM parm, a default maximum line will print at the control break. It will begin with the following text:

```
*** MAXIMUM VALUE
```

After the above text, the maximum values themselves will print, lined up under the numeric columns of the report. If you would like the maximum line to begin with some other text, specify a print expression with the MAXIMUM parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the MAXIMUM parm is discussed on page 186.

```
Example: BREAK: REGION  MAXIMUM
```

The above example causes a default maximum line to print whenever the REGION field changes value.

```
Example: BREAK: REGION  MAXIMUM('MAXIMUMS FOR'  REGION)
```

The above example specifies that the maximum line should begin with the text "MAXIMUMS FOR xxxxx" (where xxxxx is the value of the REGION field).

MINIMUM[(print–expression)]

Specifies that each numeric column's minimum value should print at the control break, and optionally can specify how the minimum line should look. The default is *not* to print

minimums at each break. If you simply specify the MINIMUM parm, a default minimum line will print at the control break. It will begin with the following text:

```
*** MINIMUM VALUE
```

After the above text, the minimum values themselves will print, lined up under the numeric columns of the report. If you would like the minimum line to begin with some other text, specify a print expression with the MINIMUM parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the MINIMUM parm is discussed on page 186.

```
Example: BREAK: REGION  MINIMUM
```

The above example causes a default minimum line to print whenever the REGION field changes value.

```
Example: BREAK: REGION  MINIMUM('MINIMUMS FOR'  REGION)
```

The above example specifies that the minimum line should begin with the text "MINIMUMS FOR xxxxx" (where xxxxx is the value of the REGION field).

NZAVERAGE[(print–expression)]

Specifies that each numeric column's average value (*not considering zero values*) should print at the control break, and optionally can specify how the non–zero average line should look. (Non–zero averages are useful if missing data — zero values — is throwing off a column's average.) The default is *not* to print non–zero averages at each break. If you simply specify the NZAVERAGE parm, a default non–zero average line will print at the control break. It will begin with the following text:

```
*** AVERAGE OF NON-ZERO VALUES
```

After the above text, the non–zero averages themselves will print, lined up under the numeric columns of the report. If you would like the non–zero average line to begin with some other text, specify a print expression with the NZAVERAGE parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the NZAVERAGE parm is discussed on page 186.

```
Example: BREAK: REGION  NZAVERAGE
```

The above example causes a default non–zero average line to print whenever the REGION field changes value.

```
Example: BREAK: REGION  NZAVERAGE('NON-ZERO AVERAGES FOR'  REGION)
```

The above example specifies that the non–zero average line should begin with the text "NON–ZERO AVERAGES FOR xxxxx" (where xxxxx is the value of the REGION field).

NZMINIMUM[(print–expression)]

Specifies that each numeric column's minimum value (*not considering zero values*) should print at the control break, and optionally can specify how the non–zero minimum line should look. The default is *not* to print non–zero minimums at each break. If you simply

specify the NZMINIMUM parm, a default non–zero minimum line will print at the control break. It will begin with the following text:

```
*** MINIMUM OF NON-ZERO VALUES
```

After the above text, the non–zero minimums themselves will print, lined up under the numeric columns of the report. If you would like the non–zero minimum line to begin with some other text, specify a print expression with the NZMINIMUM parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the NZMINIMUM parm is discussed on page 186.

```
Example: BREAK: REGION  NZMINIMUM
```

The above example causes a default non–zero minimum line to print whenever the REGION field changes value.

```
Example: BREAK: REGION  NZMINIMUM('NON-ZERO MINIMUMS FOR'  REGION)
```

The above example specifies that the non–zero minimum line should begin with the text "NON–ZERO MINIMUMS FOR xxxxx" (where xxxxx is the value of the REGION field).

REPEAT

Specifies that all heading lines (defined in HEADING parms in the BREAK statement) should be repeated at the top of each new page of a control group (following the titles and column headings). Otherwise, the heading lines print only once, at the beginning of the control group.

```
Example: BREAK: REGION  REPEAT
                HEADING('REGION' REGION 'FOLLOWS')
                HEADING('====================')
```

The above example specifies two heading lines for the REGION control break. In addition to printing at the beginning of each new control group (which may occur in the middle of a page), the heading lines will also be repeated at the top of each subsequent page for that control group.

SPACE(n/PAGE/PAGE1/NEWSHEET/NEWSHEET1/ODDPAGE/ODDPAGE1)

Specifies the type of spacing desired at the control break, after any footing lines, total lines and statistics lines have printed. If no SPACE parm is specified, the default is to print 2 blank lines. A description of each SPACE option is shown in the following table.

SPACE PARM VALUES

SPACING OPTION	DESCRIPTION
n	Skips this number of **blank lines**.
PAGE	Skips to the top of the **next page** of the report.
PAGE1	Works like PAGE, but also resets the page number to "one".

SPACE PARM VALUES (CONTINUED)	
SPACING OPTION	**DESCRIPTION**
NEWSHEET	Skips to a **new sheet** of paper. In order for this feature to work, you must also use the OPTIONS statement's PRTSHEET parm to specify a character string that can be sent to your printer to tell it to skip to a new sheet of paper. (The PRTSHEET option is described on page 572.)
NEWSHEET1	Works like NEWSHEET, but also resets the page number to "one".
ODDPAGE	Skips to the next **odd numbered** page. This parm accomplishes the same thing as the NEWSHEET parm, but can be used even if you do not have a character string to send to the printer to force it to skip to a new sheet. However, for this option to work you must ensure that the first page of your report prints on the front side of a sheet of paper. As long as page 1 of your report prints on the front side of a sheet of paper, all other odd numbered pages will also be on front sides.
ODDPAGE1	Works like ODDPAGE, but also resets the page number to "one".

```
Example: BREAK:  REGION  SPACE(PAGE1)
```

The above example requests that the report skip to a new page whenever the REGION field changes value. Page numbering will also start over with page one for each new region.

TOTAL[(print–expression)]/NOTOTAL

Specifies whether or not to print totals at the control break, and optionally can specify how the total line should look. The default is to print totals at each break. Specifying NOTOTAL suppresses the total line at a control break.

By default, total lines begin with the following text:

```
*** TOTALS FOR xxxxxxx  (n,nnn ITEMS)
```

After the above information, the actual total values print, lined up under the numeric columns of the report. If you would like the total line to begin with some other text, specify a print expression within the TOTAL parm. The print expression can contain any combination of literal text, data from input files, and certain control-group-wide statistics for numeric and time fields. The syntax of the print expression is shown on page 488. The use of the TOTAL parm is discussed on page 182.

```
Example: BREAK: REGION  NOTOTAL
```

The above example specifies that a control break should occur whenever the REGION field changes value. However, no total line should print. (Spectrum Writer will just print two blank lines and continue to the next region.)

```
Example: BREAK: REGION  TOTAL('TOTALS FOR'  REGION)
```

The above example specifies that a total line should print and begin with the text "TOTALS FOR xxxxx" (where xxxxx is the value of the REGION field).

PRINT EXPRESSION SYNTAX

print–expression

Specifies how to build one print line that will print at the control break. The syntax for a print expression within a BREAK statement parm is similar to print expressions used in other statements. There are, however, some additional features that can be used in BREAK statement print expressions. These include additional built–in fields, and certain control-group-wide statistical parms to use with numeric and time fields. The complete syntax for a print expression within a BREAK statement follows. BREAK statement print expressions are discussed beginning on page 188.

PRINT–EXPRESSION SYNTAX (IN BREAK STATEMENT)

A **print–expression** consists of one or more **items**, optionally separated by numeric **spacing factors**:

```
[n] item [n] item [n] item ...
```

Each **item** can be either a **fieldname** or a **literal text**. Each item can optionally be followed by a parm list in parentheses:

```
fieldname[(  [   ASCII                                          ]
             [   BIZ                                            ]
             [   display-format                                 ]
             [   LEFT/CENTER/RIGHT                              ]
             [   TOTAL/AVERAGE/MAXIMUM/MINIMUM/NZAVERAGE/NZMINIMUM   ]
             [   width                                          ]   )]

'literal'[(     width                                              )]
```

(continued on next page)

PRINT–EXPRESSION SYNTAX (IN BREAK STATEMENT) (Continued)

Standard Spelling	Alternate Spellings
AVERAGE	AVER, AVG
CENTER	CJ
LEFT	LJ
MAXIMUM	MAX
MINIMUM	MIN
NZAVERAGE	NZAVER, NZAVG
NZMINIMUM	NZMIN
RIGHT	RJ
TOTAL	TOT

fieldname

Specifies that the print line should contain the contents of this field. For all print expressions that print at the *end* of a control group, the field's data is taken from the *last record* in the control group (unless a statistical parm is specified for the field). For heading print expressions (which print at the *beginning* of a control group), the data is taken from the *first record* in the control group that follows.

The field must be available to Spectrum Writer at the time the BREAK statement is processed. That is, the field name must be one of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)
- a **built–in** field. (See Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields.)

Notice that several of the built–in fields listed in Appendix C are exclusively for use in the BREAK statement. These fields may be used in any BREAK statement print expression *except* within the HEADING parm. (The use of these special fields is discussed on page 198.) The special built–in fields are:

BUILT-IN FIELDS ALLOWED ONLY IN THE BREAK STATEMENT (NOT ALLOWED IN THE HEADING PARM)

BUILT-IN FIELD	TYPE	DESCRIPTION
#ITEMS	Numeric	Contains the number of items (records) included in the control group that has just ended.
#ITEM–ENDING	Character	Contains either the letter "S", or a blank, depending on the value of #ITEMS. When #ITEMS equals one, #ITEM–ENDING is a blank. Otherwise, #ITEM–ENDING is an "S".
#COUNTER	Numeric	Contains the cumulative number of items (records) that have been processed up through the control group just ended. This field is like #ITEMS, except that it is *not* reset to zero at every control break.

```
Example: BREAK: REGION
                FOOTING(#ITEMS 'ITEM' 0 #ITEM-ENDING 'IN REGION' REGION)
```

The print expression in the above example uses a combination of literals, fieldnames and built–in fieldnames. It also contains one spacing factor. The resulting footing line would print when there is only one record in the control group:

```
1 ITEM  IN REGION xxxxx
```

The same statement causes a footing line like the following to print, when the control group contains more than one record. Notice that the word "ITEMS" is now plural.

```
2 ITEMS IN REGION xxxxx
```

A spacing factor of 0 is used to prevent a blank space from appearing between the literal text "ITEM" and the contents of the field #ITEM–ENDING. Without the spacing factor, the footing line would say "2 ITEM S", rather than "2 ITEMS".

'literal'

Specifies that the print line should contain this literal text.

Example: (See the example above under the fieldname parm. The FOOTING parm print expression in that example uses the literal texts "ITEM" and "IN REGION".)

n

This is a numeric spacing factor. It specifies how many blank spaces to leave between two items in the print line. A spacing factor of zero is allowed. (It results in two items appearing in the print line with no blank spaces between them.) If no spacing factor is given, the default is to leave one blank space between items.

Example: (See the example above under the fieldname parm. A spacing factor of 0 is used in that example.)

ASCII

Specifies that the final, formatted field should be converted from EBCDIC to ASCII in the print line. See page 143 for more information on creating ASCII output files.

Note: To specify your own EBCDIC-to-ASCII translation table, use the ASCIITABLE option in the OPTIONS statement (page 558). Otherwise, Spectrum Writer uses a default translation table.

```
Example: COMPUTE: BREAK-LIT = 'TOTALS FOR REGION '
         BREAK:   REGION NOTOTALS
           FOOTING(BREAK-LIT(ASCII) 0 REGION(ASCII) 0 X'20' 0 AMOUNT(TOTAL, ASCII))
```

The above example shows how to print an ASCII line containing a literal text, the contents of the REGION field, and the total value of the AMOUNT field at a control break. An ASCII space (X'20') will appear between each field. (To get the first space, we included a trailing blank in the COMPUTE field literal. We specified the second ASCII space directly in the FOOTING parm.) Without the zero spacing factors between items, an EBCDIC blank would have been inserted between items.

BIZ

This "blank if zero" parm specifies that blanks should appear in the print line for the field if it has a value of zero. This parm is allowed only for numeric, date and time fields. A date is considered to have a zero value if the month, day and last 2 digits of the year are all zeros (regardless of the value of the century part of the year).

```
Example: BREAK: HIRE-DATE
                FOOTING('END OF EMPLOYEES HIRED ON' HIRE-DATE(BIZ))
```

The above example causes the HIRE–DATE field in the footing line to be left blank whenever it contains a zero date.

display–format

Specifies how a field should be formatted in the print line. A complete list of display formats is found in Appendix B, "Display Formats" (page 617). If this parm is not specified, Spectrum Writer will use the display format from:

- the FIELD or COMPUTE statement that defined the field
- an OPTIONS statement FORMAT parm
- the default display format shown in the table on page 618

```
Example: BREAK: HIRE-DATE
                FOOTING('END OF EMPLOYEES HIRED ON' HIRE-DATE(LONG1))
```

The above example causes the HIRE–DATE field in the footing line to be spelled out in LONG1 format:

```
END OF EMPLOYEES HIRED ON MAY 1, 1995
```

LEFT/CENTER/RIGHT

Specifies how the data should be justified within the space allocated for it in the print line. If none of these parms is specified, no justification is performed.

```
Example: BREAK: HIRE-DATE
                FOOTING('END OF EMPLOYEES HIRED ON' HIRE-DATE(LONG1,RIGHT))
```

The above example also displays the HIRE–DATE field (in LONG1 format) in the footing line. Dates displayed in LONG1 format are allocated 18 characters in a print line (in order to print long dates like "SEPTEMBER 31, 1999"). The RIGHT parm causes the contents of the HIRE–DATE field to be right–justified within its 18–character area in the print line.

```
END OF EMPLOYEES HIRED ON        MAY 1, 1995
```

TOTAL/AVERAGE/MAXIMUM/MINIMUM/NZAVERAGE/NZMINIMUM

Allowed only for numeric and time fields. Specifies that a statistical value for a field should appear in the print line, rather than the field's contents from an individual record. (These statistical parms may *not be* used in HEADING print expressions.) When none of these parms is specified, the contents of a field will be taken from the *last* record in the control group (for print lines that appear at the end of a control group). If one of these parms is specified, then the control group *total* (or *average*, *maximum*, etc.) will appear in the print line instead. The use of these parms is illustrated in the section beginning on page 188.

```
Example: BREAK: REGION
                FOOTING('LARGEST SALE IN REGION WAS'  AMOUNT(MAXIMUM))
                FOOTING('AVERAGE SALE IN REGION WAS'  AMOUNT(AVERAGE))
```

The above example causes two footing lines to print at the end of a control group. The first footing line will display the control group's maximum AMOUNT value. The second footing line will show the control group's average AMOUNT value.

width

This numeric parm specifies the number of characters to reserve for an item in the print line. Use this parm if the default width is too large or too small.

```
Example: BREAK: REGION
                FOOTING(#ITEMS(11) 'ITEM' 0 #ITEM-ENDING 'IN REGION' REGION)
```

The above example causes 11 characters to be reserved for printing the number of items (#ITEMS) in the footing line:

```
nnn,nnn,nnn ITEMS IN REGION xxxxx
```

COBOL Statement

PURPOSE

Specifies that a Cobol record layout follows. Spectrum Writer processes the Cobol record layout and creates "internal" FIELD statements corresponding to the Cobol fields in the record layout. This lets you define the fields in a file by using a Cobol record layout, rather than writing FIELD statements.

Also use this statement to have Spectrum Writer convert a Cobol record layout into FIELD statements and write those FIELD statements to an output file.

Beginning immediately after the COBOL statement (and any of its continuation lines) Spectrum Writer treats input lines as Cobol code. The Cobol code is assumed to end when the next Spectrum Writer control statement prefix is encountered. The only exception is that Spectrum Writer COPY statements may be imbedded in the Cobol code and do not end the scope of the COBOL statement.

FEATURES

Use the COBOL statement to:

- specify that a **Cobol record layout** follows
- specify whether to **print or write out FIELD statements** that correspond to the Cobol record layout
- specify **various options** that affect the way the Cobol code is processed

LEARNING MORE

The complete syntax of the COBOL statement is shown on the following pages. In addition, the following parts of the manual relate to the COBOL statement:

- the use of Cobol record layouts to define input files is discussed beginning on page 369

SYNTAX

COBOL STATEMENT SYNTAX

```
COBOL:  [ COLUMN[(ALL)]/DISP[(ALL)]                                        ]
        [ FILE(filename/*)                                                 ]
        [ MAXOCCURS(nnnnn/100)                                             ]
        [ NOSEQ                                                            ]
        [ OUTATTR(type,'dlbl/tlbl' [,SYSnnn] [,80] [,blksize]) (VSE only)  ]
        [ OUTDDN(ddname)                                       (OS/390 only)]
        [ RELOC                                                            ]
        [ SHOWFLDS(YES/NO)                                                 ]
        [ STARTCOL(nnnnn)/STARTDISP(nnnnn)                                 ]
```

Standard Spelling	Alternate Spellings
COLUMN	COL
NO	N
YES	Y

No parms are required. The parms may appear in any order. Note that the ASM statement also uses most of these same parms.

COLUMN[(ALL)]/DISP[(ALL)]

Specifies whether the COLUMN parm or the DISP parm should be used in the FIELD statements that Spectrum Writer creates from the Cobol record layout. (This parm is only meaningful if you also specify the SHOWFLDS(YES) parm and/or the OUTDDN/OUTATTR parm.) If neither COLUMN nor DISP is specified, the COLUMN parm will be used whenever necessary in the FIELD statements created. If ALL is specified with either parm, the COLUMN or DISP parm will be present in *all* of the FIELD statements created. If ALL is not specified, the COLUMN or DISP parm will appear only in those FIELD statements where it is necessary (that is, in FIELD statements that define fields out of the normal sequence).

The ALL parm may be useful if you're having problems using a new record layout. Specify DISP(ALL) to see the displacement that Spectrum Writer has assigned to each field. Then compare these displacements with those printed in the Data Map section of an actual Cobol compilation of the same record layout. This may help you locate the source of the error.

```
Example: COBOL: DISP
```

The above statement specifies that the FIELD statements printed in the control listing or written to an output file will use DISP parms (rather than COLUMN parms). The DISP parm will only be present in FIELD statements that define fields out of the normal order.

```
Example: COBOL: COLUMN(ALL)
```

The above statement specifies that the COLUMN parm (rather than the DISP parm) should be used in FIELD statements printed or written out. All FIELD statements will have a COLUMN parm.

FILE(filename/*)

Specifies the file to which the fields defined by the record layout belong. An asterisk indicates the current file (which is the default). The current file is the file named in the most recent FILE statement.

```
Example: COBOL: FILE(EMPL-FILE)
```

The above statement specifies that the fields defined by the Cobol record layout belong to the EMPL–FILE (rather than the current file).

MAXOCCURS(nnnnn/100)

Specifies the maximum number of occurrences for which individual field definition is necessary. This applies only to items having an OCCURS clause (in Cobol) or a repetition factor (in Assembler). By default, up to 100 occurrences of each such item are defined as individual fields. If your record layout has a field with a large number of occurrences and you need to be able to reference all of these occurrences *individually*, specify a MAXOCCURS parm with a sufficiently large value. However, if you do not need to address such fields individually, it will save memory and processing time to leave the default in effect. In extreme cases (with many thousands of occurrences) creating an internal field definition for each occurrence may require more memory than is available in the region (or partition) and an "out of memory" abnormal end could occur.

Specifying MAXOCCURS(0) means that *all* occurrences of each array should be defined individually.

Note: See the section beginning on page 377 for more information on how the individual fields in an array are named.

```
Example: COBOL: MAXOCCURS(2000)
```

The above statement will cause up to 2000 individual fields to be defined for each array in the record layout. (With the ASM statement, it will cause up to 2000 individual fields to be defined for each item defined with a repetition factor.)

NOSEQ

Valid only for the COBOL statement. Specifies that numeric checking of Cobol sequence numbers should not be performed. Spectrum Writer normally performs this checking to help detect a Cobol record layout that is not formatted correctly and which may result in wrong field definitions. Use this parm if the Cobol record layout you use has non–numerics in columns 1 through 6 and you do not want warning messages to appear in the control listing.

Note: When NOSEQ is not specified, Spectrum Writer prints warning messages for the first five sequence number errors encountered.

```
Example: COBOL: NOSEQ
```

The above statement specifies that Spectrum Writer should not examine the contents of columns 1 through 6 of the Cobol record layout.

OUTATTR(type, 'dlbl/tlbl' [,SYSnnn] [,80] [,blksize])

VSE only. Specifies that FIELD statements corresponding to the record layout be written to the specified output file. The output file must be defined as a fixed length file with 80–byte records. The blocksize may be any multiple of 80. The OUTATTR parm describes various attributes of the desired output file. The allowed values within the OUTATTR parm are:

SUBPARMS ALLOWED IN THE OUTATTR PARM

SUBPARM	MEANING
type	This parm is required. It tells Spectrum Writer what kind of device to write the FIELD statements to. It must be one of the following values: **DASD** a SAM file on a DASD device (disk). Use DASD (rather than VSAM) for VSAM–managed SAM files. **TAPE** a SAM file on a magnetic tape **VSAM** an ESDS VSAM file
'dlbl/tlbl'	This parm is required. It tells Spectrum Writer what DLBL or TLBL is used in the JCL for the output file. The 1- to 7-byte name within apostrophes (or quotation marks) must be the same as the filename in a DLBL or TLBL statement in the execution JCL.
SYSnnn	This parm is required for TAPE output. It is treated as a comment for other output types. It identifies the logical unit to write the output to. The value specified here must also be "assigned" in the JCL.
80	This parm is optional. It specifies the length of the output records to be written. If specified, it must be 80, which is also the default.
blksize	This parm is optional. It specifies the block size to use when writing a DASD or TAPE output file. (This parm is not allowed for VSAM output types.) This value must be a multiple of 80. If omitted, single record blocking is used.

```
Example: COBOL: OUTATTR(DASD,'FLDOUT')
```

The above statement specifies that FIELD statements should be written to the disk output file identified by the FLDOUT DLBL statement in the execution JCL.

OUTDDN(ddname)

OS/390 only. Specifies that FIELD statements corresponding to the record layout should be written to an output file identified by this DDNAME in the execution JCL. The output file must be defined as a fixed length file with 80–byte records. The blocksize may be any multiple of 80.

```
Example: COBOL: OUTDDN(FLDOUT)
```

The above statement specifies that FIELD statements should be written to the output file identified by the FLDOUT DD statement in the execution JCL.

RELOC

Specifies that any FIELD statements that are printed or written out should be "relocatable" whenever possible. This option may make it easier for you to modify your Spectrum Writer file definition when a record layout changes. That is, you may be able to insert new FIELD statements without having to change all of the FIELD statements following the new one. When RELOC is specified, Spectrum Writer attempts to use fieldnames, rather than numbers, in the FIELD statements' COLUMN/DISP parm whenever possible.

```
Example: COBOL:  RELOC  OUTDDN(FLDOUT)
```

The above statement specifies that the FIELD statements written to the FLDOUT DD should be made as relocatable as possible.

SHOWFLDS(YES/NO)

Specifies that FIELD statements corresponding to the record layout should be printed in the control listing. This is especially useful when working with a new record layout. It allows you to see the names Spectrum Writer has assigned to each field (including the names of individual items within arrays, and items that were renamed to make them unique). The listing also shows the data type of each field (character or numeric).

```
Example: COBOL: SHOWFLDS(YES)
```

The above statement specifies that FIELD statements corresponding to the Cobol record layout should be printed in the control listing.

STARTCOL(nnnnn)/
STARTDISP(nnnnn)

Specifies the column (or displacement) to be used for the first item in the record layout that follows. If not specified, the first field in the record layout will start in the "current" location for the file it belongs to. Thus, if there were no earlier FIELD statements for the file, the first field from the record layout will begin in column 1. If there were some earlier FIELD statements, the first record layout field will begin immediately after the field defined in the last FIELD statement for the file.

In Cobol layouts, this starting column/displacement will also be used for the first item in any subsequent 01 level implicit (or explicit) redefines. In Assembler layouts, this starting column/displacement will also be used for the first item in any subsequent DSECT.

```
Example: COBOL: STARTCOL(251)
```

The above statement specifies that the first field defined by the Cobol record layout begins in column 251. Any subsequent record layouts starting with a 01 level item will also begin in column 251.

COLUMNS Statement

PURPOSE

This statement determines what columns of data the report or PC file will have. Each field named in this statement will result in one column of data in the output. These columns will appear in the same order as the field names appear in the COLUMNS statement.

Also use the COLUMNS statement to specify column headings and other formatting details.

You may have any number of COLUMNS statements per run. Each COLUMNS statement results in one detail line in the report or PC file. A request with *no* COLUMNS statement will have no detail lines in the output.

FEATURES

Use the COLUMNS statement to:

- specify the **columns** (of data fields or of literal texts) desired in the report or output file
- specify the **column headings** to be used in the report or PC file
- specify how many **blank spaces** should appear between each column in reports
- specify a column's **width**
- specify how to **format** the data within a column. (For example, should a numeric field be displayed with or without commas? Should leading zeros be printed or not? Should a date field be printed as MM/DD/YY or should the name of the month be spelled out completely, etc.)
- specify how to **justify** the data within a column (left, center, or right)
- specify that **repeating values** should be blanked out
- specify which numeric columns should be **totalled** at control breaks and at the Grand Total

LEARNING MORE

The complete syntax of the COLUMNS is shown on the following pages. In addition, the following parts of the manual relate to the COLUMNS statement:

- a lesson on using the COLUMNS statement in reports begins on page 34
- a lesson on using the COLUMNS statement in PC files begins on page 88
- advanced uses of the COLUMNS statement are discussed beginning on page 125
- the use of multiple COLUMNS statements is discussed beginning on page 151

SYNTAX

COLUMNS STATEMENT SYNTAX

```
COLUMNS:  print-expression
```

Note: the syntax for the print-expression is shown on page page 500.

Standard Spelling	Alternate Spellings
COLUMNS	COLUMN, COLS, COL

The contents of the COLUMNS statement is simply a print expression. It is also valid to have an *empty* COLUMNS statement. An empty COLUMNS statement results in a blank detail line in the report or PC file.

PRINT–EXPRESSION SYNTAX (IN COLUMNS STATEMENT)

A **print–expression** consists of one or more **items**, optionally separated by numeric **spacing factors**:

```
COLUMNS:  [n] item [n] item [n] item ...
```

Each **item** can be a **fieldname**, a **record name** or a **literal text**. Each item can optionally be followed by a parm list in parentheses:

```
fieldname[(  [   ACCUM/NOACCUM              ]
             [   ASCII                      ]
             [   BIZ                        ]
             [   display-format             ]
             [   'heading1|heading2...'     ]
             [   LEFT/CENTER/RIGHT          ]
             [   NOREPEAT/NOREPEATPAGE      ]
             [   width                      ]   )]

record-name/#COMPUTES[(
             [   exclude-field1 exclude-field2 ...]
             [   INNER/OUTER                ]
             [   BYDEF/BYNAME/BYCOL         ]
             [   LIST/NOLIST                ]   )]

'literal'[(  [   'heading1|heading2...'     ]
             [   width                      ]   )]
```

Standard Spelling	Alternate Spellings
#COMPUTES	#COMPUTE
ACCUM	ACC
BYCOL	#BYCOL
BYDEF	#BYDEF
BYNAME	#BYNAME
CENTER	CJ
COLUMNS	COLUMN, COLS, COL
INNER	#INNER
LEFT	LJ
LIST	#LIST
NOACCUM	NOACC
NOLIST	#NOLIST
NOREPEAT	NOREP
NOREPEATPAGE	NOREPPAGE
OUTER	#OUTER
RIGHT	RJ

Description of Print-Expression Items

fieldname

Names a field that should appear as a column in the report or PC file. The field must be available to Spectrum Writer at the time the COLUMNS statement is processed. That is, the field must be one of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement).
- a **built–in** field. (See Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields.)

```
Example: COLUMNS: LAST-NAME  HIRE-DATE  TOTAL-SALES
```

The above example specifies that the report (or PC file) should contain three columns. The fields displayed in the columns will be LAST–NAME, HIRE–DATE, and TOTAL–SALES.

record-name/#COMPUTES

Specifying a record-name in the COLUMNS statement causes a column to be added for *every* field in that record, as well as for every COMPUTE field that is part of the same file definition. Specifying the keyword #COMPUTES causes a column to be added for every COMPUTE field that is not part of any file's definition. The use of record names in the COLUMNS statements is discussed on page 158.

```
Example: OPTION:  PC
         INPUT:   SALES-FILE
         COLUMNS: SALES-FILE
```

The three statements above would reformat the entire contents of the SALES-FILE into a comma-delimited "PC" file.

'literal'

Specifies that the report should have a column displaying this literal text. (Enclose the text in either apostrophes or quotation marks.) This feature is especially useful when multiple COLUMNS statements are used. A literal text at the beginning of each line serves to identify the data on that line. A column with a literal text (such as dashes) can also be used to print a "blank" column in a report, to be filled in by hand.

```
Example: COLUMNS: '1ST QUARTER'  SALES-QTR1
         COLUMNS: '4TH QUARTER'  SALES-QTR4
```

The above example produces a report with two detail lines per input record. In each line, the first column will contain literal text, and the second column will contain a sales figure. The first column in each line identifies which quarter's data is displayed in the second column. (See page 152 for a similar report example.)

```
Example: COLUMNS: LAST-NAME  TELEPHONE 'NEW TELEPHONE: _________'
```

The above example produces a report with three columns. The first two contain the contents of fields (last name and the current telephone number). The third contains the literal text

```
NEW TELEPHONE: _________
```

which provides an area that can be filled in by hand on the hardcopy report. (See page 127 for a similar report example.)

n

This is a numeric spacing factor. It specifies how many blank spaces to leave between two report columns. (Spacing factors are not used in PC files.) A spacing factor of zero is allowed if you want *no* spaces between two columns. If no spacing factor is given, the default is to leave one blank space between columns. The use of spacing factors is discussed on page 128.

```
Example: COLUMNS: LAST-NAME  7  HIRE-DATE
```

The above example specifies that 7 blank spaces should be left between the LAST–NAME column and the HIRE–DATE column in the report.

> **Note:** To change the default spacing factor between *all* columns, use the COLSPACE parm of the OPTIONS statement (page 560).

Description of Parms

ACCUM/NOACCUM

This parm is valid only for numeric and time fields. It specifies whether a column should be accumulated or not. Columns that are accumulated will appear in the totals line, as well as in any other statistics lines that have been requested (such as the average line, the maximum line, etc.) Columns that are not accumulated will not appear in the totals and statistics lines.

By default, Spectrum Writer accumulates all **numeric fields**, with one exception. Numeric fields that are displayed using a PICTURE which contains special characters are not accumulated. (Special characters include such things as parentheses, imbedded dashes, asterisks, etc.) By default, numeric fields displayed with such a PICTURE are *not accumulated* and therefore do not appear in the total line and other statistical lines.

By default, **time fields** are *not* accumulated. Specify ACCUM if you want to see totals for a time field. This might be desired for time fields that contain *durations*, rather than times of day.

If an ACCUM or NOACCUM parm is specified in the COLUMNS statement, it overrides any such parm that may have been specified in the FIELD or COMPUTE statement used to define the field. The use of the ACCUM and NOACCUM parms is discussed on page 148.

```
Example: COLUMNS: EMPL-NAME  AMOUNT(NOACCUM)  TIME-ON-PHONE(ACCUM)
```

The above example specifies that the AMOUNT column in the report should *not* be accumulated. Therefore, that column will not appear in the Grand Totals, or in control break totals. On the other hand, the time field named TIME–ON–PHONE *will* be accumulated. Therefore, it will appear in the Grand Totals and in control break totals.

ASCII

Specifies that the final, formatted field should be converted from EBCDIC to ASCII in the print line. To specify your own EBCDIC-to-ASCII translation table, use the ASCIITABLE option in the OPTIONS statement (page 558). Otherwise, Spectrum Writer uses a default translation table. See page 143 for more information on creating ASCII output files.

```
Example: OPTIONS: COLSEP(X'20')
         COLUMNS: REGION(ASCII) SALES-DATE(ASCII) AMOUNT(ASCII)
```

The above example causes the REGION, SALES-DATE and AMOUNT fields to be formatted in ASCII. The COLSEP option specifies that an ASCII blank (X'20') should be used to separate the columns (rather than the default EBCDIC blank, which is X'40').

BIZ

This "blank if zero" parm specifies that a column should be left blank if the field has a value of zero. This parm is allowed only for numeric, date and time fields. A date is considered to have a zero value if the month, day and last 2 digits of the year are all zeros (regardless of the value of the century part of the year).

```
Example: COLUMNS:  REGION  SALES-DATE(BIZ)  SALES-TIME(BIZ)  AMOUNT(BIZ)
```

The above example specifies that the SALES-DATE, SALES-TIME and AMOUNT columns should be left blank when their respective fields contain zero values.

BYDEF/BYNAME/BYCOL

Used with a record name to specify the order in which the columns for that record's fields should appear. The default is BYDEF, which means the order in which the fields were defined. You can specify BYNAME to put the columns in alphabetical order. Or specify BYCOL to put them in starting column order (that is, the order in which they occur in the input record).

```
Example: COLUMNS: SALES-FILE(BYCOL)
```

display–format

Specifies how the contents of a field should be formatted in a report. A complete list of display formats is found in Appendix B, "Display Formats" (page 617).

If you do not specify a display format in the COLUMNS statement, Spectrum Writer chooses a default display format. This will be:

- the display format (if any) specified when the field was defined (in a FIELD or COMPUTE statement)
- the display format (if any) specified in a previous OPTIONS statement's FORMAT parm (see page 562). Use the FORMAT option if you want to change the default way that *all* dates, times or numbers in your report are formatted.
- the default display format shown in the table on page 618.

PC Note: Display formats should not normally be used when creating PC files. Spectrum Writer chooses the display format needed to create an import file for the PC program specified in the OPTIONS statement. After importing your PC file into a PC spreadsheet, you can use the PC program's features to change the way dates or numbers are formatted.

```
Example: COLUMNS: LAST-NAME  HIRE-DATE(LONG1)  TOTAL-SALES(PIC'$$$,$$9')
```

The above example uses display formats for two of the columns. The HIRE–DATE field will be displayed in the LONG1 format (that is, with the month name spelled out). The TOTAL–SALES field will be formatted using a floating dollar sign, and will print whole dollars only — no decimal digits. The use of this parm is discussed on page 137.

exclude-field1, exclude-field2, ...

Used with a record name to exclude fields from the COLUMNS statement. Any field named in the parms after a record name will *not* have a column in the report or output file.

```
Example: COLUMNS: SALES-FILE(BACKUP-EMPL-NUM COMMISSION-RATE TIME-ON-PHONE)
```

The above statement would write out all fields from the SALES-FILE except for the three fields named in the parms as exclude fields.

'heading1|heading2...'

Specifies the column heading to use for an item in a report or PC file. Enclose the column heading text in either apostrophes or quotation marks. If you need to use that same character (an apostrophe or quotation mark) within the text, use two of those characters for each character desired.

Use a vertical bar (|) to separate the column heading text into separate lines. It is not necessary to add your own "padding" spaces in order to make the column heading texts stack neatly in your report. Spectrum Writer automatically centers each part of the column heading for you

See page 130 for more information on column headings. A list of special options related to the column headings appears on page 133.

```
Example: COLUMNS: LAST-NAME('EMPLOYEE|NAME')  '____________'('NEW TELEPHONE')
```

The above example specifies column headings for both columns. The column heading for the LAST–NAME field will be "EMPLOYEE" on the first line, and "NAME" on the second line. Even though the two texts are different lengths, they will be correctly centered over the report column. The column that just contains literal underscores will have a column heading that says "NEW TELEPHONE" on a single line.

> **Note:** You may use the HDGSEP parm of the OPTIONS statement to select a character other than the vertical bar (|) to use as the separator character for column heading texts.

If you **do not want** any column headings for a particular column, specify a blank column heading text. To suppress even the column heading underscores, specify a null column heading text.

```
Example: COLUMNS:  LAST-NAME(' ')  HIRE-DATE('')
```

The above statement specifies that neither the LAST–NAME column nor the HIRE–DATE column should have columns headings. The width of the LAST–NAME column will still be indicated by a number of underscores in the column heading. The HIRE–DATE column will not even have underscores over it.

If a column heading text is not specified in the COLUMNS statement, Spectrum Writer uses the column headings specified when the field was defined (in a FIELD or COMPUTE statement). If no columns headings were specified when the field was defined, Spectrum Writer uses the field name itself as the column heading. The field name will be broken apart at each dash or underscore, with each part of the name going onto a separate heading line.

INNER/OUTER

Used with a record name to specify which field to exclude from the output when two or more fields overlap in the input record. Consider this definition of a date field, with a redefinition of its component parts:

```
FIELD: SALES-DATE TYPE(YYMMDD)
FIELD: SALES-YY   LEN(2) COLUMN(SALES-DATE)
FIELD: SALES-MM   LEN(2)
FIELD: SALES-DD   LEN(2)
```

The SALES-DATE field is considered the "outer" field. The last three fields are all "inner" fields, since they are within another field. By default, Spectrum Writer will create an output column for all four fields. Specify the **OUTER parm** if you want to exclude outer fields when overlaps occur. Specify the **INNER parm** to exclude inner fields when overlaps occur.

```
Example: COLUMNS: SALES-FILE(INNER)
```

The above statement would result in SALES-YY, SALES-MM and SALES-DD being excluded from the report or output file. SALES-DATE will be in the output.

LEFT/CENTER/RIGHT

Specifies how the data should be justified within a column. If none of these parms is specified, no justification is performed. The use of these parms is discussed on page 146.

```
Example: COLUMNS: LAST-NAME(CENTER)  HIRE-DATE(LONG1,RIGHT)
```

The above example specifies that the names printed in the LAST–NAME column should be centered within the column. The HIRE–DATE column (in LONG1 format) will be right-justified.

LIST/NOLIST

Used with a record name to specify whether or not to list the individual fields that will be output from the record. By default, the fields will be listed. Specify NOLIST to suppress the list from the control statement listing.

```
Example: COLUMNS: SALES-FILE(NOLIST)
```

NOREPEAT/NOREPEATPAGE

These parms specify that "repeated" values should not be printed in the report or PC file. The NOREPEAT parm blanks out repeated values except at the top of each new page and at the beginning of each new control group. The NOREPEATPAGE parm blanks out repeated values except at the top of each new page. The use of these parms is discussed beginning on page 144.

```
Example: COLUMNS: REGION(NOREPEAT)  EMPL-NAME  SALES-DATE  CUSTOMER  AMOUNT
```

width

This is a numeric parm that specifies the number of characters to reserve for a particular column in a report or PC file. Use this parm if the default column width is larger or smaller than you desire. The use of the width parm is discussed on page 135.

```
Example: COLUMNS: LAST-NAME   TOTAL-SALES(20)
```

The above example specifies that 20 bytes should be reserved for printing the TOTAL-SALES column in the report. This might be needed if the sales figures were very large and the default column width was not big enough to display all of the digits.

COMPUTE Statement

PURPOSE

Defines a new field that can be **computed** from other values. You may use arithmetic operations, string operations and built–in functions to compute the value of the new field. You may also use logical conditions to determine what value to assign to a field.

A computed field can be used in any way that a field from an actual file may be used. That is, you can print it in a report column or title, output it to a PC file, sort on it, break on it, total it, compare it to other fields, and even use it to compute additional new fields.

FEATURES

Use the COMPUTE statement to:

- define a **new field** using a name of your choice
- specify one or more **computational expressions** to use in assigning a value to that field
- specify certain **conditions** that should be evaluated to determine what value to assign to the new field.
- specify that control break totals and Grand Totals for this field should be computed by performing a **group–wide division** rather than merely summing its individual values (DIVTOTS parm)
- specify the **column heading** to use when the field appears in a report or PC file
- specify the **display format** to use when displaying the field
- specify whether or not the field should be **accumulated** (and thus appear in the Grand Total line, etc.)
- specify the **size** of a character field
- specify the number of **decimal places** to be retained in a numeric or a time field

LEARNING MORE

The complete syntax of the COMPUTE statement is shown on the following pages. In addition, the following parts of the manual relate to the COMPUTE statement:

- a lesson on using the COMPUTE statement in reports begins on page 46
- a lesson on using the COMPUTE statement in PC files begins on page 98
- use of the DIVTOTS parm in the COMPUTE statement is discussed beginning on page 202

- a complete list of built-in fields available for use in COMPUTE statements appears in Appendix D, "Built-In Functions" (page 628).
- suggestions on writing COMPUTE statements for maximum CPU efficiency are given in Appendix G, "Speed-Up Tips" (page 652).

SYNTAX

COMPUTE STATEMENT SYNTAX

```
COMPUTE:  fieldname[( parms )]  = computational-expression
```

or

```
COMPUTE:   fieldname[( parms )] =
           WHEN(conditional-expression)ASSIGN(computational-expression)
       [   WHEN(conditional-expression)ASSIGN(computational-expression)]
       [   WHEN(conditional-expression)ASSIGN(computational-expression)]
           ...
       [   ELSE                   ASSIGN(computational-expression)/RETAIN]
```

The **parms** available are:

```
ACCUM/NOACCUM
display-format
DIVTOTS
'heading1|heading2...'
nnn
```

Standard Spelling	Alternate Spellings
ACCUM	ACC
ASSIGN	ASS
COMPUTE	COMP
DIVTOTS	DIVTOT, DT
NOACCUM	NOACC
WHEN	WH

Values are assigned to computed fields each time a new primary input file record is read. (Or, in certain cases, each time a new logical input record is assembled.)

There are two forms of the COMPUTE statement. A **simple COMPUTE statement** contains a single computational expression. Each time a new primary input record read, the specified computation is performed and the result is assigned to the computed field.

A **conditional COMPUTE statement** may contain *multiple* computational expressions. It will also contain one or more conditional expressions. Each time a new primary input record is read, one of the following actions will be taken:

- a computational expression from one of the **ASSIGN parms** will be used to assign a value to the computed field, or
- the previous value of the computed field will be **retained**, or
- a **default value** will be assigned to the computed field.

The action taken depends on the conditions contained in the conditional expressions in the WHEN parms. Spectrum Writer evaluates each of the WHEN expressions, in the same order in which they are written. As soon as a WHEN expression is found that is "true," the corresponding ASSIGN expression is calculated and the field is assigned this value. (Any remaining WHEN parms are not evaluated.)

If none of the WHEN expressions are "true", the field is assigned the value of the ELSE ASSIGN expression, if any. Or, if ELSE RETAIN was specified the compute field will retain the value it had for the previous input record. If none of the WHEN expressions are "true", and no ELSE ASSIGN/RETAIN parm is present, the field will be set to a default value. The default value depends on the type of field being defined, as shown in the following table:

DEFAULT VALUE ASSIGNED TO COMPUTE FIELDS

FIELD TYPE	DEFAULT VALUE
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

In general, the **data type** of the COMPUTE field will be the data type of the first operand found in the first (or only) computational expression.

There is one exception to this rule and it involves time fields. A computational expression for a time value may contain a mixture of time and numeric operands. A COMPUTE field will be considered a time field if any of the computational expressions use a time operand, regardless of the data type of the first operand in the expression. This allows you to begin time–type computational expressions with a numeric operand.

A description of the **size** of character compute fields and the **number of decimal digits** in numeric and time compute fields appears under the "nnn" parm (page 511).

fieldname[(parms)]

Specifies the name of the field being created, and optionally specifies certain attributes for it. The fieldname must not have been previously used (in either a FIELD statement for the same file, or in a previous COMPUTE statement). You may name the new field anything you like, within the rules governing field names given on page 446.

No parms are required with the fieldname. If desired, specify one or more parms by placing them in parentheses immediately after the fieldname. (Do *not* leave a space between the field name and the open parenthesis). Separate the parms with a comma and/or blanks.

```
Example: COMPUTE: SEMI-ANNUAL-SALES  =  SALES-QTR1 + SALES-QTR2
```

The above example creates a new field named SEMI–ANNUAL–SALES. It will be a numeric field, since the first operand in the computational expression (SALES–QTR1) is a numeric field.

computational–expression

Used in the simple form of the COMPUTE statement. Specifies how to compute the value to assign to the field. The syntax for computational expressions is shown on page 472.

Example: (See the examples beginning on page 513.)

ACCUM/NOACCUM

This parm is valid only for numeric and time fields. It specifies whether the field should be accumulated or not when it appears as a column in a report or PC file. Fields that are accumulated will appear in the totals line, as well as in any other statistics lines that have been requested (such as the average line, the maximum line, etc.) Fields that are not accumulated will not appear in the totals and statistics lines.

By default, Spectrum Writer accumulates all **numeric fields** listed in the COLUMNS statement, with one exception. Numeric fields that are displayed using a PICTURE which contains special characters are not accumulated. (Special characters include such things as parentheses, imbedded dashes, asterisks, etc.) By default, numeric fields displayed with such a PICTURE are not accumulated and therefore do not appear in the total line and other statistical lines.

By default, **time fields** are *not* accumulated. Specify ACCUM if you want to see totals for a time field. This might be the case for time fields that contain *durations*, as opposed to times of day.

Any ACCUM or NOACCUM parm specified here can be overridden directly in the COLUMNS statement. The use of the ACCUM and NOACCUM parms is discussed on page 148.

```
Example: COMPUTE: AVERAGE-SALES(NOACCUM) = YEARLY-SALES / 4
```

The above example specifies that the AVERAGE–ANNUAL–SALES field will not be accumulated when it appears as a column in a report. Therefore, it will not receive Grand Totals or totals at control breaks.

```
Example: COMPUTE: DURATION(ACCUM) = END-TIME - START-TIME
```

The above example specifies that the DURATION field should be accumulated. Therefore, a total value for it will appear in the total lines at control breaks and in the Grand Total line.

ASSIGN(computational–expression)

Specifies how to compute a value which might be assigned to the compute field. If more than one ASSIGN expressions are used in the COMPUTE statement, they must all compute a result of the *same data type*. The syntax for computational expressions is shown on page 472.

Note: No space is allowed between the word ASSIGN and the parenthesis that follows it.

Example: See the examples beginning on page 513.

display–format

Specifies the default format to be used when displaying this field in a report. A complete list of display formats is found in Appendix B, "Display Formats" (page 617).

The display–format specified in the COMPUTE statement tells Spectrum Writer the *default* format to use when displaying the field anywhere in a report — in the titles, the main report lines, the break headings and footings, etc. The display format specified here can later be overridden by specifying a display format parm directly in the COLUMNS statement, the TITLE statement, etc.

If this parm is not specified, Spectrum Writer uses a default display format when printing the field in a report. Default display formats are shown in the table on page 618.

Note: Specifying a PC file option (LOTUS, for example) causes any display format specified in the COMPUTE statement to be overridden (with a display format appropriate for the desired PC program).

```
Example: COMPUTE: AVERAGE-SALES(PIC'$$$,$$9') = YEARLY-SALES / 4
```

The above example specifies that the AVERAGE–SALES field should be displayed using the PICTURE "$$$,$$9" when it is printed in a report. This picture uses a floating dollar sign, and does not display any decimal digits.

DIVTOTS

This parm is valid only for certain types of numeric computations. It specifies how the "total" value for this field should be computed at control breaks and at the Grand Totals line. By default, a field's total is merely the sum of all the individual values for the field. For percentages and ratios, such a total is often meaningless. Instead, what is desired is that the percentage or ratio be computed for the entire control group (or for the entire report at the Grand Total). The DIVTOTS ("divide totals") parm tells Spectrum Writer to compute the field's total by performing just such a control-group-wide division. The DIVTOTS parm is discussed in more detail in "Computing True Percentages and Ratios at Control Breaks" (page 202).

DIVTOTS may only be specified for COMPUTE statements that meet all of the following requirements:

- At its highest level, the expression must consist of a single division operation. The numerator and/or denominator themselves, however, can be expressions within parentheses.
- Neither the numerator nor the denominator may be literal values. Each must be either a field or an expression.
- Only simple COMPUTE statements may use the DIVTOTS parm. It is not allowed in conditional COMPUTE statements. (Conditional COMPUTE statements are those that use the WHEN and ASSIGN parms to assign different values to a field.) However, either or both of the numerator and the denominator can be COMPUTE fields that may have been computed with conditional COMPUTE statements.

Example: See the DIVTOTS example on page 515.

ELSE

Indicates the action to take if *none* of the preceding WHEN parms are "true." When the ELSE parm is followed by an ASSIGN parm, the value from that ASSIGN parm is assigned to the compute field. When the ELSE parm is followed by a RETAIN parm, the value of the compute field is not changed— it retains whatever value it had for the previous input file record. If present, the ELSE parm and its associated ASSIGN/RETAIN parm must be the last items in the COMPUTE statement.

Example: See the examples beginning on page 513.

'heading1|heading2...'

Specifies the column heading lines to use for this field when it appears as a column in a report or PC file. Enclose the column heading in either apostrophes or quotation marks. If you need to use that same character (an apostrophe or quotation mark) within the text, use two of those characters for each character desired.

Use a vertical bar (|) to separate the column heading text into separate lines. It is not necessary to add your own "padding" spaces in order to make the column heading texts stack neatly in your report. Spectrum Writer automatically centers each part of the column heading for you.

Note: You may use the HDGSEP parm of the OPTIONS statement to select a character other than the vertical bar (|) to use as the separator character for column heading texts.

If no column headings are specified, Spectrum Writer will use the field name itself as the column heading. The name will be broken apart at each dash or underscore, with each part of the name going onto a separate heading line.

Note: Any column headings specified here can be overridden by using a column heading parm directly in the COLUMNS statement.

See page 130 for more information on column headings.

```
Example: COMPUTE: AVERAGE-SALES('AVERAGE|ANNUAL|SALES') = YEARLY-SALES / 4
```

The above example specifies that "AVERAGE|ANNUAL|SALES" should be used as the column heading when this field appears in a report. The vertical bars specify that each word will go on a separate column heading line.

nnn

This parm is valid only for character, numeric and time compute fields. For **character fields**, this numeric parm specifies the **size** of the character field being created. If this parm is omitted, the default size of the field will be the sum of the size of all operands in the computational expression. If there are more than one computational expressions in the statement, the size of the *largest* possible result is used. If an explicit size parm is specified and it is not the same as this default size, the computed result will either be truncated or right–padded with blanks to create a field of the desired size. The maximum size of a character field is 32K.

```
Example: COMPUTE: SHORT-NAME(5) = LAST-NAME
```

The above example creates a character field that is only 5 bytes long. The SHORT-NAME field will contain the first five bytes of the LAST–NAME field. If the "5" parm had been omitted, the SHORT–NAME field would have been the same size as the only operand in the expression— the LAST–NAME field.

For **numeric and time fields**, this numeric parm specifies how many **decimal digits** should be retained at each stage of the computation. The final result, as well as each intermediate result obtained during the computation, will be rounded to this precision. If this parm is omitted, Spectrum Writer chooses a default number of decimal places to maintain, based on the precision of the operands involved and on the operations performed.

If you choose to override the default and specify a number of decimal digits here, be sure to specify ***at least* 2 decimal digits** more than you actually plan to display in the report (to allow for intermediate rounding). This is especially important if your computational expression involves division.

Here is an illustration of why this is important. Suppose we compute a "percent tax" field by dividing TAX into AMOUNT and then multiplying by 100. Even assuming we only want to show one decimal digit for this field in our report, you should still specify at least 3 decimal digits in your COMPUTE statement:

```
COMPUTE: TAX-PERCENT(3) = TAX / AMOUNT * 100
COLUMNS: TAX-PERCENT(PIC'ZZ9.9%')
```

Here's why. The TAX field in the first SALES-FILE record contains 6.09: the AMOUNT field contains 101.38. For that record, the first division in the COMPUTE statement (6.09 / 101.38) gives a rounded intermediate result of 0.060. That is then multiplied by 100, giving a final rounded result of 6.000. (This result will then be displayed with just one decimal digit — as 6.0% — in the report.) If we had specified only one decimal digit in the COMPUTE statement, the first division would have been rounded to 0.1 (that is, 0.060 rounded down to just one decimal digit.) Multiplying by 100 would then have resulted in a final result of 10.0, — an incorrect result caused by the loss of precision in the earlier operation.

> **Note:** The maximum number of digits (including decimal digits) allowed for numeric fields is 31.

Example:
```
COMPUTE: PERCENT-CHANGE(4) = (NEW - OLD) / OLD * 100
COLUMNS: PERCENT-CHANGE(P'ZZ9.9')
```

The above example specifies that 4 decimal digits should be maintained while computing the value of PERCENT–CHANGE. In the COLUMNS statement, however, we specified that only 1 decimal digit should actually be displayed for that field.

Example:
```
COMPUTE: DURATION(1) = END-TIME - START-TIME
```

The above example specifies that the DURATION field should contain 1 decimal digit. If the "1" parm had not been specified, the result would have had the same number of decimal digits as the operands.

Also see the examples beginning on page 513.

RETAIN

When used, this keyword must be the last item in the COMPUTE statement and must be preceded by the keyword ELSE. It specifies that if none of the WHEN parm conditional

expressions are true, the COMPUTE field should retain its previous value (rather than be assigned a default value). For a speed–up tip relating to the RETAIN parm, see page 656.

WHEN(conditional–expression)

Specifies a conditional expression to be evaluated before assigning a value to the field being created. The WHEN parms are evaluated in the order in which they appear in the COMPUTE statement. Evaluation of WHEN parms stops as soon as the first WHEN parm is found whose conditional expression is "true". The value specified in the following ASSIGN parm is assigned to the computed field.

The syntax for conditional expressions is shown on page 459.

Note: No space is allowed between the word WHEN and the parenthesis that follows it.

Note: If a field containing invalid data is encountered while evaluating the conditional expression in a WHEN parm, the entire WHEN expression will be considered *false*. The associated ASSIGN parm will not be used.

Example: See the examples beginning on page 513.

EXAMPLES

See page 474 for additional examples of COMPUTE statements.

Case 1. Creating a numeric field with a simple COMPUTE statement.

```
COMPUTE:  BONUS = TOTAL-SALES * .08
```

The above example creates a new field named BONUS. Its value will be computed by multiplying the TOTAL–SALES field by .08.

Case 2. Creating a numeric field, based on conditions.

```
COMPUTE: BONUS(2) = WHEN(HIRE-DATE < 1/1/1980)  ASSIGN(TOTAL-SALES * .08)
                    WHEN(HIRE-DATE < 1/1/1985)  ASSIGN(TOTAL-SALES * .07)
                    ELSE                        ASSIGN(TOTAL-SALES * .05)
```

The above example creates a new field named BONUS, which will have two decimal digits. The value assigned to this new field depends on conditions involving the HIRE-DATE field. When the hire date is before January 1, 1980, the bonus is computed as 8 percent of the total sales (TOTAL–SALES * .08). If the hire date is not before January 1, 1980, but *is* before January 1, 1985, then the bonus is computed based on 7 percent. Otherwise (if neither of the two preceding conditions is true) the bonus is computed using 5 percent of total sales.

Case 3. Creating a character field, based on conditions.

```
COMPUTE: STATE-NAME = WHEN(STATE = 'CA')  ASSIGN('CALIFORNIA')
                      WHEN(STATE = 'TX')  ASSIGN('TEXAS')
                      WHEN(STATE = 'NY')  ASSIGN('NEW YORK')
                      ELSE                ASSIGN('?????')
```

The above example creates a new field called STATE–NAME. It will be a 10-byte character field, since "CALIFORNIA" is the largest possible value that may be assigned to it. The value assigned to the STATE–NAME field depends on conditions involving the value of the STATE field. If the STATE field contains some value other than those listed in the three WHEN parms, the STATE–NAME field will be assigned a value of 5 question marks. Use this technique to perform "table lookups." (See page 655 for a speed-up tip involving table lookups.)

Case 4. Creating a date field, based on conditions.

```
COMPUTE: START-DATE = WHEN(HIRE-DATE > 1/1/1990)  ASSIGN(HIRE-DATE)
                      ELSE                        ASSIGN(1/1/1990)
```

The above example creates a new date field called START–DATE. Its value will either be the value of the HIRE–DATE field (if the hire date is after January 1, 1990), or else it will be the literal date 1/1/1990.

Case 5. Creating a bit field, based on conditions.

```
COMPUTE: VIP = WHEN(TOTAL-SALES > 50000 OR HIRE-DATE < 1/1/1985) ASSIGN(#ON)
               ELSE                                              ASSIGN(#OFF)
```

The above example creates a new bit field named VIP. The value of the field will be ON if the TOTAL–SALES field is greater than 50000, *or* if the HIRE–DATE field is earlier than 1/1/1985. Otherwise, the VIP field will be OFF. (The ELSE ASSIGN pair in this example are not actually necessary, since OFF is the default value assigned to bit fields when none of the WHEN parms is true.)

Case 6. Creating a character field using a hexadecimal literal.

```
COMPUTE: MASTER-FILE-KEY = X'FF' + EMPL-NUM + X'0000'
```

The above example creates a new character field named MASTER–FILE–KEY. It will be a 6 byte field, consisting of 1 byte of "high–values" (X'FF'), followed by the contents of the 3–byte character field EMPL–NUM, followed by 2 bytes of "low values" (hex zeros).

Case 7. Creating a field using a built–in function.

```
COMPUTE: BIGGEST-QTR = #MAX(SALES-QTR1, SALES-QTR2, SALES-QTR3, SALES-QTR4)
```

The above example creates a new numeric field named BIGGEST–QTR. Its value will be the greater of the four quarterly sales values. A complete list of built–in functions that can be used in the COMPUTE statement is found in Appendix D, "Built-In Functions" (page 628).

Case 8. Creating a field based on the contents of a bit field.

```
COMPUTE: BONUS = WHEN(FULL-TIME)  ASSIGN(TOTAL-SALES * .10)
                 ELSE             ASSIGN(TOTAL-SALES * .07)
```

The above example creates a new numeric field named BONUS. Its value will depend on the contents of the FULL–TIME bit field. When the FULL–TIME bit is "on," the bonus is computed as 10 percent of TOTAL–SALES. Otherwise (when the bit field is "off") the bonus is computed as 7 percent of TOTAL–SALES.

Case 9. Using the DIVTOTS ("divide totals") parm with a percentage computation.

```
COMPUTE: PERCENT-TAX(DIVTOTS) = TAX / AMOUNT
```

The above example computes the PERCENT–TAX field by dividing the TAX field by the AMOUNT field. If the DIVTOTS parm had not been specified, the sum of all of the PERCENT–TAX fields would have printed in the total lines. The DIVTOTS parm tells Spectrum Writer to use the result of a group-wide division as the total value instead of such a sum. At control breaks and at Grand Totals time, Spectrum Writer will now divide the total value of TAX by the total value of AMOUNT. This group–wide division will then be used instead of the normal total for the PERCENT–TAX field.

COPY Statement

PURPOSE

Specifies that control statements stored in a dataset should be processed at this point. This is useful for groups of control statements that are used in many different jobs.

Also use this statement to copy Cobol or Assembler record layouts from their respective libraries.

You are allowed to have additional COPY statements imbedded among the statements that are being copied. This nesting of COPY statements is allowed to any level.

FEATURES

Use the COPY statement to:

- specify **where** the control statements to be copied are located
- specify whether or not to **list** the copied control statements in the control listing

LEARNING MORE

The complete syntax of the COPY statement is shown on the following pages. In addition, the following parts of the manual relate to the COPY statement:

- the Spectrum Writer Copy Library is discussed beginning on page 360
- the OS/390 JCL aspects of the copy library are discussed beginning on page 423
- the VSE JCL aspects of the copy library are discussed beginning on page 437
- the use of the COPY statement in conjunction with Cobol and Assembler record layouts is discussed on page 382

SYNTAX

COPY STATEMENT SYNTAX (OS/390)

```
COPY:     copyname/DDNAME(ddname)
       [  LIST(YES/NO)          ]
       [  NOTALIAS              ]
       [  PDSDDN(ddname)        ]
```

COPY STATEMENT SYNTAX (VSE)

```
COPY:     copyname
       [  LIST(YES/NO)          ]
       [  NOTALIAS              ]
       [  SUBLIB('libr.sublib') ]
```

Standard Spelling	Alternate Spellings
DDNAME	DDN
NO	N
YES	Y

Either a copyname parm or a DDNAME parm is required. All other parms are optional. If present, the copyname parm must be the first parm in the COPY statement. Other parms may appear in any order.

copyname (under OS/390)

Identifies a member to be copied from a PDS. If present, copyname must be the first parm in the COPY statement. The copyname parm can be either:

- a PDS member name. Example: COPY: SALES
- a Spectrum Writer alias (under certain circumstances). Example: COPY: SALES–FILE

A **member name** is a 1– to 8–byte alphanumeric name that begins with a non–numeric character. The special characters #, $, and @ are also allowed anywhere in member names.

Normally, the member will be copied from your Spectrum Writer Copy Library — the PDS pointed to by the **SWCOPY DD** in your JCL (see page 423). You can, however, copy from a different PDS by adding a PDSDDN parm to the COPY statement

> **Note:** If a COPY statement occurs within a COBOL record layout, the member will be copied from the PDS named in the COBLIB DD (not the SWCOPY DD). Similarly, if the COPY statement occurs within an Assembly language record layout, the member will be copied from the PDS named in the ASMLIB DD.

Instead of a member name, you can also use an **alias name** (*if* you are copying from the standard SWCOPY PDS)**.** Alias names can be up to 70 characters long and must conform to the Spectrum Writer naming conventions for file names (page 446). Aliases for library

members are assigned in a special member named SWALIAS in the standard Spectrum Writer Copy Library. The use of aliases is discussed in the section beginning on page 367.

> **Note:** Aliases may *not* be used when the PDSDDN parm is used, or when the COPY statement occurs within a Cobol or Assembler record layout.

```
Example: COPY:  SALES
```

The above example copies the member named SALES from the PDS pointed to by the SWCOPY DD.

```
Example: COPY:  SALES-FILE
```

The above example specifies that the member associated with the alias "SALES-FILE" should be copied. That name must be defined as an alias in the SWALIAS member of the copy library (SWCOPY DD). As shown in Appendix F, "Files Used in Examples" (page 648), "SALES–FILE" is an alias for the member named "SALES". Therefore, the above statement would also copy the control statements from the copy library member named SALES.

copyname (under VSE)

Identifies a member to be copied from a Librarian sublibrary. The copyname parm is required and must be the first parm in the COPY statement. The copyname parm can be any of the following:

- a member name. Example: COPY: SALES
- a member name and a member type, separated with a dot.
  Example: COPY: SALES.SW
- an alias name (under certain circumstances). Example: COPY: SALES–FILE

A **member name** is a 1– to 8–byte alphanumeric name that begins with a non–numeric character. The special characters #, $, and @ are also allowed anywhere in member names.

Normally, the member will be copied from your Spectrum Writer Copy Library — the sublibrary named in an OPTIONS statement SUBLIB parm (see page 437). You can, however, copy from a different sublibrary by adding a SUBLIB parm to the COPY statement.

> **Note:** You can also specify special sublibraries to be used for COPY statements that occur within Cobol or Assembler record layouts. Use the COBLIB or ASMLIB options (in an OPTIONS statement), respectively.

You may also append a **member type** after the member name, separated by a dot. (For example: SALES.SW). If no member type is specified in this way, the member type used for the copy will be:

- "C", if the COPY statement is embedded within a Cobol record layout, or
- "A", if the COPY statement is embedded within an Assembler record layout, or
- the member type specified in an OPTIONS statement MEMTYPE parm, if any, or
- "SPECTWTR" otherwise

Instead of a member name, you can also use an **alias name** (*if* you are copying from the standard copy library specified in an OPTIONS statement SUBLIB parm)**.** Alias names can be up to 70 characters long and must conform to the Spectrum Writer naming conventions for

file names (page 446). Aliases for library members are assigned in a special member named SWALIAS.SPECTWTR in the sublibrary named in the SUBLIB parm of an OPTIONS statement. The use of aliases is discussed in the section beginning on page 367.

Note: Aliases may *not* be used when the SUBLIB parm (of the COPY statement) is used, or when the COPY statement occurs within a Cobol or Assembler record layout.

```
Example: OPTIONS: SUBLIB('SPECTRUM.PROD')
         COPY:    SALES
```

The above example copies the member named SALES.SPECTWTR from the SPECTRUM.PROD sublibrary.

```
Example: OPTIONS: SUBLIB('SPECTRUM.PROD')
         COPY:    SALES-FILE
```

The above example specifies that the member associated with the alias "SALES-FILE" should be copied. That name must be defined as an alias in the SWALIAS.SPECTWTR member of the copy library (SPECTRUM.PROD). As shown in Appendix F, "Files Used in Examples" (page 648), "SALES–FILE" is an alias for the member named "SALES". Therefore, the above statement would also copy the control statements from the copy library member named SALES.SPECTWTR

DDNAME(ddname)

OS/390 only. Specifies the DD name of a sequential input file that is to be copied. This feature is useful when the control statements that you want to copy are not in a PDS. This parm and the copyname parm are mutually exclusive.

One use of the DDNAME parm is to copy datasets that are stored in proprietary libraries that Spectrum Writer does not access directly (such as PANVALET or LIBRARIAN). Add a job step ahead of Spectrum Writer to copy the desired proprietary data to a temporary sequential dataset. Then have Spectrum Writer copy that sequential dataset by using the DDNAME parm.

```
Example: COPY:  DDNAME(TEMPDD)
```

The above example specifies that the control statements to be copied are located in a sequential dataset identified by the TEMPDD DD in the JCL.

LIST(YES/NO)

The LIST parm specifies whether the copied control statements should be listed in the control listing. If the LIST parm is not specified, the default is not to list the copied statements.

Note: If an error is detected in any of the copied control statements, that statement *will* be listed, along with the error message, regardless of the value of this parm.

```
Example: COPY:  SALES  LIST(YES)
```

The above example specifies that the control statements copied from the SALES member should be listed in the control listing.

NOTALIAS

Specifies that the copyname parm is not an alias. When this parm is present, no alias checking is performed and the copyname must be the name of the member to be copied.

Use this parm if the name of the member you want to copy also happens to be the alias name of a different member.

```
Example: COPY:  SALES  NOTALIAS
```

The above example specifies that the control statements should be copied from the member named SALES. This will be done even if SALES has been defined as an alias for some other member.

PDSDDN(ddname)

OS/390 only. The PDSDDN parm specifies the DDNAME of a DD statement in the JCL that points to the PDS containing the member to be copied. This parm is valid only in conjunction with the copyname parm. When PDSDDN is used, the copyname must specify a member name rather than an alias. (No alias checking is performed on the copyname.)

```
Example: COPY:  SALES  PDSDDN(MYLIB)
```

The above example specifies that the control statements to be copied are in the PDS identified by the MYLIB DD in the JCL. The member copied is named SALES.

SUBLIB('lib.sublib')

VSE only. The SUBLIB parm specifies the name of the Librarian sublibrary containing the member to be copied. When SUBLIB is used, the copyname must specify a member name (and optionally a member type) rather than an alias. (No alias checking is performed on the copyname.)

Note: Ensure that your JCL contains any DLBL and EXTENT statements needed to define the sublibrary named in this parm.

```
Example: COPY:  SALES.TEST  SUBLIB('TEST.MYLIB')
```

The above example specifies that the control statements to be copied are in the sublibrary named TEST.MYLIB. The member copied is named SALES.TEST.

FIELD Statement

PURPOSE

Defines one field from an input file to Spectrum Writer. This statement provides certain essential information about a field, such as where it is located in a record, how long it is, etc. Before a field can be referred to in any other control statement, it must first be defined using the FIELD statement.

You can also use the FIELD statement to specify various display options to be used when the field appears in a report or PC file. These options include: the columns headings to use; the display format to use; whether to include the field in Grand Totals, etc.

You may have as many FIELD statements as you like. These statements are normally kept in the Spectrum Writer Copy Library (see page 420).

FEATURES

Use the FIELD statement to:

- define **where a field** is located within a record
- define the **type of data** contained within the field
- define the default **column headings** to be used whenever the field is printed in a report or PC file
- define the default **display format** to be used whenever the field is printed in a report
- define whether or not a numeric field should be **accumulated**, and therefore appear in total lines (and other statistical lines)
- define the texts that should be used in a report to indicate whether a **bit field** is ON or OFF
- define how to use a **data exit** to obtain a field's value

LEARNING MORE

The complete syntax of the FIELD statement is shown on the following pages. In addition, the following parts of the manual relate to the FIELD statement:

- how to write FIELD statements is discussed beginning on page 328.

SYNTAX

FIELD STATEMENT SYNTAX

```
FIELD:   fieldname
      [  ACCUM/NOACCUM                                ]
      [  BIT(n)                                       ]
      [  COLUMN(nnnnn/expr/*)/DISP(nnnnn/expr/*)      ]
      [  DECIMALS(nn/0)                               ]
      [  DXPARM('text')                               ]
      [  DXPROG('program')                            ]
      [  DXRETDEC(nn)                                 ]
      [  DXRETLEN(nnnnn)                              ]
      [  FILE(filename/*)                             ]
      [  FORMAT(display-format)                       ]
      [  HEADING('heading1|heading2|heading3...')     ]
      [  LENGTH(nnnnn)                                ]
      [  OFFSET(numeric-expression)                   ]
      [  OFFTEXT('text')                              ]
      [  ONTEXT('text')                               ]
      [  TYPE(datatype/CHAR)                          ]
```

Standard Spelling	Alternate Spellings
ACCUM	ACC
COLUMN	COL
DECIMALS	DECIMAL, DEC
DESCRIPTION	DESC
DISP	DISPLACEMENT
DXRETLEN	DXRETLGTH
FIELD	FLD
FORMAT	FMT
HEADING	HEADINGS, HEAD
LENGTH	LGTH, LEN
NOACCUM	NOACC
OFFTEXT	OFF
ONTEXT	ON
TYPE	TYP

The fieldname is *required* in a FIELD statement, and must be the *first* item after the statement prefix. After that, one or more other parms will be required, depending on the type of field being defined. Those parms may appear in any order.

fieldname

Specifies the name of the field being defined. All other control statements will use this name when referring to this field. You may assign any name you like, within the rules governing field names given on page 446.

```
Example: FIELD:  LAST-NAME  COLUMN(4)  LENGTH(15)
```

The above example defines a field named LAST–NAME.

ACCUM/NOACCUM

This parm is valid only for numeric and time fields. Specifies whether a field should be accumulated or not when it appears as a column in a report. Fields that are accumulated will appear in the totals line, as well as in any other statistics lines that have been requested (such as the average line, the maximum line, etc.) Fields that are not accumulated will not appear in the totals and statistics lines.

By default, Spectrum Writer accumulates all **numeric fields** listed in the COLUMNS statement, with one exception. Numeric fields that are displayed using a PICTURE which contains special characters are not accumulated. (Special characters include such things as parentheses, imbedded dashes, asterisks, etc.) By default, numeric fields displayed with such a PICTURE are not accumulated and therefore do not appear in the total line and other statistical lines.

By default, **time fields** are *not* accumulated. Specify ACCUM if you want to see totals for a time field. This might be desired for time fields that contain *durations*, as opposed to times of day.

Any ACCUM or NOACCUM parm specified here can be overridden directly in the COLUMNS statement.

The use of the ACCUM and NOACCUM parms is discussed beginning on page 148.

```
Example: FIELD: DEPT-NUM COLUMN(37) LENGTH(1) TYPE(NUM) NOACCUM
```

The above example defines a numeric field called DEPT–NUM. When this field appears as a column in a report, it will not be accumulated. Therefore, the column will not appear in the Grand Totals line or in control break totals.

```
Example: FIELD: TIME-ON-PHONE COLUMN(73) LENGTH(4) TYPE(SECS) DECIMALS(1) ACCUM
```

The above example defines a time field called TIME–ON–PHONE. Since this time field represents a *length* of time (as opposed to a time of day), it is appropriate to total this field. The ACCUM parm tells Spectrum Writer to accumulate this field by default. Therefore, it will be included in total and statistical lines.

BIT(n)

This parm is valid only for bit type fields. Identifies the specific bit (within a byte) that is being defined. The bits are numbered from 1 to 8, starting with the leftmost (high order) bit. If this parm is present, you do not need to specify the TYPE parm. TYPE(BIT) will be assumed. The use of this parm is discussed beginning on page 347.

```
Example: FIELD: PART-TIME  COLUMN(42)  BIT(2)
```

The above example defines a bit field named PART–TIME. The bit is the second bit from the left (the X'40' bit) in the 42nd byte of the record. Notice that the TYPE and LENGTH parms are not needed when defining a bit type field.

Also be aware that the current location counter is *not* incremented after a bit field is defined. That means that a COLUMN or DISP parm will required on the next FIELD statement (unless it happens to be for another bit field within the same byte).

COLUMN(nnnnn/expr/*)/
DISP(nnnnn/expr/*)

Specifies where the field begins within the record. If you use the COLUMN parm, the bytes in the record are numbered beginning with 1. If you use the DISP parm, the bytes in the record are numbered beginning with 0. For example, both of the following statements define the LAST–NAME field as beginning in the 4th byte of the record:

```
FIELD:  LAST-NAME  COLUMN(4)  LENGTH(15)
FIELD:  LAST-NAME  DISP(3)    LENGTH(15)
```

When reading variable–length records, Spectrum Writer ignores the 4-byte "record descriptor word" (RDW) at the beginning of each record. Thus, column 1 always refers to the first byte of actual user data in a record. It does *not* refer to the first byte of the RDW, if present. Use the KEEPRDW option (in the FILE, INPUT or OPTIONS statement) if you do want to define fields within the RDW.

Instead of using actual numbers within these parms, you may use an expression. (When using expressions, it makes no difference whether you use the COLUMN or the DISP parm.) An **expression** consists of another field name or an asterisk, optionally followed by a plus or minus sign and a number:

```
COLUMN( fieldname/*   [ +/-  nnnnn ]   )
DISP(   fieldname/*   [ +/-  nnnnn ]   )
```

If a field name is used, that field's *starting* byte in the record is used as the base of the expression. If an asterisk is used, the "default location" in the record is used as the base of the expression. (The default location is defined as the first byte *after* the previously defined field.) Following the base, the expression can optionally contain a number to add to, or subtract from,— that base. The result is then used as the field's starting position in the record.

```
Example: FIELD:  HIRE-DATE  COLUMN(LAST-NAME + 30)
```

The above example specifies that the HIRE–DATE field begins 30 bytes after the beginning of the LAST–NAME field. If the LAST–NAME field began in column 4 (displacement 3), then the HIRE–DATE field will begin in column 34 (displacement 33). Here is another example:

```
Example: FIELD:  HIRE-DATE  COLUMN(34)     TYPE(YYMMDD)
         FIELD:  HIRE-DD    COLUMN(* - 2)  LENGTH(2)
```

The first statement above defines HIRE–DATE as a 6 byte date field in the format YYMMDD. The second statement defines a field which *redefines* the last two bytes of the previous field. The second field starts two bytes *before* the current position in the record. This field, named HIRE–DD, is just a two byte character field which contains the DD portion of the HIRE–DATE field.

> **Note:** Blanks are *required* around any minus sign used in these parms (to avoid ambiguity with dashes used within fieldnames). Blanks are optional around the plus sign.

If neither COLUMN nor DISP is specified for a field, the default is to use the "default" location in the record. In other words, the default is to assume that COLUMN(*) — or DISP(*) — was specified.

The use of the COLUMN and DISP parms is discussed beginning on page 350.

DECIMALS(nn/0)

This parm is valid only for numeric and time fields. Specifies how many decimal digits are contained within the data in the record. If this parm is omitted, the data is assumed to contain zero decimal digits.

```
Example: FIELD: SALARY COLUMN(42) LENGTH(4) TYPE(PACKED) DECIMALS(2)
```

The above example defines a numeric field named SALARY. There are two decimal digits in this field's data. Thus, if the value in a record is X'0123456C', the SALARY value would be 1234.56.

```
Example: FIELD: TIME-ON-PHONE COLUMN(69) LENGTH(4) TYPE(SECS) DECIMALS(1)
```

The above example defines a time field named TIME–ON–PHONE. It is a 4–byte field containing a time expressed as a number of seconds. The number of seconds includes one decimal digit. Thus, if the value in a record is C'0123', the TIME–ON–PHONE value would be 12.3 seconds (00:00:12.3).

DXPARM('text')

This parm is valid only for fields whose TYPE is a data exit (NUMEXIT, for example). Anytime the user data exit program is called by Spectrum Writer, the text specified in this parm will be passed to the exit program. Typically this text is used to tell the exit program what function to perform. The use of this parm is discussed in the section beginning on page 357.

Example: See the example below under the DXPROG parm.

DXPROG('program')

This parm is valid only for fields whose TYPE is a data exit (NUMEXIT, for example). Specifies the name of the load module (OS/390) or phase (VSE) that Spectrum Writer will call in order to obtain the field's value. The use of this parm is discussed in the section beginning on page 357.

```
Example: FIELD:  DECRYPTED-NAME  TYPE(CHAREXIT)  COLUMN(29)  LENGTH(15)
                 DXPROG('DECRYPGM')
                 DXPARM('DECRYPT NAME')
                 DXRETLEN(20)
```

The above example defines a character field named DECRYPTED–NAME. The contents of this field *do not exist* within the record itself, but can be created by an exit program named DECRYPGM. (This imaginary program takes a 15 byte encrypted value and decrypts it into a readable 20-byte name). The DECRYPGM program will be passed the 15 bytes of data beginning at column 29 in the record. It will also be given the contents of the parm ("DECRYPT NAME") to tell it what function it should perform. The exit program will then perform its decryption logic and return a 20-byte value to be used by Spectrum Writer as the value for the DECRYPTED–NAME field.

DXRETDEC(nn)

This parm is required for all fields whose TYPE is NUMEXIT or TIMEEXIT, and is not allowed for any other type of field. This parm tells Spectrum Writer how many decimal digits there will be in the numeric or time value returned by the data exit for this field. For any kind of data exit field, the FIELD statement's DECIMALS parm value (if any) is simply *passed to* the data exit (which may or may not choose to make any use it). The DXRETDEC parm tells how many decimal digits to expect in the value *passed back* from the exit. Spectrum Writer needs to know how many decimal digits have been returned so that it can correctly format

the value (including the decimal point) when printing this field in a report. The use of this parm is discussed in the section beginning on page 357.

```
Example: FIELD: LAST-YEAR-SALES  TYPE(NUMEXIT)
                COLUMN(EMPL-NUM) LENGTH(3)
                DXPROG('SALELKUP')
                DXPARM('LAST YEAR')
                DXRETDEC(2)
```

The above example defines a numeric field named LAST–YEAR–SALES. The contents of this field *do not exist* within the record itself, but can be looked up in a special table by an exit program named SALELKUP. (This program takes a 3 byte employee number and looks up the sales figure for the year specified in the parm.) The SALELKUP program will be passed the 3 bytes of data beginning at the EMPL–NUM field in the record. It will also be given the contents of the parm ("LAST YEAR") to tell it what function it should perform. The exit program will then return the numeric value to be used for the LAST-YEAR-SALES field. That value will contain two decimal digits.

DXRETLEN(nnnnn)

This parm is required for all fields whose TYPE is CHAREXIT, and is not allowed for any other type of field. This parm tells Spectrum Writer the length of the character data that will be returned by the data exit program for this field. For a CHAREXIT field, the FIELD statement's LENGTH parm specifies how many bytes of raw data from the input record should be *passed to* the data exit. The DXRETLEN parm tells how many bytes will be *passed back* from the data exit. Spectrum Writer needs to know how much data will be passed back from the exit so that it can reserve the correct amount of space when printing this field in a report. The use of this parm is discussed in the section beginning on page 357.

Example: See the example above under the DXPROG parm.

FILE(filename/*)

Identifies the file in which the field is found. If this parm is omitted, it is assumed that the field being defined exists in the most recently defined file. (Files are defined using the FILE control statement.) This parm is useful for defining fields "out of order." This might occur if you used a COPY statement to read in the FILE and FIELD statements for several different files, and you want to go back and define additional fields for an earlier file.

```
Example: FIELD: WHOLE-NAME  COLUMN(4)  LENGTH(30)  FILE(EMPL-FILE)
```

The above example defines a field named WHOLE–NAME. This field is defined as a field in the EMPL–FILE file, even if other files have been defined more recently.

FORMAT(display–format)

Specifies the default format to be used when displaying the field in a report. This parm is used mainly for numeric, date and time fields. Appendix B, "Display Formats" (page 617) contains a complete list of the display formats available for each type of data.

If the FORMAT parm is omitted, a default display format will be used to format the field in a report. (The default display formats are listed on page 618.)

The FORMAT parm that you specify in the FIELD statement tells Spectrum Writer the *default* format to use when displaying the field anywhere in the report— in the titles, the main report lines, the break headings and footings, etc. Any display format specified here, however, can be overridden by putting a display format parm directly in the COLUMNS statement (or TITLE statement, etc.)

Note: The display–format parm is *not allowed* for bit fields. Bit fields are always displayed in a report with either the ONTEXT or OFFTEXT text.

Note: Specifying a PC file option (EXCEL, for example) causes any display format specified in the FIELD statement to be overridden (with a display format appropriate for the desired PC program).

Note: Fields containing invalid data are normally displayed using a special error indicator (for example, ****I****). This happens regardless of what display format may have been specified for the field.

```
Example: FIELD: SALARY     COLUMN(56) LENGTH(4) TYPE(PACK) FORMAT(PIC'$$,$$$,$$9.99')
         FIELD: HIRE-DATE   COLUMN(34)           TYPE(YYMMDD) FORMAT(LONG1)
         FIELD: STATUS-BYTE COLUMN(42) LENGTH(1)              FORMAT(HEX)
```

The first example above defines a numeric field named SALARY. When SALARY is displayed in a report, the PICTURE specified in the FORMAT parm will be used to format it. It will occupy 13 bytes, will include a floating dollar sign, and will show two decimal digits.

The second example defines a date field named HIRE–DATE. When this field is displayed in a report, the date will be formatted in the LONG1 format (with the month name spelled out completely).

The third example defines a one byte character field named STATUS–BYTE. When this field is displayed in a report, it will be shown in hexadecimal format (for example C1).

HEADING('heading1|heading2|heading3 ...')

Specifies the column heading line(s) to use for this field when it appears in a report or PC file. Enclose the column heading text in either apostrophes or quotation marks. If you need to use that same character (an apostrophe or quotation mark) within the text, use two of those characters for each character desired.

Use the vertical bar (|) to separate the column heading text into separate lines.

Note: You may use the HDGSEP parm of the OPTIONS statement to select a character other than the vertical bar (|) to use as the separator character for column heading texts.

If no HEADING parm is specified, Spectrum Writer will use the field name itself as the column heading. The name will be broken apart at each dash or underscore, with each part of the name going onto a separate heading line.

Any column headings specified here can be overridden by using an override column heading parm in the COLUMNS statement.

See page 130 for more information on column headings.

```
Example: FIELD: LAST-NAME  HEADING('NAME OF|EMPLOYEE')  COLUMN(4)  LENGTH(15)
```

The above example defines a field called LAST–NAME. When this field appears as a column in a report, its column heading will be "NAME OF" on the first line, and "EMPLOYEE" on the second line.

LENGTH(nnnnn)

Specifies how many bytes the field occupies in the record. Some data types imply a particular length (for example, FULLWORD and YYMMDD). For such data types, the LENGTH parm is not required. For data types that can be of various lengths, the LENGTH parm is required. The maximum length allowed varies according to the data type of the field being defined. The tables in Appendix A, "Data Types" (page 609) show the maximum length allowed for each data type. They also show which data types have a standard default length.

> **Note:** This parm tells how many *bytes* a field occupies in the input record. This is not necessarily equal to the number of *digits* that a numeric field contains. Refer to page 335 which discusses how to compute a numeric field's length based on how many digits it has.

```
Example: FIELD:  FIRST-NAME COLUMN(19)  LENGTH(15)
         FIELD:  SALARY     COLUMN(46)  LENGTH(4)  TYPE(PACKED)
```

The first example above defines a character field (FIRST–NAME) that occupies 15 bytes in the record. The second example defines a numeric field (SALARY) which occupies 4 bytes in the record. Since the field is defined as a PACKED type field, it will actually contain *7 digits*.

OFFSET(numeric–expression)

Used to define the beginning of a "variably located" record segment. (That is, a segment that begins at different locations in different input records.) The contents of the OFFSET parm tell Spectrum Writer the offset from the beginning of the record to the variably located segment. The OFFSET parm can contain any numeric expression. Spectrum Writer computes the value of the OFFSET parm anew for each input record (since the value can vary from record to record). It then *adds* this value to the location specified in the DISP or COLUMN parm (if there is one) to determine where the field is located within the input record. For more information on using the OFFSET parm, see "Variably Located Record Segments" on page 353.

> **Note:** The OFFSET value remains in effect for *all subsequent FIELD statements*, until another OFFSET parm is used to change or cancel it. Use OFFSET(0) in a FIELD statement if you later want to define fields without any OFFSET value.

> **Note:** The OFFSET parm also resets the default location pointer to "displacement zero." Thus, unless an explicit DISP (or COLUMN) parm is also specified, the field being defined will be at displacement zero within the variably located segment. If needed, you can use an explicit DISP (or COLUMN) parm to specify that the field begins at some other displacement within the segment.

Example

```
** FIELDS IN FIXED LOCATIONS
FLD: ID-OFFSET    DISP(32)  TYPE(FULLWORD)  /* DISP TO IDENTIFICATION SEGMENT */
FLD: ID-LEN                 TYPE(HALFWORD)
FLD: ID-NUM                 TYPE(HALFWORD)
FLD: IO-OFFSET              TYPE(FULLWORD)  /* DISP TO I/O ACTIVITY SEGMENT */
FLD: IO-LEN                 TYPE(HALFWORD)
FLD: IO-NUM                 TYPE(HALFWORD)
**
** SELECTED FIELDS FROM THE VARIABLY LOCATED IDENTIFICATION SEGMENT
FLD: JOBNAME                LEN(8)   OFFSET(ID-OFFSET)
FLD: PGMNAME                LEN(8)
FLD: STEPNAME               LEN(8)
**
```

```
** SELECTED FIELDS FROM THE VARIABLY LOCATED I/O ACTIVITY SEGMENT
FLD: NUM-CARDS              TYPE(FULLWORD)  OFFSET(IO-OFFSET)
FLD: NUM-TPUTS              TYPE(FULLWORD)  DISP(*+4)
FLD: NUM-TGETS              TYPE(FULLWORD)
**
FLD: FILLER OFFSET(0) LEN(1)     /* A "DUMMY" FIELD TO CANCEL THE OFFSET PARM */
```

The above statements use OFFSET parms to define two variably located segments within an SMF type 30 record. See "Variably Located Record Segments" (page 353) for a further explanation of this example.

OFFTEXT('text')

This parm is valid only for bit type fields. Specifies a text to print in reports for a bit field when its value is OFF. If omitted, the default is to print the word "NOT" followed by the field name itself (when its value is OFF). The use of this parm is discussed in the section beginning on page 347.

```
Example: FIELD:  PART-TIME  COLUMN(42)  BIT(2)
                 ONTEXT('PART TIME EMPL')  OFFTEXT('FULL TIME EMPL')
```

The above example defines a bit field named PART–TIME. When this field is printed in a report, the text "PART TIME EMPL" will print if the field's value is ON. The text "FULL TIME EMPL" will print if the field's value is OFF.

```
Example: FIELD:  DELETE-BIT  COLUMN(100)  BIT(8)  ONTEXT('1')  OFFTEXT('0')
```

The above example defines a bit field named DELETE–BIT. When this field is printed in a report, a "1" will print if the field's value is ON, and a "0" will print if the field's value is OFF.

ONTEXT('text')

This parm is valid only for bit type fields. Specifies a text to print in reports for a bit field when its value is ON. If omitted, the default is to print the field name itself in the report (when its value is ON). The use of this parm is discussed in the section beginning on page 347.

Example: See the example above under the OFFTEXT parm.

TYPE(datatype/CHAR)

Specifies what type of data the field contains. There are five general categories of data that Spectrum Writer recognizes: character, numeric, date, time, and bit. However, within each category there is more than one way that the data can actually be represented in a record. The TYPE parm specifies exactly how the data is stored in a record. Appendix A, "Data Types" (page 609) contains a complete list of the data types that Spectrum Writer supports.

If the TYPE parm is omitted, the default data type of CHAR (character) is assumed. However, there is one exception to this rule. If a BIT parm is present in the FIELD statement, then a data type of BIT is assumed.

```
Example: FIELD: SALARY    TYPE(COMP-3) COLUMN(46) LENGTH(4)
         FIELD: HIRE-DATE TYPE(YYMMDD) COLUMN(34)
         FIELD: PART-TIME TYPE(BIT)    COLUMN(42) BIT(2)
```

The first example above defines a numeric field (SALARY) which contains COMP-3 data. (COMP-3 data is called "packed" in Assembler and "fixed decimal" in PL/1.)

The second example defines a date field (HIRE–DATE) which contains a date in character YYMMDD format.

The third example defines a bit field (PART–TIME). The bit is the second bit from the left (the X'40' bit) in the 42nd byte of the record. In this example, it was not actually necessary to specify the TYPE parm, since the BIT parm implies a data type of BIT.

FILE Statement

PURPOSE

Defines one input file to Spectrum Writer. Before a file can be used as input for a report or PC file, it must first be defined using the FILE statement.

This statement by itself does *not* specify that a file should be used as input for a particular run. This statement simply defines a filename to Spectrum Writer so that subsequent control statements can refer to it. After a file has been defined using this statement, an INPUT or READ statement may be used to request that the file be used as input to a report or PC file.

You may have as many FILE statements as you like. These statements are normally kept together with FIELD statements in the Spectrum Writer Copy Library (see page 420).

FEATURES

Use the FILE statement to:

- define the **DDNAME** or **DLBL/TLBL** to use when reading a file
- define the **type of file** (for example, whether it's VSAM)
- define a file's **record length**

LEARNING MORE

The complete syntax of the FILE statement is shown on the following pages. In addition, the following parts of the manual relate to the FILE statement:

- how to write FILE statements is discussed beginning on page 328
- using a file that is accessed in a user I/O Exit is discussed in Appendix I, "I/O Exits" (page 673).

SYNTAX

FILE STATEMENT SYNTAX

```
FILE:     filename
       [  ATTR(type, 'dlbl/tlbl' [,SYSnnn] [,F/V] ,recsize
              [,blksize] [,STDLABEL/NOLABEL])          (VSE only)   ]
       [  DB2NAME('[qualifier.]name')                  (DB2 only)   ]
       [  DDNAME(ddname)                               (OS/390 only)]
       [  EXITPARM('text')                                          ]
       [  IOEXIT('program' [,'parm'] [,TRACE])                      ]
       [  KEEPRDW                                                   ]
       [  LRECL(nnnnn/1000)                            (OS/390 only)]
       [  NORMALIZE(fieldname, num-expr [, ...]) ...                ]
       [  NORMWHEN(conditional-expression)                          ]
       [  STOPWHEN(conditional-expression)                          ]
       [  TYPE(SEQ/VSAM/DB2/EXIT)                      (OS/390 only)]
```

Standard Spelling	Alternate Spellings
DDNAME	DDN
DESCRIPTION	DESC
EXITPARM	PARM
FILE	FIL
NORMALIZE	NORM
NORMWHEN	NORMALIZEWHEN

The filename is *required* in a FILE statement and must be the *first* item after the statement prefix. After that, one or more other parms may be required, depending on the type of file being defined. Those parms may appear in any order.

filename

This parm specifies the name of the file being defined. All other control statements will use this name when referring to this file. You may assign any name you like, within the rules governing file names given on page 446.

```
Example: FILE:  SALES-FILE  DDNAME(SALESDD)
```

The above example defines a file named SALES–FILE.

ATTR(type, 'dlbl/tlbl' [,SYSnnn] [,F/V] ,recsize [,blksize] [,STDLABEL/NOLABEL])

VSE only. This parm describes the attributes of a VSE file. This parm can also be specified in the INPUT and READ statements (and omitted from the FILE statement. For a file to be used as an input file in a report, the ATTR *must be specified* either in this statement, or in the INPUT or READ statement.

The following table describes the subparms. It is not necessary to use "place holding" commas for omitted subparms.

SUBPARMS ALLOWED IN THE ATTR PARM	
SUBPARM	**MEANING**
type	This parm is required. It tells Spectrum Writer what kind of file is being defined. It must be one of the following values: **DASD** a SAM file on a DASD device (disk). Use DASD (rather than VSAM) for VSAM–managed SAM files. **TAPE** a SAM file residing on a magnetic tape **VSAM** a VSAM file **EXIT** a file accessed via an I/O Exit program
'dlbl/tlbl'	This parm is required (except for exit files). It specifies the filename of a DLBL or TLBL statement present in the JCL. This DLBL/TLBL statement in the JCL will identify the actual data set to be read. Spectrum Writer uses the DLBL/TLBL to open an input file and read from it. This parm must be a 1– to 7–byte name within apostrophes (or quotation marks). For EXIT type files, the this parm is not required, but will be passed to the I/O Exit program if it is specified.
SYSnnn	This parm is required for TAPE files. It is treated as a comment for other file types. It identifies the logical unit on which the file will reside. The value specified here must also be "assigned" to a tape drive in your execution JCL.
F/V	This parm specifies whether your file contains fixed (F) or variable (V) length records. If omitted, fixed (F) is assumed.
recsize	This parm is required. It specifies the length of the records in your file. For variable length files, this parm specifies the length of the *largest* record that may be encountered in the file. Also, for variable length files this value *should include* the length of the 4–byte "record descriptor word" (RDW) which begins each variable–length record.
blksize	This parm is optional. It is treated as a comment for VSAM and EXIT files. For DASD and TAPE files, it specifies the length of each block in the file. For variable length files, this parm specifies the length of the *largest* block that may be encountered in the file. Also, for variable length files this value *should include* the length of the 4–byte block prefix. If block size is not specified, single record blocking is assumed. For fixed length files, this means a block size equal to the record size is assumed. For variable length files, this means that a block size equal to the record size plus 4 is assumed.
STDLABEL/ NOLABEL	This parm is optional and is allowed only for TAPE files. It specifies whether the tape file has standard labels (the default) or no labels.

```
Example: FILE: SALES-FILE  ATTR(DASD, 'SALEFIL', 80, 160)
```

The statement above defines a file named SALES–FILE. It is a SAM file on DASD, uses SALEFIL as the DLBL name, has fixed length 80–byte records, and has 160–byte blocks.

See page 331 for more examples and for a discussion of the ATTR parm.

DB2NAME('[qualifier.]name')

DB2 only. Specifies the name of the DB2 table or view to associate with this file. The table name must be enclosed in quotation marks or apostrophes. Generally the table name will be qualified. If it is not explicitly qualified, DB2 will assume an implicit qualifier, which will be the Authorization ID of the job which is executing Spectrum Writer. When this parm is present, no parms other than the filename are required in the FILE statement. The TYPE(DB2) parm is assumed.

> **Note:** A FILE statement is not required when working with DB2 inputs. You can specify the DB2NAME directly in your INPUT and READ statements. See page 393.

```
Example: FILE: PROJECT  DB2NAME('DSN8230.PROJ')
```

The above example defines a file that will be referred to in Spectrum Writer control statements as PROJECT. It refers to the DB2 table (or view) named DSN8230.PROJ.

DDNAME(ddname)

OS/390 only. The DDNAME parm specifies the name of a DD statement present in the JCL. This DD statement will identify the actual data set to be read. Spectrum Writer uses the DDNAME value to open an input file and read from it. The DDNAME parm can also be specified in the INPUT or READ statements (and omitted in the FILE statement.)

For a file to be used as an input file in a report, the DDNAME *must be specified* either in this statement, or in the INPUT or READ statement. For EXIT type files, the DDNAME parm is not required, but will be passed to the I/O Exit program if it is specified.

```
Example: FILE: SALES-FILE  DDNAME(SALESDD)
```

The above example defines a file named SALES–FILE. When records from this file are needed in a report, the DD named SALESDD in the JCL will be used.

EXITPARM('text')

This parm specifies any information that should be passed to user **data exit** programs (not to be confused with "I/O Exit" programs). (Most installations will not use data exits, and will not need this parm.) Anytime a data exit program is called by Spectrum Writer for a field within this file, the text string specified in this parm will be passed to it. The use of the EXITPARM parm is discussed in the section beginning on page 357.

```
Example: FILE: SALES-FILE  EXITPARM('XYZ')
```

The above example defines a file named SALES–FILE. If any fields within this file are defined as exit type fields, the text "XYZ" will be passed to their data exit programs each time they are called.

IOEXIT('program' ['parm'] [,TRACE])

EXIT files only. This parm provides the information necessary for Spectrum Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix I, "I/O Exits" (page 673).

> **OS/390 Note:** When this parm is present, a file type of EXIT is assumed and an explicit TYPE parm is not required.

VSE Note: When this parm is present, an ATTR parm specifying a type of EXIT and a RECSIZE is required.

'program' This parm is required. It specifies the name of the load module (OS/390) or phase (VSE) that Spectrum Writer will call in order to obtain records from the file.

'parm' This parm is optional. Each time the I/O Exit program is called by Spectrum Writer, the text specified in this parm is passed to the exit program. Typically this text is used to provide the exit program with any special information it may need in order to process the file. This parm can be up to 255 bytes in length.

TRACE This parm is optional. When specified, Spectrum Writer prints trace information in the control listing before and after each call to the I/O Exit. This information can be useful when developing and debugging a new I/O Exit program. The TRACE parm is normally not used in production runs.

```
Example: FILE: MASTER-FILE  IOEXIT('MYEXIT')
```

The above example specifies that a program named MYEXIT should be called whenever a record is needed from MASTER-FILE.

KEEPRDW

Meaningful only for non–VSAM, variable length files. This parm means that the 4–byte "record descriptor word" (RDW) at the beginning of each record should be considered a part of the record. The default is to treat the record as starting *after* the RDW. The use of this parm is discussed on page 352.

```
Example: FILE: PAYROLL-FILE  KEEPRDW
```

The above example specifies that the RDW should be kept when reading records from the PAYROLL–FILE. Thus, assuming that PAYROLL–FILE is a non–VSAM variable length file, a field defined as starting in column 1 would point to the 2–byte record length within the RDW.

LRECL(nnnnn/1000)

OS/390 only. Specifies the length of the largest record that might be found in the file. If this parm is not specified, Spectrum Writer assumes a default LRECL of 1000.

Note: Spectrum Writer uses this value only to determine the size of the I/O area that it allocates for use with the input file. Therefore it is not required that this value match the file's actual LRECL parm in the JCL or in the dataset's label information. In fact, if you suspect that a file's record size may grow in the future, you may want to specify a larger LRECL parm with some "growth" room in it. On the other hand, specifying an excessively large LRECL may result in higher CPU usage in some circumstances.

Note: When defining variable length SEQ files, the LRECL should include the length of the 4–byte "record descriptor word" (RDW) at the beginning of each record.

```
Example: FILE: SMF-FILE  LRECL(32767)
```

The above example defines a file named SMF–FILE. The LRECL parm specifies that records as large as 32,767 bytes may be encountered in the file. Spectrum Writer will reserve a 32,767 byte I/O area for reading records from this file.

NORMALIZE(fieldname, num-expr [, ...]) ...

Specifies how to normalize the file. (See "Using Normalization to Process Arrays" on page 237 for an explanation of file normalization.)

The NORMALIZE parm can be specified either here in the FILE statement or later in an INPUT or READ statement. If a file will be normalized for most runs, it is usually more convenient to put the parm here in the FILE statement. (NORMALIZE parms in the FILE statement can be cancelled, if desired, by specifying NONORMALIZE in the INPUT or READ statement.)

A full description of the NORMALIZE parm appears under the INPUT statement syntax on page 548.

NORMWHEN(conditional-expression)

Specifies which records from the file to normalize. (This parm is discussed in more detail under "Normalizing only Certain Records" on page 247.)

The NORMWHEN (and NORMALIZE) parms can be specified either here in the FILE statement or later in an INPUT or READ statement. If a file will be normalized for most runs, it is usually more convenient to put the parms here in the FILE statement.

A full description of the NORMWHEN parm appears under the INPUT statement syntax on page 549

STOPWHEN(conditional-expression)

Tells Spectrum Writer to stop reading this file when a certain condition is met.(when the file is used as the primary input for a run).

The STOPWHEN parm can be specified either here in the FILE statement or later in the INPUT statement. It is safer to specify this parm in each report's INPUT statement unless you are sure that the parm should apply to *all* runs from the file. (This is because any users who are not aware of the STOPWHEN parm in this statement could unknowingly get inaccurate results.)

If you do use a STOPWHEN parm in the FILE statement, it can be overridden by specifying NOSTOPWHEN in the INPUT statement.

A full description of the STOPWHEN parm appears under the INPUT statement syntax on page 551.

TYPE(SEQ/VSAM/DB2/EXIT)

OS/390 only. Specifies the type of access method to use when reading this file. If not specified, SEQ is assumed. Valid types are listed in the following table:

FILE TYPES ALLOWED IN THE TYPE PARM	
FILE TYPE	**DESCRIPTION**
SEQ	Standard sequential files, including tapes and disk datasets. The QSAM access method will be used. Sequential files can only be used as a primary input file (in the INPUT statement). They may *not* be used as an auxiliary input file (in a READ statement).

FILE TYPES ALLOWED IN THE TYPE PARM (CONTINUED)	
FILE TYPE	**DESCRIPTION**
VSAM	Any VSAM file, whether keyed or not. The IDCAMS access method will be used. Any kind of VSAM file can be used as a primary input file (in the INPUT statement). However, only *keyed* VSAM files may be used as auxiliary input files (in READ statements).
DB2	A DB2 table or view. This parm is optional, since Spectrum Writer assumes a TYPE of DB2 whenever the DB2NAME parm is used in the FILE statement. You may use this parm for documentation purposes if you wish.
EXIT	A file accessed via an I/O Exit program. This parm is optional, since Spectrum Writer assumes a TYPE of EXIT whenever the IOEXIT parm is used in the FILE statement. You may use this parm for documentation purposes if you wish.

```
Example: FILE: EMPL-FILE  TYPE(VSAM)  DDNAME(EMPLFILE)  LRECL(150)
```

The above example defines a VSAM file named EMPL–FILE.

FOOTNOTE Statement

PURPOSE

Specifies a footnote to print at the bottom of each page of the report. You may have as many FOOTNOTE statements in your report as you like. Each FOOTNOTE statement creates one line at the bottom of your report.

FOOTNOTE statements are ignored when producing PC files.

FEATURES

Use the FOOTNOTE statement to:

- specify the **contents** of the footnote lines (which can include literal text, data from input files, and special items like the current page number, date, time, etc.)
- specify how to **left align, center** and **right align** different parts of the same footnote
- specify the desired **width, display format**, and **justification** for data fields that appear in a footnote

LEARNING MORE

The complete syntax of the FOOTNOTE statement is shown on the following page. In addition, the following parts of the manual relate to the FOOTNOTE statement:

- the use of the FOOTNOTE statement is discussed beginning on page 175

SYNTAX

FOOTNOTE STATEMENT SYNTAX

```
FOOTNOTE: print-expression [/ print-expression] [/ print-expression]
```

Note: the syntax for the print-expressions — which is identical to that of TITLE statement print-expressions — is shown on page page 604.

Standard Spelling	Alternate Spellings
FOOTNOTE	FOOT

The FOOTNOTE statement consists of from one to three print expressions, separated by slashes. If a FOOTNOTE statement has no slashes, the single print expression will be centered under the report. If there is one slash, the first print expression will be left aligned and the second print expression will be right aligned under the report. If there are two slashes, the first print expression will be left aligned, the second one will be centered, and the third one will be right aligned. It is okay for one or more of the print expressions to be empty. Examples of using various combinations of print expressions and slashes is illustrated in the section beginning on page 168.

You may also use empty FOOTNOTE statements. An empty FOOTNOTE statement results in one blank footnote line.

The syntax of the FOOTNOTE statement is identical to that of the TITLE statement. See page 604 for the complete syntax information.

INCLUDEIF Statement

PURPOSE

Specifies which input records to include in the report or PC file. Each time a record is read from the primary input file, the expression in the INCLUDEIF statement is evaluated using the data from that record (and from any necessary auxiliary input file records). If the expression in the INCLUDEIF statement is *true*, then that record will be included in the run. If the expression is *not true*, then the record will not be included in the run. This process goes on until all records in the primary input file have been read and evaluated. The records that were included are then sorted and formatted into the desired report or output file.

Only one INCLUDEIF statement is allowed per report, but it may contain as many conditions as you like.

If **no INCLUDEIF** statement is specified, *all* records from the input file will be included in the run.

To include only a certain **number of records** from the input file in your report, use the MAXINPUT or MAXINCLUDE parms in the OPTIONS statement.

During the evaluation of the INCLUDEIF expression, if a test is attempted that involves a field with an **error condition**, the whole INCLUDEIF expression is automatically considered false and the input record is *not* included in the run. An example of such an error condition is when a packed–type field contains hex zeros or spaces. Other examples include computed fields where an overflow or divide–by–zero error occurred during their computation. However, see the OPTIONS statement's ZEROINVDATA, ZEROOVERFLOW and ZERODIVZERO parms. These options can be used to treat fields with error conditions as though they contained a zero value.

FEATURES

Use the INCLUDEIF statement to:

- select which input records will appear in a report or output file

LEARNING MORE

The complete syntax of the INCLUDEIF statement is shown on the following page. In addition, the following parts of the manual relate to the INCLUDEIF statement:

- a lesson on using the INCLUDEIF statement with reports begins on page 40
- a lesson on using the INCLUDEIF statement with PC files begins on page 93
- the syntax of conditional expressions (including examples) is described beginning on page 459

- suggestions on writing INCLUDEIF statements for maximum CPU efficiency are given in Appendix G, "Speed-Up Tips" (page 652)

SYNTAX

INCLUDEIF STATEMENT SYNTAX

```
INCLUDEIF:  conditional-expression
```

Standard Spelling	Alternate Spellings
INCLUDEIF	INCLUDE, INCL, INC

conditional–expression

Specifies one or more conditions to evaluate. As each record is read from the input file, the conditions specified in this expression are evaluated. If the conditional expression is *true*, the record is included in the run. Otherwise, the record is not included in the run. The syntax for conditional expressions is shown on page 459.

INPUT Statement

PURPOSE

Specifies which file should be used as the primary input for a report or PC file. One (and only one) INPUT statement is *required* in order to produce a Spectrum Writer report or PC file.

FEATURES

Use the INPUT statement to:

- specify the **name** of the primary input file for a report or PC file
- to automatically **copy** additional control statements from the Spectrum Writer Copy Library (typically to copy the FILE and FIELD statements that define the input file)
- specify a **record name** to be associated with records from this input file
- temporarily override certain aspects of the input **file definition** (such as the DDNAME, the file type, etc.)

LEARNING MORE

The complete syntax of the INPUT statement is shown on the following pages. In addition, the following parts of the manual relate to the INPUT statement:

- a lesson on using the INPUT statement begins on page 34
- information on using the INPUT statement with DB2 tables begins on page 393
- reading a file that is processed by a user I/O Exit is discussed in Appendix I, "I/O Exits" (page 673)

SYNTAX

INPUT STATEMENT SYNTAX

```
INPUT: filename
     [ ATTR(type, 'dlbl/tlbl' [,SYSnnn] [,F/V] ,recsize
            [,blksize] [,STDLABEL/NOLABEL])              (VSE only)   ]
     [ BUFND(nnn)                                        (VSAM only)  ]
     [ BUFNI(nnn)                                        (VSAM only)  ]
     [ CLEAR(SPACES/ZEROS/NO)                                         ]
     [ COPY(YES/NO)                                                   ]
     [ DB2NAME('[qualifier.]name')                       (DB2 only)   ]
     [ DDNAME(ddname)                                    (OS/390 only)]
     [ EXITPARM('text')                                               ]
     [ IOEXIT('program' [,'parm'] [TRACE])                            ]
     [ KEEPRDW                                                        ]
     [ KEYRANGE('begin' ['end'])                                      ]
     [ LIST(YES/NO)                                                   ]
     [ LRECL(nnnnn)                                      (OS/390 only)]
     [ NONORMALIZE                                                    ]
     [ NORMALIZE(fieldname, num-expr [, ...]) ...                     ]
     [ NORMWHEN(conditional-expression)                               ]
     [ NOSTOPWHEN                                                     ]
     [ ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)                        ]
     [ ORDERBY(fieldname [ASC/DESC] [, ...] )            (DB2 only)   ]
     [ RECNAME(name/filename)                                         ]
     [ SHOWFLDS(YES/NO)                                               ]
     [ STOPWHEN(conditional-expression)                               ]
     [ TYPE(SEQ/VSAM/DB2/EXIT)                           (OS/390 only)]
     [ WHERE(search-condition)                           (DB2 only)   ]
```

Standard Spelling	Alternate Spellings
DDNAME	DDN
DEFAULT	DEF
ERROR	ERR
EXITPARM	PARM
INPUT	INP
NO	N
NONORMALIZE	NONORM
NORMALIZE	NORM
NORMWHEN	NORMALIZEWHEN
ONNORMERROR	ONNORMERR
TYPE	TYP
WARNING	WARN
YES	Y

The filename is *required* in an INPUT statement, and must be the *first* item after the statement prefix. All other parms are optional and can appear in any order in the INPUT statement.

filename

Specifies the primary input file for the run. This file will be read sequentially — from beginning to end. (Certain options, such as KEYRANGE and STOPWHEN, can reduce the

number of records read.) Each record that passes the conditions in the INCLUDEIF statement (if there is one) will be included in the run.

The filename specified in this parm must have been defined in an earlier FILE statement. However, that FILE statement may be in a copy library member that is automatically copied at the time the INPUT statement is processed. This process is explained beginning on page 360.

```
Example: INPUT:  EMPL-FILE
```

The above example specifies that the file named EMPL–FILE will be the primary input file for the run.

ATTR(type, 'dlbl/tlbl' [,SYSnnn] [,F/V] ,recsize [,blksize] [,STDLABEL/NOLABEL])

VSE only. Specifies override file attributes to use for this file (for the current run only). For a complete description of the ATTR parm, see under the FILE statement syntax (page 531). For examples of using this parm, see page 331.

```
Example: INPUT: SALES-FILE  ATTR(DASD,'SALEFIL',80,160)
```

The statement above names SALES–FILE as the primary input file for the run. Regardless of how SALES–FILE was earlier described in a FILE statement, it will be treated in the current run as a SAM file on DASD, with SALEFIL as the DLBL name, with fixed length 80-byte records and with 160–byte blocks.

BUFND(nn)

VSAM files only. Specifies the number of "data buffers" that the VSAM access method should maintain when processing this input file. When this parm is not specified for a VSAM file, Spectrum Writer chooses a default number of data buffers to maintain.

> **Note:** According to the VSAM manual, increasing the number of data buffers to 4 or 5 (from VSAM's default of 2) should improve performance for sequential processing. At some point after that, excessive paging may cancel any benefit. You may wish to experiment with this parm if you have long–running, VSAM-intensive jobs.

```
Example: INPUT: EMPL-FILE  BUFND(5)
```

The above statement specifies that VSAM should allocate buffer space for 5 data control intervals when processing the EMPL–FILE.

BUFNI(nn)

VSAM files only. Specifies the number of "index buffers" that the VSAM access method should maintain when processing this input file. When this parm is not specified for a VSAM file, Spectrum Writer chooses a default number of index buffers to maintain.

> **Note:** According to the VSAM manual, VSAM's default number of index buffers (which is 1) should be sufficient for sequential processing of VSAM files that have index components. You may wish to experiment with this parm if you have long–running, VSAM–intensive jobs.

```
Example: INPUT: EMPL-FILE  BUFND(5)  BUFNI(2)
```

The above statement specifies that VSAM should allocate buffers for 5 data control intervals and 2 index control intervals when processing the EMPL–FILE.

CLEAR(SPACES/ZEROS/NO)

When processing certain types of input files, Spectrum Writer clears the entire I/O area to blanks before each read. This is to ensure that when a short record is read, it is not followed by leftover data from a previous longer record. For certain record layouts such leftover data could cause misleading results. Specifying CLEAR(NO) suppresses this clearing, which may result in improved performance. You might want to specify CLEAR(NO) if you are certain that any leftover data in the I/O area will not adversely affect your run. Specifying CLEAR(ZEROS) causes Spectrum Writer to initialize the I/O area to hex zeros (rather than blanks) before each read.

> **Note:** You can also specify the CLEAR parm in the FILE statement to avoid having to put it in the INPUT statement each time. The NOCLEARIO parm in the OPTIONS statement can be used to prevent clearing of *all files* in a run.

```
Example: INPUT: PAYROLL-FILE  CLEAR(NO)
```

The above statement names the PAYROLL–FILE as the input file for a run. Spectrum Writer will not clear its I/O area each time it reads a record from that file.

COPY(YES/NO)

Specifies whether control statements should be copied from the copy library before evaluating the file name. If the COPY parm is omitted and the file name has not been previously defined, the default is to attempt to perform a copy. Normally, the control statements that are copied will include the FILE and FIELD statements that describe the input file. This process is explained beginning on page 360.

If an attempt to copy records is unsuccessful (due to a missing copy library or a missing member) that is *not* considered an error. Normal control statement processing continues, without any copy being performed.

```
Example: INPUT:  EMPL-FILE  COPY(NO)
```

The above example specifies that no attempt should be made to copy records from the copy library.

DB2NAME('[qualifier.]name')

DB2 only. Specifies the name of the DB2 table or view that you wish to use as input for the run. For DB2 inputs, this parm is required unless the filename in this statement was defined in an earlier FILE statement that included a DB2NAME parm. The table name must be enclosed in quotation marks or apostrophes. Generally the table name will be qualified. If it is not explicitly qualified, DB2 will assume an implicit qualifier, which will be the DB2 Authorization ID of the job executing Spectrum Writer.

```
Example: INPUT:  PROJECT
                 DB2NAME('DSN8230.PROJ')
```

The above example specifies that the DB2 table named 'DSN8230.PROJ' should be used as the primary input "file" for the run. This input file has a Spectrum Writer file name of PROJECT. That is, other Spectrum Writer control statements that refer to this input file will refer to PROJECT (rather than to DSN8230.PROJ).

DDNAME(ddname)

OS/390 only. Specifies an override DDNAME to use when reading the input file (for the current run only). If omitted, the DDNAME will be taken from the FILE statement that defined the file. A DDNAME parm *must be present* in either the FILE statement or the INPUT statement.

```
Example: INPUT:  EMPL-FILE  DDNAME(TEMPDD)
```

The above example specifies that the TEMPDD DD statement in the JCL should be used to read the EMPL–FILE file, regardless of the DDNAME specified when the file was originally defined.

EXITPARM('text')

Most installations will not use exits, and will not need this parm. Specifies an override exit parm text. If this parm is omitted, the exit parm text (if any) will be taken from the FILE statement that defined the file. Exit parm text is passed to user data exit programs. Anytime a user data exit is called by Spectrum Writer for a field within this file, the text string specified in this parm will be passed to the exit. The use of this parm is discussed beginning on page 357.

```
Example: INPUT:  EMPL-FILE  EXITPARM('12345')
```

The above example specifies that the text '12345' should be passed to user data exit programs involving this file, regardless of the EXITPARM specified when the file was originally defined.

IOEXIT(‘program' [,'parm'] [,TRACE])

EXIT files only. Specifies override I/O Exit information for the input file. May also override the input file type (if it was something other than EXIT in the FILE statement). This parm provides the information necessary for Spectrum Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix I, "I/O Exits," on page 673.

OS/390 Note: When this parm is present, a file type of EXIT is assumed and an explicit TYPE parm is not required.

VSE Note: When this parm is present, an ATTR parm specifying a type of EXIT and a RECSIZE is required (in either this statement or the FILE statement).

'program' This parm is required. It specifies the name of the load module (OS/390) or phase (VSE) that Spectrum Writer will call in order to obtain records from the file.

'parm' This parm is optional. Each time the I/O Exit program is called by Spectrum Writer, the text specified in this parm will be passed to the exit program. Typically this text is used to provide the exit program with any special information it needs in order to process the file. This parm can be up to 255 bytes in length.

TRACE This parm is optional. When specified, Spectrum Writer will print trace information in the control listing before and after each call to the I/O Exit. This information can be useful when developing and debugging a new I/O Exit program. The TRACE parm is normally not used in production runs.

```
Example: INPUT: MASTER-FILE IOEXIT(‘MYEXIT')
```

The above example specifies that a program named MYEXIT should be called to read records from the primary input file MASTER-FILE.

KEEPRDW

Meaningful only for non–VSAM, variable length files. This parm means that the 4–byte "record descriptor word" (RDW) at the beginning of each record will be considered a part of the record. The default is to treat the record as starting *after* the RDW. The use of this parm is discussed beginning on page 352.

```
Example: INPUT:  HISTORY-FILE  KEEPRDW
```

The above example specifies that the RDW should be kept when reading records from the HISTORY–FILE. Thus, assuming that HISTORY–FILE is a non–VSAM variable length file, a field defined as starting in column 1 would point to the 2–byte record length within the RDW.

KEYRANGE(‘begin' ['end'])

KSDS VSAM files and EXIT files only. This parm specifies that only a certain range of records from the primary input file should be processed. Only records whose keys are greater than or equal to the ‘begin' value and less than or equal to the ‘end' value will be processed. If no ‘end' value is specified, the ‘end' value is assumed to be the same as the ‘begin' value.

The ‘begin' and ‘end' values in the KEYRANGE parm can each be a full or a partial (generic) key. Partial ‘begin' values are treated as if they were right-padded with hex zeros. Partial ‘end' values are treated as if they were right-padded with high values.

Speed-Up Tip: The use of this parm, where appropriate, can speed up your runs by eliminating unnecessary VSAM I/O.

Note: For files which are not KSDS VSAM files, you may be able to use the STOPWHEN parm to achieve similar savings in I/O processing.

```
Example: INPUT:  EMPL-FILE  KEYRANGE(‘03')
```

The above example specifies a partial key in the KEYRANGE parm. Only those records whose 3-byte EMPL-NUM (which is the key to this file) begins with "03" will be read from the EMPL-FILE.

```
Example: INPUT:  EMPL-FILE  KEYRANGE(‘032' ‘036')
```

The above example specifies that only records with keys between "032" and "036" (inclusive) should be read from the EMPL-FILE.

LIST(YES/NO)

Applies only if the COPY function is performed. The LIST parm specifies whether the copied control statements should be listed in the control listing. If no LIST parm is present, the default is to *not list* the copied statements.

Note: If an error is detected in any of the copied control statements, that statement *will* be listed, along with the error message, regardless of the value of this parm.

```
Example: INPUT:  EMPL-FILE  LIST(YES)
```

The above example specifies that the records copied from the copy library should be listed in the control listing.

LRECL(nnnnn)

OS/390 only. Specifies an override record length for the input file. This is the length of the largest record that might be found in the file. If this parm is omitted, the LRECL value (if

any) from the FILE statement is used. If no LRECL parm is found in either the FILE or the INPUT statement, a default LRECL of 1000 is assumed.

> **Note:** Spectrum Writer uses this value only to determine the size of the I/O area that it allocates for use with the input file. Therefore it is not required that this value match the file's actual LRECL parm in the JCL or in the dataset's label information. In fact, if you suspect that a file's record size may grow in the future, you may want to specify a larger LRECL parm with some "growth" room in it. On the other hand, specifying an excessively large LRECL may result in higher CPU usage in some circumstances.

```
Example: INPUT:  EMPL-FILE  LRECL(4000)
```

The above example specifies that a record as large as 4000 bytes long may be encountered in the EMPL–FILE file.

NONORMALIZE

Specifies that the input file should not be normalized. Any NORMALIZE parms specified in the FILE statement for this file (or in this INPUT statement) will be ignored.

```
Example: INPUT:  SALES-HISTORY  NONORMALIZE
```

The above example specifies that the sales history input file should not be normalized for the current run, even though there may be NORMALIZE parms in its FILE statement.

NORMALIZE(fieldname, num-expr [, ...]) ...

Specifies that the input file should be normalized. (See "Using Normalization to Process Arrays" on page 237 for an explanation of file normalization.) The fieldname must be the name of a field that defines the entire first occurrence of the array that is to be normalized. The numeric expression specifies how many occurrences the array contains. (This numeric expression is evaluated individually for each input record to determine the number of occurrences in that record.)

The NORMALIZE parm may contain any number of fieldname-numeric-expression pairs. Each pair identifies one array to be normalized. When multiple arrays are thus specified within a NORMALIZE parm, those arrays will be normalized *in parallel* (see "Normalizing Multiple, Non-Nested Arrays" on page 245).

In addition, you may have any number of NORMALIZE parms in the INPUT statement. When multiple NORMALIZE parms are present, they represent *nested* arrays (see "Normalizing Nested Arrays" on page 244). The last NORMALIZE parm in the INPUT statement specifies the most deeply nested array, and is normalized first.Then the array specified in the next-to-last NORMALIZE parm is normalized, and so on.

```
Example: INPUT:  SALES-HISTORY  NORMALIZE(SALE-ARRAY, NUM-SLOTS)
```

The above example specifies that the input records from the SALES-HISTORY file should be normalized. The first occurrence of the array being normalized is defined by the SALE-ARRAY field. The number of occurrences to be used from the array is contained in the NUM-SLOTS field.

> **Note:** If only certain records from the input file should be normalized, use a NORMWHEN parm before the NORMALIZE parm.

Note: NORMALIZE (and NORMWHEN) parms can also be specified in FILE statements. If you have a file that will be normalized most of the time, you may want to put the NORMALIZE parms in the FILE statement.That way you won't need to include the parms in every report request. Spectrum Writer will automatically normalize the file according to the NORMALIZE parms from the FILE statement. If you later do *not* want to normalize such a file for a particular report, just use the NONORMALIZE parm in the INPUT statement. Spectrum Writer will then ignore the NORMALIZE parms from the FILE statement for that run.

NORMWHEN(conditional-expression)

Specifies which records from the input file to normalize. (This parm is discussed in more detail under "Normalizing only Certain Records" on page 247.) When the conditional expression is true for a record, then all subsequent NORMALIZE parms (up until the next NORMWHEN parm) will be processed. If the conditional expression is false, the subsequent NORMALIZE parms will not be processed for that input record. (Any NORMALIZE parms that are not preceded by a NORMWHEN parm are processed for every input record.)

```
Example:  INPUT: BATCH-FILE
                 NORMWHEN(RECORD-TYPE = 'HDR') NORMALIZE(STATUS-ARRAY, 5)
                 NORMWHEN(RECORD-TYPE = 'DET') NORMALIZE(CUSTOMER-ARRAY, 8)
```

The above statement tells Spectrum Writer to normalize the STATUS-ARRAY only for those records where the RECORD-TYPE field contains 'HDR'. And the CUSTOMER-ARRAY will be normalized only for those records where the RECORD-TYPE field contains 'DET'. Records with any other value in the RECORD-TYPE field will not be normalized at all.

NOSTOPWHEN

Specifies that any STOPWHEN parm in this file's FILE statement (or in this INPUT statement) should be ignored for the current run.

```
Example: INPUT:  SALES-FILE  NOSTOPWHEN
```

The above example specifies that STOPWHEN processing is not wanted for the current run, even though there may be a STOPWHEN parm in the FILE statement for SALES-FILE.

Note: The NOSTOPWHEN parm in this statement does *not* cancel STOPWHEN parms specified in the OPTIONS statement

ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)

Specifies how normalization errors in this file should be treated. (For examples of normalization errors, see "Normalization Errors" on page 248.)

By default, normalization error messages are treated as informational messages only. When a normalization error occurs, Spectrum Writer processes the physical record, and then skips the normalization for that record. If you want normalization errors to be treated as more serious errors, use the ONNORMERROR parm.

Specify WARNING in this parm to change the control listing message from "informational" to a "warning" (which also sets the job completion code to 4).

Specify ERROR in this parm to change the control listing message from "informational" to an "error" (which also sets the job completion code to 8).

Specify STOP in this parm to have Spectrum Writer halt the run immediately when a normalization error occurs. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: You can also specify an ONNORMERROR parm in the OPTIONS statement, if you want it to apply to *all* of the normalized files used in a run.

Note: When normalization errors occur, Spectrum Writer also prints a dump of the record in error. You can use the MAXNORMDUMP option (in an OPTIONS statement) to control how many such dumps appear in your control listing.

```
Example: INPUT:  SALES-HISTORY  NORMALIZE(SALE-ARRAY, NUM-SLOTS)  ONNORMERROR(ERROR)
```

The above example specifies that any normalization errors in the SALES-FILE should be treated as an "error."

ORDERBY(fieldname [ASC/DESC] [, ...])

DB2 only. This parm is optional and not normally used in the INPUT statement. If this parm is omitted, DB2 passes the rows from the DB2 table to Spectrum Writer in an "arbitrary" order. This is not normally of any consequence, as Spectrum Writer then sorts the selected rows according to the SORT statement before producing your report or PC file. Use this parm if you want to specify the order in which the rows from the DB2 table should be passed to Spectrum Writer. The contents of this parm is one or more column names from the DB2 table, optionally separated with commas. You may also include the DB2 keywords ASC or DESC after the column names. This parm is discussed in more detail on page 399.

```
Example: INPUT: PROJECT
                DB2NAME('DSN8230.PROJ')
                ORDERBY(DEPTNO, PROJNAME)
```

The above example would cause DB2 to pass the rows from the project table to Spectrum Writer in department number order, with "ties" being passed in project name order.

RECNAME(name/filename)

Specifies a record name to use when referring to fields in this input file. This is especially useful when you will be reading multiple records from the same input file (by using a READ statement in addition to the INPUT statement). The RECNAME parm (in each statement) can be used to assign unique names to each record read from the file. You may give the record any name you like, within the rules governing names given on page 446. The use of the RECNAME parm is discussed on page 228.

If no RECNAME parm is specified, the *filename* is used as the record name.

```
Example: INPUT:  EMPL-FILE  RECNAME(EMP)
```

The above example specifies that the records read from the EMPL–FILE file will be named EMP. Assume that a field named DATE exists in both this file and in some other input file. You can use the record name EMP to indicate that you are referring to the DATE field from the EMPL–FILE, like this:

```
Example: COLUMNS:  EMP.DATE
```

SHOWFLDS(YES/NO)

Specifies whether Spectrum Writer should print a list of all fields that have been defined for the input file. (For DB2 inputs, the DB2 columns defined for the DB2 table are listed.) This

list appears immediately after the INPUT statement in Spectrum Writer's control statement listing. The list will include the data type of each field (character, numeric, date, time or bit). Use this parm if you aren't sure of the names or spellings of the fields (or DB2 columns) in your input file.

```
Example: INPUT: PROJECT
                DB2NAME('DSN8230.PROJ')
                SHOWFLDS(YES)
```

The above statement causes a list to be printed showing each DB2 field ("column") defined for the DSN8230.PROJ table.

STOPWHEN(conditional-expression)

Tells Spectrum Writer to stop reading the primary input file when a certain condition is met. Use this parm to reduce I/O processing (and run time) if you know that your report will not need records after a certain point in your input file (and if that point can be specified in a conditional expression).

```
Example: INPUT: SALES-FILE
                STOPWHEN(EMPL-NUM > '039')
```

The above statement tells Spectrum Writer to stop reading the primary input file when it encounters a record whose EMPL-NUM field is greater than '039'. When Spectrum Writer reads a record whose EMPL-NUM is greater than '039', it will ignore that record and act as if it has hit EOF on that file. No more records will be read from the file. The report will be produced based on the records read up to that point.

Note: The STOPWHEN parm can also be specified in the FILE statement. However, it is safer to specify this parm in each report's INPUT statement unless you are sure that the parm should apply to *all* runs from the file. (Any users who were unaware of the STOPWHEN parm in the FILE statement could unknowingly get inaccurate results.) If you do use a STOPWHEN parm in the FILE statement, it can be overridden by specifying NOSTOPWHEN in the INPUT statement. Spectrum Writer will then ignore the STOPWHEN parm from the FILE statement for that run.

Note: The STOPWHEN parm in the INPUT statement (or in the FILE statement) affects the input file processing for *all* reports in a multi-report run. However, you can also specify a STOPWHEN parm in the OPTIONS statement. A STOPWHEN parm in an OPTIONS statement applies only to the report that is currently being defined. Thus, if you want each report in a multi-report run to have its own STOPWHEN condition (or no STOPWHEN condition), do not put a STOPWHEN parm in the INPUT statement. Instead, put each report's STOPWHEN parm in an OPTIONS statement within that report's definition statements.

Note: If your input file is a KSDS VSAM file and you want to read only a certain range of records, you should use the KEYRANGE parm instead of the STOPWHEN parm.

TYPE(SEQ/VSAM/DB2/EXIT)

OS/390 only. Specifies an override file type for the input file. If this parm is omitted, the file type will be taken from the FILE statement that defined the file. A complete list of file types is given under the FILE statement description on page 536.

```
Example: INPUT:  EMPL-FILE  TYPE(VSAM)
```

The above example specifies that the VSAM access method should be used when reading the EMPL–FILE file, regardless of the file type specified when the file was originally defined.

WHERE(search–condition)

DB2 only. This parm is optional. If this parm is omitted, DB2 will pass all rows in the DB2 table to Spectrum Writer. (Spectrum Writer will then decide which of those rows to use based on the INCLUDEIF statement, if any.) Use this parm to specify a "search condition" for DB2 to use in deciding which rows from the DB2 table to pass to Spectrum Writer. The syntax of the search–condition is generally the same as DB2's syntax for the WHERE clause in a DB2 SELECT statement. The use of this parm is discussed in the section beginning on page 397. The precise syntax rules for the WHERE parm are given on page 405.

```
Example: INPUT: PROJECT
                DB2NAME('DSN8230.PROJ')
                WHERE(DEPTNO = 'D21')
```

In the example above, the WHERE parm causes DB2 to return to Spectrum Writer only those rows from the project table whose DEPTNO field is equal to "D21". Those are the only rows that could then appear in the Spectrum Writer report or PC file. An INCLUDEIF statement could be used to reduce even further the number of rows that are actually included in the run.

NOTES

How the Primary Input File is Processed

The file specified in the INPUT statement is called the **primary input file** for a run. Each run must have one (and only one) primary input file.

Spectrum Writer opens this file for sequential input starting with the first record in the file. (Or starting with the first record in the specified key range, if the KEYRANGE parm was used.) Records are then read sequentially from the input file until any one of the following occurs:

- the EOF (end of file) condition occurs
- the last record in the specified KEYRANGE has been read
- a record is read for which the STOPWHEN condition is true

After each record is read, the conditions specified in the INCLUDEIF statement (if any) are evaluated, using the data from that record (as well as from any linked auxiliary input file records that may be needed). Based on these INCLUDEIF conditions, the record will either be included in the run or will be ignored. If the record is to be included, a smaller internal sort record is built containing just the necessary data from the input record (and from any required auxiliary input records). This internal sort record is then passed to the sortin phase of the system sort program and the next sequential record is read from the primary input file.

This process is repeated until all records in the primary input file have been evaluated. Then, the sorted sort records are retrieved from the sortout phase of the system sort

program. The final report (or output file) is then written from the information in these sorted internal records.

NEWOUT Statement

PURPOSE

The NEWOUT statement is used to create an additional output (report or PC file) during the same run. All control statements following the NEWOUT statement apply only to the new output.

You may have more than one NEWOUT statement in a run.

LEARNING MORE

The following parts of the manual relate to the NEWOUT statement:

- for a discussion of multiple output runs, see "Creating Multiple Reports in a Single Run" (page 289)

SYNTAX

NEWOUT STATEMENT SYNTAX

```
NEWOUT:
```

Standard Spelling	Alternate Spellings
NEWOUT	NEWOUTPUT

The NEWOUT statement has no parms.

OPTIONS Statement

PURPOSE

Specifies various special options. You may specify as many options as you like on a single OPTIONS statement. In addition, you may have as many separate OPTIONS statements as you like.

OPTIONS statements normally must appear before all other control statements.

FEATURES

Use the OPTIONS statement to:

- specify options that affect the overall appearance of a report
- specify that a PC or mainframe file should be created rather than a report
- change defaults settings used in producing a report
- limit the amount of processing performed, for test runs
- specify printer setup texts

LEARNING MORE

The complete syntax of the OPTIONS statement is shown on the following pages. In addition, certain individual options are discussed and illustrated in other parts of this manual. To see if additional information is available about a specific option, check under the name of the option in the Index to this manual.

SYNTAX

OPTIONS STATEMENT SYNTAX

```
OPTIONS:
     [  ASCIITABLE(‘text’)                                                    ]
     [  ASMLIB('library.sublibrary')                          (VSE only)      ] †
     [  AUTOSORT                                                              ]
     [  CENTURY(nn/50)                                                        ] †
     [  COBLIB('library.sublibrary')                          (VSE only)      ] †
     [  COLHDGONCE                                                            ]
     [  COLSEP('text')                                                        ] †
     [  COLSPACE(nnn/1)                                                       ]
     [  DATEDELIM('char'/'/')                                                 ]
     [  DB2PLAN('plan'/'SPECTnnn')                            (DB2 only)      ]
     [  DB2SUBSYS('subsystem')                                (DB2 only)      ] †
     [  DDMMYYLIT                                                             ] †
     [  DETAIL(nnnnn)                                                         ]
     [  EBCDICTABLE(‘text’)                                                   ]
     [  EMPTYCC(nn)                                                           ]
     [  EMPTYMSG('text')                                                      ]
     [  FORMAT(display-format [,display-format] ... )                         ]
     [  HEADINGSEP('char'/'|')                                                ]
     [  HGCOLHDG                                                              ]
     [  HTML[('title')]                                                       ]
     [  HTMLAID[(YES/NO)]                                                     ]
     [  KEEPRDW                                                               ] †
     [  LEFTMARGIN(nnn/0)                                                     ]
     [  MAINFRAME                                                             ]
     [  MAXINCLUDE(nnnnn)                                                     ]
     [  MAXINPUT(nnnnn)                                                       ]
     [  MAXINVSHOW(nnnnn/10)                                                  ] †
     [  MAXNORMDUMP(nnnnn/10)                                                 ] †
     [  MAXPAGES(nnnnn)                                                       ]
     [  MAXPRINT(nnnnn)                                                       ]
     [  MEMTYPE('type'/'SPECTWTR')                            (VSE only)      ] †
     [  MISSOFFSET                                                            ] †
     [  MULTICOLHDG                                                           ]
     [  NOBLOCKSIZE                                           (OS/390 only)   ]
     [  NOCC                                                                  ]
     [  NOCHECK                                                               ]
     [  NOCLEARIO                                                             ] †
     [  NOCOLHDGS                                                             ]
     [  NOGRANDSPACES                                                         ]
     [  NOGRANDTOTAL                                                          ]
     [  NOMAXMSG                                                              ]
     [  NOOVERPRINT                                                           ]
     [  NOSORTSIZE                                                            ]
     [  NOSYSINLIMIT                                                          ] †
     [  NOTEMPTYCC(nn)                                                        ]
     [  NOTITLES                                                              ]
     [  NOUNDERSCORES                                                         ]
     [  ONIOERROR(DEFAULT/ERROR/STOP)                                         ] †
     [  ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)                               ] †
     [  OUTATTR(type [,'dlbl/tlbl'][,SYSnnn][,recsize][,blksize])   (VSE only)   ]
     [  OUTDDN(ddname)                                        (OS/390 only)   ]
```

(Continued on next page)

OPTIONS STATEMENT SYNTAX (CONTINUED)

```
     [  OUTLRECL(nnnnn)                                          (OS/390 only)   ]
     [  OUTTYPE(SEQ/VSAM)                                        (OS/390 only)   ]
     [  PAGELENGTH(nnn/60)                                                       ]
     [  PC/OUTPUT/ACCESS/COREL/CSV/DBASE3/DBASE4/EXCEL
        /FOXPRO/HARVARD/LOTUS/MS-WORKS/PARADOX/QUATTRO/RBASE                     ]
     [  POSTSCRIPT('text')                                                       ]
     [  PRESCRIPT('text')                                                        ]
     [  PRTSETUP('text')                                                         ]
     [  PRTSHEET('text')                                                         ]
     [  QCHAR('char'/'"')                                                        ]
     [  SCALEPIC                                                                 ]
     [  SINGLE/DOUBLE/TRIPLE                                                     ]
     [  SKIPBLANKDET                                                             ]
     [  SKIPZERODET                                                              ]
     [  SORTDD('prefix')                                                         ]
     [  SORTNAME('program'/'SORT')                                               ]
     [  SORTOPT('text')                                                          ]
     [  SORTSIZE(nnnn/256)                                                       ]
     [  SORTWORKNUM(n/0)                                         (VSE only)      ]
     [  SPLITDETAIL                                                              ]
     [  STCKADJ(nn)                                                              ] †
     [  STOPWHEN(conditional-expression)                                         ]
     [  SUBLIB('library.sublibrary')                             (VSE only)      ] †
     [  SUMMARY                                                                  ]
     [  TIMEDELIM('char'/':')                                                    ]
     [  TITLEONCE                                                                ]
     [  ZERODIVBYZERO                                                            ] †
     [  ZEROINVDATA                                                              ] †
     [  ZEROOVERFLOW                                                             ] †
```

Standard Spelling	Alternate Spellings
DB2SUBSYS	DB2SUBSYSTEM
DDMMYYLIT	DDMMYYYYLIT
DETAIL	DET
DOUBLE	DOUBLESPACE
FORMAT	FMT
HEADINGSEP	HDGSEP
LEFTMARGIN	LEFTMARG
MAINFRAME	MAIN
MAXINCLUDE	MAXINCL, MAXINC
MAXINPUT	MAXINP
MAXPAGES	MAXPAGE
MAXPRINT	MAXPRT
MEMTYPE	MEMBERTYPE
NOBLOCKSIZE	NOBLKSIZE
NOCOLHDGS	NOCOLHDG
NOGRANDSPACES	NOGRANDSPACE
NOGRANDTOTAL	NOGRANDTOTALS, NOGRANDTOT, NOGRANDTOTS, NOGRAND

(Continued on next page)

OPTIONS STATEMENT SYNTAX (CONTINUED)

Standard Spelling	Alternate Spellings
NOOVERPRINT	NOOVERPRT
NOTITLES	NOTITLE
NOUNDERSCORES	NOUNDERSCORE, NOUNDER
OPTIONS	OPTION, OPTS, OPT
PAGELENGTH	PAGELGTH, PAGELEN, PAGEL
SINGLE	SINGLESPACE
SKIPBLANKDET	SKIPBLKDET
SPLITDETAIL	SPLITDET
SUBLIB	SUBLIBRARY
TITLEONCE	TITLESONCE
TRIPLE	TRIPLESPACE
ZERODIVBYZERO	ZERODIVZERO, ZERODIVZ
ZEROINVDATA	ZEROINV
ZEROOVERFLOW	ZEROOVER

Notes

† means that the option is a **"run wide"** option. The option applies to *all* reports and output files created in the run. It does not matter which output was being defined when the OPTION statement was encountered among the control statements.

All other options are **"by output."** Those options are in effect only for the report or output file that is being defined when the OPTION statement is encountered. The option does not affect any other report or output file created in the same run.

ASCIITABLE('text')

Use this option to specify your own translation table to be used for EBCDIC-to-ASCII translations. (Such translations are performed when the ASCII parm is specified in a print-expression, as well as for the #ASCII built-in function.) The text parm for this option must be a string that is exactly 256 bytes long. For convenience, you can split this 256-byte string into as many smaller strings as you like. This string tells Spectrum Writer what value to return for each of the 256 possible byte values it could encounter when translating a character string. If this option is not specified, Spectrum Writer uses a default ASCII translation table.

```
Example: OPTION: ASCIITABLE(X'000102030405060708090A0B0C0D0E0F'
                            X'101112131415161718191A1B1C1D1E1F'
                            X'202122232425262728292A2B2C2D2E2F'
                            X'303132333435363738393A3B3C3D3E3F'
                            X'404142434445464748494A4B4C4D4E4F'
                            X'505152535455565758595A5B5C5D5E5F'
                            X'606162636465666768696A6B6C6D6E6F'
                            X'707172737475767778797A7B7C7D7E7F'
                            X'808182838485868788898A8B8C8D8E8F'
                            X'909192939495969798999A9B9C9D9E9F'
                            X'A0A1A2A3A4A5A6A7A8A9AAABACADAEAF'
                            X'B0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF'
                            X'C0C1C2C3C4C5C6C7C8C9CACBCCCDCECF'
                            X'D0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF'
                            X'E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF'
                            X'F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF')
```

The above example merely serves to illustrate the syntax of the ASCIITABLE option. This example uses 16 hex strings of 16 bytes each to provide the necessary 256-byte table. (The values shown in the example would cause the #ASCII function to simply return the same operand without change.)

ASMLIB('library.sublibrary')

VSE only. Specifies the default sublibrary to copy members from while in the scope of an ASM statement.

```
Example: OPTION: ASMLIB('TEST.COPYASM')
```

The above statement means that COPY statements appearing within the scope of an ASM statement will copy members from the TEST.COPYASM sublibrary by default. This default can be overridden, however, by specifying a sublibrary name directly in the COPY statement.

AUTOSORT

When no SORT statement is specified, the AUTOSORT option tells Spectrum Writer to sort the report or output file on the first 5 fields named in the COLUMNS statement. When an explicit SORT statement is used, the AUTOSORT option tells Spectrum Writer to add up to 5 "tie–breaker" sort fields to the fields named in the SORT statement. The tie–breaker fields will be the first 5 fields named in the COLUMNS statement (not considering those fields explicitly named in the SORT statement).

CENTURY(nn/50)

Specifies the century cutoff year for all YY dates in the input files and in the control statements. This option tells Spectrum Writer which century a 2–digit year belongs to. Any year *below* the specified value is considered to be in the 21st century (20YY). Any year greater than or equal to the specified value is considered to be in the 20th century (19YY).

The default value of 50 means that:

- 2–digit years from 00 to 49 are treated as 2000 though 2049, and
- 2-digit years from 50 to 99 are treated as 1950 through 1999.

Note: This option does not affect the way dates with 4–digit years are processed.

```
Example: OPTION: CENTURY(20)
```

The above example specifies that all 2-digit dates with a year less than 20 are in the 21st century. Thus, the date 8/31/19 would mean August 31, 2019. However, 8/31/20 would mean August 31, 1920.

COBLIB('library.sublibrary')

VSE only. Specifies the default sublibrary to copy members from while in the scope of a COBOL statement.

```
Example: OPTION: COBLIB('TEST.COPYCOB')
```

The above statement means that COPY statements appearing within the scope of a COBOL statement will copy members from the TEST.COPYCOB sublibrary by default. This default can be overridden, however, by specifying a sublibrary name directly in the COPY statement.

COLHDGONCE

Print column headings only once, at the very beginning of the report or PC file. This is Spectrum Writer's default when creating many type of PC files. This option also suppresses titles, footnotes and all page break logic. (Use the similar TITLEONCE option if you want the report titles to print once along with the column headings.)

COLSEP('text')

Specifies a default column separator text. This text will appear between each column in the report. Normally, the column separator text is a single blank space.

This option is useful when creating output files (especially PC files). In that case, use this option to specify a "delimiter" character (such as a comma, or a "tab" character) to separate the fields in the output record.

```
Example: OPTIONS:  COLSEP(',')
```

The above statement causes the fields ("columns") in the output record to be separated by commas.

Note: Specifying this option also causes the COLSPACE option to be set to the length of the COLSEP text.

COLSPACE(nnn/1)

Specifies the default number of spaces to leave between columns in the report. (This default spacing factor can be overridden directly in the COLUMNS statement.) The normal default is to leave one blank space between each report column.

This option is also useful when creating mainframe output files. You may then want to specify COLSPACE(0) to eliminate all blanks between the fields in the output records.

Note: This parm only affects spacing between the actual report columns — in the detail lines of the report. It does *not* change the default spacing factor (of one) that is used in titles, footnotes, control break footings, etc.

Note: Specifying the COLSEP option also changes the COLSPACE value.

DATEDELIM('char'/'/')

This option lets you specify any character you choose to be used as the delimiter when formatting dates. This delimiter will be used with all date display formats that use a delimiter. The default date delimiter is a slash (/). For example, to format all dates using dots rather than slashes, you would specify:

```
Example: OPTIONS: DATEDELIM('.')
```

This would cause the MM-DD-YY display format to appear as "12.31.99" and the DD-MM-YYYY format to appear as "31.12.1999".

Note: Use of this parm does *not* affect the way Spectrum Writer recognizes date literals in the control statements. Date literals must always be written using slashes as delimiters.

DB2PLAN('plan'/'SPECTnnn')

DB2 only. Specifies the DB2 plan name to use. This parm is needed only if the default plan name was *not* used during installation of Spectrum Writer's DB2 Option (see page 409).

Spectrum Writer assumes that you use a plan name of "SPECTnnn", where nnn is the Spectrum Writer version number. (Thus, for release 3.0 of Spectrum Writer, a plan name of SPECT300 is assumed.) If you used a different plan name to bind Spectrum Writer in your shop, you must tell Spectrum Writer your plan name via the DB2PLAN option. Enclose the plan name in quotation marks or apostrophes. For example, if you bound Spectrum Writer with a plan name of XYZ12345, you would need to use the following statement:

```
OPTION: DB2PLAN('XYZ12345')
```

DB2SUBSYS('subsystem')

DB2 only. Specifies the name of the DB2 subsystem to use for the run. This option is required for any run that uses DB2 data. Enclose the subsystem ID in quotation marks or apostrophes.

```
Example: OPTIONS: DB2SUBSYS('DB2T')
```

The above statement causes Spectrum Writer to use the DB2 subsystem named DB2T for all DB2 requests in the run.

DDMMYYLIT

Indicates that all date literals used in the control statements are in DD/MM/YY or DD/MM/YYYY format.

```
Example: OPTIONS:   DDMMYYLIT
         ...
         INCLUDEIF: SALES-DATE < 31/12/1996
```

The above OPTIONS statement specifies that any date literals in the control statements are in DD/MM/YY (or DD/MM/YYYY) format. In the INCLUDEIF, we select all records whose SALES–DATE field is before December 31, 1996.

Note: The slash (/) is always used as the delimiter in date literals. The DATEDELIM option, if any, only changes the way dates are formatted in the output— not the way date literals must be written in the control statements.

DETAIL(nnnnn)

Specifies how many detail lines should be printed within each control break. (If no control breaks are used, it specifies how many detail lines to print in the whole report.) The default is to print *all* detail lines.

You may specify DETAIL(0) to suppress all detail print lines. In that case you would see only the lines printed at control breaks and at Grand Total time.

This option is useful for printing "Top Ten Sales in each Department" type of reports. It is also helpful when developing new reports that have lots of detail lines. Use this option to print just a few detail records for each control group while you develop the new report. This will keep your trial run down to a smaller, more convenient size. Remove the option when your are ready for the final run.

```
Example: OPTIONS: DETAIL(10)
         INPUT:   EMPL-FILE
         TITLE:   'TOP 10 SALES PER DEPARTMENT'
         COLUMNS: LAST-NAME FIRST-NAME TOTAL-SALES
         SORT:    DEPT-NUM TOTAL-SALES(DESC)
         BREAK:   DEPT-NUM
```

The above example produces a report that lists the top 10 sales people in each department, in descending sales volume order.

EBCDICTABLE('text')

Use this option to specify your own translation table to be used by the #EBCDIC built-in function. The text parm for this option must be a string that is exactly 256 bytes long. For convenience, you can split this 256-byte string into as many smaller strings as you like. This string tells Spectrum Writer what value to return for each of the 256 possible byte values it could encounter when performing the #EBCDIC built-in function. If this option is not specified, Spectrum Writer uses a default EBCDIC translation table.

Example: See the example under the ASCIITABLE option (page 558) which has the same syntax.

EMPTYCC(nn)

Sometimes it is important to know when Spectrum Writer did not write any records to the report or output file. This option lets you specify a special completion code for Spectrum Writer to issue for such "empty" runs.

```
Example: OPTION: EMPTYCC(12)
```

The above statement tells Spectrum Writer to set the step completion code to 12 if no input records pass the INCLUDEIF conditions for the report. If one or more records are included, the standard completion code will be used (normally 0, unless an error was encountered).

Note: This option can not be used to *lower* Spectrum Writer's standard completion code (which may be 4 or 8 if warning or error messages were issued).

Note: See also the related EMPTYMSG and NOTEMPTYCC options.

EMPTYMSG('text')

Normally, when no records are included in a report, the report output file is simply left empty. (No titles, column headings or Grand Totals are written to it.) This can be confusing to some users, who might be unsure if the report is just empty, or if an error occurred while running the job. Use this parm to specify a one-line message that Spectrum Writer should write to the output file when no records have been selected.

```
Example: OPTIONS: EMPTYMSG('NO TRANSACTIONS FOUND OVER $1,000,000')
```

The above example specifies that the indicated message should be written to the output file when no records pass the INCLUDEIF tests for the report. When one or more records are included, this message will not appear in the report.

Note: see also the related EMPTYCC option.

FORMAT(display–format [,display–format] [,display–format] [,display–format])

Specifies one or more display formats to be used as default display formats. You may specify one character–type display format, one numeric–type display format, one date-type display format, and one time–type display format. You may specify any or all of these, in any order. (A complete list of display formats is found in Appendix B, "Display Formats" on page 617.) The display formats specified in this option become the *default* display format for all fields of the associated data type. This option is especially useful when

creating output files. For example, when creating a comma delimited PC file, you might use the following statement:

```
OPTIONS: FORMAT(QCHAR, Q-MM-DD-YY, Q-HH-MM-SS, NOCOMMA)
```

The above statement would cause the QCHAR display format to be used for all character fields (enclosing the character data in quotation marks). All dates would be formatted as MM/DD/YY, also enclosed in quotation marks. All times would be formatted as HH:MM:SS, also enclosed in quotation marks. And all numeric fields would be formatted in the NOCOMMA display format — without using commas to separate thousands, millions, etc.

When the FORMAT option is used, you may still specify an override display format for any particular item directly in the COLUMNS statement (or TITLE statement, etc.) The FORMAT option just changes the *default* display format used when no explicit display format is given.

Note that the output file options (PC, EXCEL, MAINFRAME, etc.) also change one or more of the default display formats.

> **Note:** When the CHARACTER or HEX display format is specified *alone* in the FORMAT option, it applies to data of *all* types.
>
> Example: `OPTIONS: FORMAT(HEX)`
>
> The above statement would cause all character, numeric, date and time fields to appear in hex format. If you want the HEX or CHARACTER display format to apply *only* to character fields, you should also specify numeric, date and time display formats *after* the CHARACTER or HEX format in the FORMAT parm.
>
> Example:
> ```
> OPTIONS: FORMAT(HEX, NUMERIC, MM-DD-YY, HH-MM-SS)
> ```
>
> The above statement would cause all character fields to be formatted in HEX format, and all numeric, date and time fields to be formatted the way they normally would be.

HEADINGSEP('char'/'¦')

Specifies the character that will be used to separate column heading texts into separate lines. The default heading separator character is the vertical bar (|).

> **Note:** The vertical bar is the "Shift 1" key on most mainframe terminals. When working at a PC running terminal emulation software, you will probably not see a key with this symbol on it. Some terminal emulator programs use the "pipeline" key as the vertical bar key. Some others use the right–hand square bracket key "]" for this purpose.

Example:
```
OPTIONS: HEADINGSEP('/')
         COLUMNS: LAST-NAME('EMPLOYEES/LAST/NAME')
```

The above example specifies that the slash character (/) should be used as the heading separator character. The COLUMNS statement specifies an override column heading text using slashes. The slashes would cause the three words in the column heading to appear on three separate lines.

HGCOLHDG

Specifies that "Harvard Graphics" style column headings are wanted. (This is also the default when the HARVARD option is specified.) This option causes the column headings to appear in a single line in the output file (rather than being split onto multiple lines). The "blank" line that normally separates the column headings from the actual data is also suppressed. This option is useful when the PC program which will be importing your output file expects the first line of input to contain a "legend" for the data in the subsequent lines.

HTML[('title')]

Tells Spectrum Writer to wrap standard HTML code around the report. This creates a Web report that can be viewed on Web browsers, such as Internet Explorer and Netscape Navigator. You can also specify an optional HTML title for the Web page. Specifying the HTML option also turns on the HTMLAID option (see below). The use of these options is discussed in Chapter 5, "How to Make a Web Report."

```
Example: OPTIONS: HTML('SALES REPORT BY REGION')
```

HTMLAID[(YES/NO)]

The HTMLAID option tells Spectrum Writer that you will be putting your own HTML tags within the report and that Spectrum Writer should recognize and support those tags. This option itself does not cause Spectrum Writer to add any HTML codes to your report. This option is implied by the HTML option. Therefore, you do not need to specify the HTMLAID option if you have specified the HTML option. (If you specify the HTML option and do *not* want the HTMLAID option, specify HTMLAID(NO) to turn it off.) The use of this option is discussed in Chapter 5, "How to Make a Web Report."

```
Example: OPTIONS: HTMLAID
```

Following are the specific actions that Spectrum Writer takes when the HTMLAID option is in effect:

1. When an HTML-format literal is specified as an item in a COLUMNS statement, Spectrum Writer also copies that HTML literal into the same location in the default total lines (that is, the lines that print by default at control breaks and at the Grand Total). The purpose of this is to keep the columns in the default total lines aligned with the body of the report. It also causes the same HTML formatting information that is applied to a particular report column to be applied to the total for that column (if it is a totalled column).
2. When an HTML-format literal is specified as an item in a COLUMNS statement and no explicit column heading is specified for it, Spectrum Writer uses the HTML literal itself as its own column heading. The HTML literal will be propagated into all column headings lines, including even the underscore line. The purpose of this is to keep the column headings aligned with the body of the report. It also causes the same HTML formatting information that is applied to a particular report column to be applied to the column headings for that column.
3. When the column heading for any item (whether a field or a literal) in a COLUMNS statement is a simple, one-line HTML literal, Spectrum Writer propagates that literal into any column heading lines above that line and also into the underscore line. The purpose of this is to keep all column headings lines aligned with each other.

4. When the column heading for any item (whether a field or a literal) in a COLUMNS statement consists solely of multiple lines of HTML literals, Spectrum Writer propagates the HTML literal for the bottom column heading line into the underscore line. (It does not propagate anything upward.) The purpose of this is to keep the column heading underscore line aligned with the other column heading lines.

KEEPRDW

When reading non–VSAM input files with variable length records, Spectrum Writer considers column 1 of the input record to be the first byte *after* the RDW (record descriptor word). This option tells Spectrum Writer that you want the RDW to be considered a part of the input record. When KEEPRDW is specified, the RDW is considered to be in column 1 of the input record. The first column after the RDW will be column 5. Specifying KEEPRDW in the OPTIONS statement makes it apply to *all* input files used in the run. You may also specify this keyword in individual FILE or INPUT statements. The use of this parm is discussed on page 352.

Note: VSAM files and DB2 tables do not have RDWs at the beginning of each record. This option is ignored for these kinds of files.

LEFTMARGIN(nnn/0)

Specifies a number of blank spaces to use as a left margin when printing the report. By default, there is no left margin.

MAINFRAME

Specifies that a mainframe output file is wanted (rather than a report). This parm:

- disables report titles and column headings
- disables Grand Totals
- sets the default inter-column spacing factor to zero bytes
- prevents the carriage control character from being written
- sets the default display formats to CHAR, DISPLAY, YYMMDD, and HHMMSS

The use of this parm is discussed in "Producing Files for Mainframe Programs" (page 280).

MAXINCLUDE(nnnnn)

Specifies the maximum number of records from the primary input file that should be *included* in the report. (That is, the maximum number of records that pass the INCLUDEIF statement conditions.) This is helpful while developing new reports that use very large input files. You can use this option to limit the number of records processed during test runs. You may need to use this option rather than the related MAXINPUT option, when the records required for your report are not the first records in the input file.

MAXINPUT(nnnnn)

Specifies the maximum number of records that should be *read* from the primary input file when producing the report. This option is helpful when you are developing a new report that uses a large input file. This allows you to read in only a few hundred records (for example) to get an idea of how your report will look. This will run much faster than a report that processes the whole file. (Also see the related MAXINCLUDE option.)

MAXINVSHOW(nnnnn/10)

Specifies the maximum number of invalid fields that should be displayed in hex format in the control listing. The default is to display the first 10 invalid fields that are encountered. Specify MAXINVSHOW(0) if you don't want to see any details about invalid fields.

MAXNORMDUMP(nnnnn/10)

Specifies the maximum number of times a record dump should be printed for normalization errors. (Normalization errors are discussed in "Normalization Errors" on page 248.) The default is to print a record dump for the first 10 normalization errors in a run. Specify MAXNORMDUMP(0) if you don't want any such dumps in your listing.

MAXPAGES(nnnnn)

Specifies the maximum number of report *pages* that should be printed. This is helpful while developing new reports. It ensures that whole boxes of paper won't accidentally be printed if there are serious errors in the control statements. (See also the related MAXPRINT, NOCHECK and NOMAXMSG options.)

MAXPRINT(nnnnn)

Specifies the maximum number of report *lines* that should be printed (including titles, column headings, footnotes, etc.) This is helpful while developing new reports. It ensures that whole boxes of paper won't accidentally be printed if there are serious errors in the control statements. (See also the related MAXPAGES, NOCHECK and NOMAXMSG options.)

MEMTYPE('type'/'SPECTWTR')

VSE only. Specifies the default member type to use when reading members from the Spectrum Writer Copy Library. If this parm is not specified, the default member type is SPECTWTR. The default member type is used for COPY statements that do not explicitly specify a member type.

> **Note:** This default member type applies only to copies performed outside the scope of ASM and COBOL statements. Different default member types are used within the scope of those statements. (See the ASMLIB and COBLIB options.)

```
Example: OPTIONS: MEMTYPE('SW')
```

The above statement tells Spectrum Writer to look for members whose member type is SW, when copying members from the copy library.

MISSOFFSET

Specifies that fields having OFFSET parm errors (page 353) should be treated as if they were "missing." (Missing fields are assigned zeros for numeric, date and time fields, blanks for character fields, and OFF for bit fields). Use this option if you want to suppress "offset error indicators" (***F***) in your report.

MULTICOLHDG

By default, when more than one COLUMNS statement is used Spectrum Writer does not automatically produce column headings. (The TITLE statement is often used in such situations to manually create column headings — see page 153.) If you want Spectrum Writer to automatically provide column headings in a report that has multiple COLUMNS statements, specify:

```
OPTIONS: MULTICOLHDG
```

Spectrum Writer will use the column headings that would have been generated if the request contained only the first COLUMNS statement. For many multi–line reports, this provides an easy way to produce useful column headings. Of course, you can specify override column headings in your first COLUMNS statement, as usual. Those override column headings will then be used in the report. Any default or explicit column headings in the second and later COLUMNS statements will be ignored.

NOBLOCKSIZ

OS/390 only. Prevents Spectrum Writer from setting a default blocksize for the SWOUTPUT DD. Otherwise, when no blocksize (or a zero blocksize) is specified in the JCL, Spectrum Writer sets the block size equal to the record size (resulting in single blocked output).

NOCC

Specifies that no "carriage control" characters should be written. Normal report lines are prefixed with a carriage control character, which contains a printer spacing command. When writing to an output file, rather than to a printer, the carriage control character is not normally wanted.

> **Note:** Specifying a PC file formatting option (or the MAINFRAME option) also suppresses the carriage control character.

NOCHECK

Only relevant if the MAXPRINT or MAXPAGES option is used. Tells Spectrum Writer that the NOCHECK option is in effect for your shop's sort program. This means Spectrum Writer can safely quit the sort early when the MAXINPUT or MAXINCLUDE limit has been reached. Otherwise, in order to prevent a SORT ABEND, Spectrum Writer must continue to process the remainder of the sort file (flushing the records), which requires more processing time.

NOCLEARIO

For some input files, Spectrum Writer clears (sets to hex zeros) the I/O area before performing each read. The NOCLEARIO option specifies that such clearing should not be performed for *all* of the files used in the run. When such clearing is not necessary, suppressing it may improve performance. (You can use a CLEAR(NO) parm in an INPUT or READ statement to suppress such clearing for *individual* files.)

NOCOLHDGS

Specifies that Spectrum Writer should not create column headings for the report or PC file. Spectrum Writer also defaults to the NOCOLHDG option for all reports that use more than one COLUMNS statement.

NOGRANDSPACES

Suppresses all spacing before the Grand Totals. Normally, the Grand Totals are separated from the body of the report by two blank lines, or are printed on a separate page. When creating output files, you may want to specify this option to prevent any blank records from being written to your output file before the Grand Total record.

NOGRANDTOTAL

Specifies that Grand Totals are not wanted for this report.

NOMAXMSG

Only relevant if the MAXPRINT or MAXPAGES option is used. Tells Spectrum Writer not to print a message in your report when the maximum limit has been reached.

NOOVERPRINT

Specifies that no lines should be "over–printed" in the report. (They will be single-spaced instead.) An example of an over–printed line is the line of underscores under the column headings. Use this option when the printer being used to print the report does not have over–print capability.

NOTEMPTYCC(nn)

Sometimes it is important to know that a report is not "empty" (that is, that one or more records did pass the inclusion tests.) For example, you may have an "exception report" that checks for error situations that should not normally occur. Normally the report is empty. When an exceptional situation *is* found, you want the completion code to call attention to that fact. This option lets you specify a special completion code for Spectrum Writer to use for runs that are not empty.

```
Example: OPTION: NOTEMPTYCC(16)
```

The above statement tells Spectrum Writer to set the step completion code to 16 if any records are included in the run. If no records are included, the standard completion code will be used (normally 0, unless an error was encountered).

Note: This option can not be used to *lower* Spectrum Writer's standard completion code (which may be 4 or 8 if warning or error messages were issued).

Note: See also the related EMPTYCC option.

NOSORTSIZE

Tells Spectrum Writer not to pass any MAINSIZE parm to the system Sort program. (The MAINSIZE parm tells Sort how much memory to use in performing the sort.) You may want to use this parm to ensure that your shop's default parm remains in effect.

NOSYSINLIMIT

By default Spectrum Writer suspects a loop when more than 50,000 control cards have been processed. (Looping can be caused by copying a member that copies itself recursively.) When this occurs, a message is printed and the run is terminated. To disable this limit on the number of control cards accepted, specify this option.

NOTITLES

Specifies that no titles are wanted for the report. By default, if no TITLE statements are specified for a report, Spectrum Writer prints a default title line. This option prevents that default title line from printing. When NOTITLES is specified, no page break processing is performed— the report will print over paper perforations, etc. This option is useful when the report output will be routed to a *dataset* for further processing, rather than to a printer.

Note: This option also suppresses the printing of all column headings and FOOTNOTE lines.

NOUNDERSCORES

Specifies that the column headings in the report should not be underscored. This is often desirable for reports that will be viewed online, since the underscore line uses up an additional line on the screen.

ONIOERROR(DEFAULT/ERROR/STOP)

Specifies how I/O errors on input files should be treated. When specified in an OPTIONS statement, this parm applies to all input files (except those with an explicit ONIOERROR parm directly in their INPUT or READ statement).

This parm affects mostly auxiliary input files, since I/O errors on the primary input file terminate the report.

When an I/O error occurs on an auxiliary input file, Spectrum Writer simply prints a warning message in the control listing and continues the run without reading from that file again. (All records from the file are treated as "missing".)

Sometimes, however, the data from the file in error may be so important to the report that it is pointless — or worse, misleading — to continue the run without it. Or, you may want to continue the run, but change the job completion code to indicate that a problem exists with the report. Use the ONIOERROR parm for such situations.

Specify ERROR in this parm to change the control listing message from a warning to an error (which also sets the job completion code to 8).

Specify STOP in this parm to have Spectrum Writer halt the run immediately when an I/O error occurs on a file. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: The OPTIONS statement containing this parm must appear early in the control statements — before any non-OPTIONS statements.

Note: "Missing" records from auxiliary input files are not considered I/O errors.

```
Example: OPTION: ONIOERROR(ERROR)
```

The above example specifies that an I/O error on any input file should be considered an "error" condition.

ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)

Specifies how normalization errors in input files should be treated. (For examples of normalization errors, see page 248.) When specified in an OPTIONS statement, this parm applies to all input files (except those with an explicit ONNORMERROR parm directly in their INPUT or READ statement).

By default, normalization error messages are treated as informational messages only. When a normalization error occurs, Spectrum Writer processes the physical record, and then skips the normalization in question for that record. If you want normalization errors to be treated as more serious errors, use the ONNORMERROR parm.

Specify WARNING in this parm to change the control listing message from "informational" to a "warning" (which also sets the job completion code to 4).

Specify ERROR in this parm to change the control listing message from "informational" to an "error" (which also sets the job completion code to 8).

Specify STOP in this parm to have Spectrum Writer halt the run immediately when a normalization error occurs. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: The OPTIONS statement containing this parm must appear early in the control statements — before any non-OPTIONS statements.

Note: When normalization errors occur, Spectrum Writer also prints a dump of the record in error. You can use the MAXNORMDUMP option (in an OPTIONS statement) to control how many such dumps appear in your control listing.

```
Example: OPTION: ONNORMERROR(ERROR)
```

The above example specifies that all normalization errors should be treated as "errors."

OUTATTR(type [,'dlbl/tlbl'] [,SYSnnn] [,recsize] [,blksize])

VSE only. This parm describes the attributes to use for Spectrum Writer's output. The section beginning on page 431 discusses the use of this parm.

SUBPARMS ALLOWED IN THE OUTATTR PARM	
SUBPARM	**MEANING**
type	This parm is required. It tells Spectrum Writer what kind of device to write the output to. It must be one of the following values: **PRINTER/PRT** a printer–type device (including POWER print queues) **DASD** a SAM file on a DASD device (disk). (Use DASD type even if your SAM files are managed by VSAM.) **TAPE** a SAM file on a magnetic tape **VSAM** an ESDS VSAM file
'dlbl/tlbl'	This parm is required unless writing to a printer device. It tells Spectrum Writer what DLBL or TLBL is used in the JCL for the output file. The 1– to 7–byte name within apostrophes (or quotation marks) must be the same as the filename in a DLBL or TLBL statement in your JCL.
SYSnnn	This parm is required for PRINTER and TAPE output. It is treated as a comment for other output types. It identifies the logical unit to write the output to. The value specified here must also be "assigned" in your JCL.
recsize	This parm is optional. It specifies the length of the output records to be written. If omitted, a record size of 133 is assumed. Note that for report output, the first byte in each record is used as a "carriage control character." Therefore, by default 132 bytes are available for the report data itself. For PC file and mainframe file output (or when using the NOCC option) no control character is written, and the entire length of the record is available for data.
blksize	This parm is optional. It specifies the block size to use when writing a DASD or TAPE output file. (This parm is not allowed for PRINTER or VSAM output types.) This value must be a multiple of the recsize value. If omitted, single record blocking is used. That is, the default is to make the block size the same as the record size.

Notice that the OUTATTR parm does not have a record format parm (F/V), which the similar ATTR parm in the FILE statement has. Spectrum Writer output is always written as fixed length records (and fixed length blocks, if blocked).

OUTDDN(ddname)

OS/390 only. Specifies the ddname (in the JCL) to be used for the output records written by Spectrum Writer. Use this parm if you do not want your output(s) to be written to the standard output ddnames (SWOUTPUT, SWOUT002, SWOUT003, etc.)

Note: The OPTIONS statement containing this parm must appear early in the report request to which is applies — before any non-OPTIONS statements. In a multi-output run, this parm applies to the output currently being defined.

OUTLRECL(nnnnn)

OS/390 only. Specifies the LRECL to be used for the output records written by Spectrum Writer. This parm is mainly intended for use when writing to a VSAM output file. The LRECL chosen by Spectrum Writer for its output records is determined in this way.

For VSAM output files, the LRECL used is:

1. the OUTLRECL parm value (if it is valid for the VSAM file's definition), or
2. 133 (if it is valid for the VSAM file's definition), or
3. the maximum LRECL value defined for the VSAM file

For QSAM output, the LRECL used is:

1. the LRECL specified in the JCL, if any, or
2. the LRECL specified in the file's label, when writing to an existing dataset, or
3. the OUTLRECL parm value, if any, or
4. 133

Note: The OPTIONS statement containing this parm must appear early in the report request to which is applies — before any non-OPTIONS statements. In a multi-output run, this parm applies to the output currently being defined.

OUTTYPE(SEQ/VSAM)

OS/390 only. Specifies the type of I/O to be used by Spectrum Writer when writing output records. When VSAM is specified, the dataset named in the output DD (ddname SWOUTPUT by default) must be an existing, ESDS VSAM dataset. If Spectrum Writer's output will be written to a SYSOUT DD or to a non–VSAM dataset, specify SEQ (which is also the default).

Note: The OPTIONS statement containing this parm must appear early in the report request to which is applies — before any non-OPTIONS statements. In a multi-output run, this parm applies to the output currently being defined.

PAGELENGTH(nnn/60)

Specifies how many lines should be printed per page. The first title line of your report is considered line 1. The default number of lines to print per page is 60. Use this option to change the number of blank lines that appear at the bottom of each page.

PC/OUTPUT/ACCESS/COREL/CSV/DBASE3/DBASE4/EXCEL/FOXPRO/HARVARD/LOTUS/MS–WORKS/PARADOX/QUATTRO/RBASE

Specifies that a particular kind of output file is wanted (rather than a report). The use of these options is discussed in the lesson that begins on page 88.

POSTSCRIPT('text')

Specifies a literal text that should be printed once at the end of the report (or output file). You may have as many POSTSCRIPT options as you like. They will print in the order they are specified in. The HTML option, if specified, also causes certain lines to print at the end of a report. The POSTSCRIPT lines print just before the lines produced by the HTML option.

PRESCRIPT('text')

Specifies a literal text that should be printed once, before the beginning of the report (or output file). You may have as many PRESCRIPT options as you like. They will print in the order they are specified in. The HTML option, if specified, also causes certain lines to print at the beginning of a report. The PRESCRIPT lines print just after the lines produced by the HTML option.

PRTSETUP('text')

Specifies a string of characters to be sent to the printer once before the report is printed. This string can contain any setup information that is valid for your printer. One use of this option is to request a "condensed font" with your laser printer. This may allow you to print reports wider than the standard 132 characters.

> **Tip:** If the text you specify doesn't seem to work, try adding an extra space at the beginning of your text. The printer may be treating the first character as a carriage control character and ignoring it.

```
Example: OPTION: PRTSETUP('+$$$DJDE$ JDE=40,FORMAT=L66200,DATA=(0,200),END;')
```

The above statement causes the specified setup string to be sent to the printer once, before the report starts printing. Of course, the actual contents of the setup string will be different for each shop.

PRTSHEET('text')

Specifies a string of characters that can be sent to a laser printer to force it to skip to a new sheet of paper. When the NEWSHEET or NEWSHEET1 space options are used at control breaks, this option *must* be specified. At the appropriate time, Spectrum Writer will send this string to the printer to cause it to skip to a new sheet.

> **Note:** If NEWSHEET or NEWSHEET1 is specified for any control break, the PRTSHEET text will *also* be sent to the printer at the very beginning of the report. This is to ensure that the first page of the report begins on a new sheet of paper.

> **Tip:** If the text you specify doesn't seem to work, try adding an extra space at the beginning of your text. The printer may be treating the first character as a carriage control character and ignoring it.

```
Example: OPTION: PRTSHEET('+$$$DJDE$ SIDE=NUFRONT,END;')
```

The above statement causes the specified string to be sent to the printer each time Spectrum Writer needs to skip to a new sheet of paper. Of course, the actual contents of the string will be different for each shop.

QCHAR('char'/'"')

Specifies the "quotation character" to use in conjunction with the QCHAR, Q–MM–DD–YY, Q–HH–MM–SS and similar display formats. The default is to use a regular (double) quotation mark as the enclosure character for those display formats. If you need to enclose such data in some other character, use this option.

```
Example: OPTIONS: QCHAR("'")
```

The above statement specifies that the apostrophe character should be used to enclose data that is formatted with a quoted-type display format. For example, a date formatted with the Q–MM–DD–YY display format would now look like
'12/31/96' rather than "12/31/96".

SCALEPIC

This option enables the automatic scaling symbols in PICTURE display formats. If not specified, the symbols "@" and "?" within a PICTURE are simply literal characters. When SCALEPIC is specified (prior to the PICTURE in question), those two symbols are treated as automatic scaling symbols.

SINGLE/DOUBLE/TRIPLE

Specifies how the report should be spaced. The default is to single space the report.

> **Note:** This option determines how many (if any) blank lines are left between the detail report line(s) for each input record. If multiple COLUMNS statements are used, the detail report lines for a single input record are always single spaced. You can use empty COLUMNS statements if you want to print blank lines *within* the detail report lines for a single input record.

SKIPBLANKDET

This option causes Spectrum Writer to skip (not write out) any detail report line (or PC file record) that is all blank. For the purposes of this option, "detail lines" means: the detail lines printed for each input record; the total lines printed at control breaks (if any); and the Grand Total lines (if any). Titles, column headings and break headings are not affected by this option. Use of this option is discussed on page 249.

> **Note:** Only the first 256 bytes of each line are examined when checking for blank detail lines.

SKIPZERODET

This option causes Spectrum Writer to skip (not write out) any detail report line (or PC file record) that contains only "zero values". The following are considered "zero" values for this purpose:

- blanks (for character fields)
- 0's (including decimal points such as 0.00)
- 00/00/00 (zero dates)
- 00:00:00 (zero times)

For the purposes of this option, "detail lines" means: the detail lines printed for each input record; the total lines printed at control breaks (if any); and the Grand Total lines (if any). Titles, column headings and break headings are not affected by this option. Use of this option is discussed on page 249.

Note: Only the first 256 bytes of each line are examined when checking for zero detail lines.

SORTDD('prefix')

This parm tells Spectrum Writer the 4-byte DDNAME prefix that the Sort program should use when opening sort work files (for the report currently being defined). (The complete DDNAMEs are then formed by appending "WK01", "WK02", etc. to this prefix. Under VSE, the complete DLBLs are formed by appending "WK1", "WK2", etc. to this prefix.) Unless your shop's system sort allocates its work files dynamically, your JCL should include one or more DD (or DLBL) statements with these names. If not specified, Spectrum Writer assumes the following prefixes for the sort work files DD (or DLBL) statements:

- SORT, for the first (or only) output in the run
- SRT2, for the second output in the run
- SRT3, for the third output in the run

  . . .

- SR10, for the tenth output in the run, and so on

Note: In a multi-output run, this parm applies only to the Sort for the output currently being defined in the control statements.

```
Example: OPTIONS: SORTDD('TEMP')
```

The above statement specifies that Spectrum Writer should tell the Sort program to use DDNAMEs TEMPWK01, TEMPWK02, TEMPWK03, etc., for any sort work files that it needs to sort the current report.

SORTNAME('program'/'SORT')

This parm specifies the name of your shop's sort program. The default name of SORT is used in almost all shops. However, some shops have multiple sort programs available and you may want to use an alternate sort program.

```
Example: OPTIONS: SORTNAME('SORT2')
```

The above statement specifies that Spectrum Writer should use the program named SORT2 to perform any necessary sorts.

SORTOPT('text')

Use this parm to pass any special option parm to the system Sort program. If specified, Spectrum Writer will append a comma and your text to the end of the Option control statement that it passes to the system Sort program.

Note: In a multi-output run, this parm applies only to the Sort for the output currently being defined in the control statements.

```
Example: OPTIONS: SORTOPT('FILSZ=E5000')
```

The above statement tells Spectrum Writer to add the parm FILSZ=E5000 to the Options statement that it builds and passes to the system Sort program. (The parm in this example tells the Sort program to expect approximately 5000 records to be sorted.)

SORTSIZE(nnnn/256)

This parm specifies the MAINSIZE parameter (in kilobytes) that should be passed to your shop's sort program when it is called. This parm tells the sort program how much memory it should use while performing the sort. If you omit this parm, Spectrum Writer passes your sort program a size parm of 256K. You may want to specify a smaller value in order to run in a smaller region or partition. Or, in some cases you may get better performance by specifying a larger value than the default. The maximum value allowed by Spectrum Writer is 8191 (which means 8191K, or 8M). (Your sort program may have a smaller maximum limit. You may also be limited by the size of the region or partition you run in.) Under VSE, you may also need to modify the SIZE parm in your EXEC JCL statement (to ensure that your partition has this much memory available for the sort program).

> **Note:** In a multi-output run, this parm applies only to the Sort for the output currently being defined in the control statements.

```
Example: OPTIONS: SORTSIZE(64)
```

The above statement tells Spectrum Writer to pass the sort program a size parm of 64K.

SORTWORKNUM(n/0)

VSE only. This parm specifies how many, if any, external work files can be used by the system Sort program. By default, zero sort work files are assumed. That is, the sort program will attempt to perform the entire sort in memory. For larger runs, you may need to provide DLBL (and EXTENT) statements for "sort work" files in your JCL. The DLBLs should generally be named SORTWK1, SORTWK2, etc. (See page 434.) Use this parm to tell Spectrum Writer how many of these sort work files are available for it to use. You may specify a number from 0 to 9.

```
Example: OPTIONS: SORTWORKNUM(3)
```

The above statement specifies that 3 sort work file DLBL statements are provided in the JCL for the sort program to use.

SPLITDETAIL

Specifies that it is OK to split the detail lines for a single input record across pages in the report. If you do not specify this option, Spectrum Writer will skip to a new page whenever the current page does not have enough room to show all of the detail lines for an input record. (Using multiple COLUMNS statements results in multiple detail lines for a single input record.) Normally you will probably not use SPLITDETAIL, since it is easier to view related data when it is all on a single page. But that does use extra paper. And, it may be impractical if you are listing 30 or 40 items from each input record, since virtually every record would end up requiring a new page. In these cases, you may specify SPLITDETAIL to allow Spectrum Writer to fill up each page before going on to the next page of the report.

STCKADJ(nn)

Specifies how many hours should be added to fields stored in the STCKDATE and STCKTIME data types in order to obtain the local date and time. IBM's STCK machine instruction stores its date–time stamps in GMT. Spectrum Writer normally converts STCKDATE and STCKTIME values from GMT to local time. The number of hours Spectrum Writer adds or subtracts to the GMT time is determined by your installation's system parms. If you do not want this automatic conversion performed, use the STCKADJ option. This option specifies the number of hours that should be added to the STCK value. (The number of hours may be a positive or negative value.)

For example, to suppress conversion altogether and leave STCKDATE and STCKTIME values in GMT, you would specify the following:

```
OPTIONS: STCKADJ(0)
```

STOPWHEN(conditional-expression)

Tells Spectrum Writer that it can stop reading the primary input file when a certain condition is met. Use this parm to reduce I/O processing — and run time — if you know that your report will not need records after a certain point in your input file (and if that point can be specified in a conditional expression).

> **Note:** When specified in an OPTIONS statement, this parm affects only the report currently being defined. Any other reports produced in the same run will not be affected by this STOPWHEN parm. (If you want the STOPWHEN parm to apply to *all* reports in a run, then put it in the INPUT or FILE statement, rather than in the OPTIONS statement.)

Example: `OPTION: STOPWHEN(EMPL-NUM > '039')`

The above statement tells Spectrum Writer that, for the purposes of the current report, it can stop reading the primary input file when it encounters a record whose EMPL-NUM field is greater than '039'. When Spectrum Writer reads a record whose EMPL-NUM is greater than '039', it will ignore that record and act as if it has hit EOF on that file. No more records will be read from the file. The report will be produced based on the records read up to that point.

SUBLIB('library.sublibrary')

VSE only. Specifies the name of the VSE sublibrary to use as the Spectrum Writer Copy Library.

Example: `OPTIONS: SUBLIB('LIB.SPECTWTR')`

The above statement causes the Librarian dataset named LIB.SPECTWTR to be used as the Spectrum Writer Copy Library.

SUMMARY

Specifies that a summary report is wanted. The report will contain no detail lines. Only lines associated with control breaks (and the Grand Total) will print. This option has the same effect as specifying DETAIL(0). However, this option also changes the default break spacing for the lowest level control break from two blank lines to zero blank lines. This prevents the report's summary lines from appearing to be triple spaced.

TIMEDELIM('char'/':')

This option lets you specify any character you choose to be used as the delimiter when formatting times. This delimiter will be used with all time display formats that include a delimiter. The default time delimiter is a colon (:). For example, to format all times using dots rather than colons, you would specify:

```
OPTIONS: TIMEDELIM('.')
```

This would cause the HH–MM–SS display format to appear as "12.00.00" (for example).

> **Note:** Use of this parm does *not* affect the way Spectrum Writer recognizes time literals in the control statements. Time literals must always be written using colons as delimiters.

TITLEONCE

Causes the titles (and any column headings) to print only once at the beginning of the report. Also, any footnotes will print only once at the end of the report. There will be no page breaks within the report. In other words, the entire report is treated as one long page. This option is sometimes useful when creating Web reports (which are viewed on PC screens rather than on pages of paper).

ZERODIVBYZERO

Tells Spectrum Writer to assign a value of zero to a COMPUTE field when a division by zero error occurs. This suppresses the `***Z***` error indicator in reports.

ZEROINVDATA

Tells Spectrum Writer to assign a value of zero to fields that contain invalid data in the input record. This suppresses the `***I***` error indicator in reports.

ZEROOVERFLOW

Tells Spectrum Writer to assign a value of zero to a COMPUTE field when an overflow error occurs. This suppresses the `***V***` error indicator in reports.

READ Statement

PURPOSE

Specifies an **auxiliary input file** to be used in producing a report or PC file. Each run must have one (and only one) *primary input file*, which is specified with an INPUT statement. If a report or PC file requires information from additional files, these files must be specified with READ statements. You may have as many READ statements in a run as you like. The READ statements must appear after the INPUT statement.

An auxiliary input file is useful if the primary input file does not contain all of the information needed for a run. After a READ statement has been processed by Spectrum Writer, all of the fields defined for that auxiliary file become available for use in producing the report (or PC file). These fields can be used in exactly the same way as fields from the primary input file. They can be used: as a column of data in the report or PC file; in report titles; as a sort field; as a control break field; as part of a conditional expression; as operands in computational expressions; even as key fields used to read records from other auxiliary input files.

The READ statement is one of the most powerful statements in Spectrum Writer.

FEATURES

Use the READ statement to:

- specify the **name** of an auxiliary input file
- specify a field containing the **read key** to be used when reading from VSAM files
- specify a **WHERE clause** to be used when reading from a DB2 table or view
- automatically **copy** additional control statements from the Spectrum Writer Copy Library (typically used to copy the FILE and FIELD statements that define the auxiliary input file)
- specify a **record name** to be associated with records from this auxiliary input file
- **override** certain aspects of the auxiliary input file definition

LEARNING MORE

The complete syntax of the READ statement is shown on the following pages. In addition, the following parts of the manual relate to the READ statement:

- a lesson on using the READ statement in reports begins on page 76
- a lesson on using the READ statement in PC files begins on page 116

- advanced techniques involving the READ statement are discussed beginning on page 224
- the use of the READ statement with DB2 tables is discussed beginning on page 400
- suggestions on writing READ statements for maximum run–time efficiency are given in Appendix G, "Speed-Up Tips" (page 652)
- reading a file that is processed by a user I/O Exit is discussed in Appendix I, "I/O Exits" (page 673)

SYNTAX

READ STATEMENT SYNTAX

```
READ:   filename
     [  ATTR(VSAM/EXIT, 'dlbl', recsize)              (VSE only)   ]
     [  BUFND(nn)                                     (VSAM only)  ]
     [  BUFNI(nn)                                     (VSAM only)  ]
     [  CLEAR(SPACES/ZEROS/NO)                                     ]
     [  COPY(YES/NO)                                               ]
     [  DB2NAME('[qualifier.]name')                   (DB2 only)   ]
     [  DDNAME(ddname)                                (OS/390 only)]
     [  EXITPARM('text')                                           ]
     [  GENERIC                                                    ]
     [  IOEXIT('program' [,'parm'] [TRACE])                        ]
     [  KGE                                                        ]
     [  LIST(YES/NO)                                               ]
     [  LRECL(nnnnn)                                  (OS/390 only)]
     [  MULTI                                                      ]
     [  NONORMALIZE                                                ]
     [  NORMALIZE(fieldname, num-expr [, ...] ) ...                ]
     [  NORMWHEN(conditional-expression)                           ]
     [  ONIOERROR(DEFAULT/ERROR/STOP)                              ]
     [  ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)                    ]
     [  ORDERBY(fieldname [ASC/DESC] [, ...] )        (DB2 only)   ]
     [  READKEY(fieldname)                                         ]
     [  RECNAME(name/filename)                                     ]
     [  SHOWFLDS(YES/NO)                                           ]
     [  TYPE(VSAM/DB2/EXIT)                           (OS/390 only)]
     [  WHERE(search-condition)                       (DB2 only)   ]
```

(continued on next page)

READ STATEMENT SYNTAX (CONTINUED)

Standard Spelling	Alternate Spellings
DDNAME	DDN
DEFAULT	DEF
ERROR	ERR
EXITPARM	PARM
GENERIC	GEN
NO	N
NONORMALIZE	NONORM
NORMALIZE	NORM
NORMWHEN	NORMALIZEWHEN
ONIOERROR	ONIOERR
ONNORMERROR	ONNORMERR
READKEY	KEY
RECNAME	NAME
TYPE	TYP
WARNING	WARN
YES	Y

The filename parm is *required*. In addition, either a WHERE parm (for DB2 files) or a READKEY parm (for other files) is also required. The syntax of the READ statement is otherwise very similar to that of the INPUT statement.

filename

Identifies the auxiliary input file to use. One or more records will be read from this file each time a new record is read from the primary input file. Files named in READ statements must be either keyed files or DB2 tables.

The filename specified in this parm must have been defined in an earlier FILE statement. However, that FILE statement may be in a copy library member that is automatically copied into the report at the time the READ statement is processed. This process is explained beginning on page 360.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)
```

The above statement specifies that the file named EMPL–FILE will be an auxiliary input file for the run.

ATTR(VSAM/EXIT, 'dlbl', recsize)

VSE only. Specifies override file attributes to use for this file (for the current run only). Files named in VSE READ statements must be keyed VSAM files or EXIT files. For examples of using this parm, see page 331.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)
                ATTR(VSAM, 'EMPLFIL', 80)
```

The statement above names EMPL–FILE as an auxiliary input file for the run. Regardless of how EMPL–FILE was defined in an earlier FILE statement, for the current run it is treated as a VSAM file, with EMPLFIL as the DLBL name, with 80–byte (or smaller) records.

BUFND(nn)

VSAM files only. Specifies the number of "data buffers" that the VSAM access method should maintain when processing this input file. When this parm is not specified for a VSAM file, Spectrum Writer chooses a default number of data buffers to maintain.

> **Note:** According to the VSAM manual, increasing the number of data buffers by one or two (from VSAM's default of 2) may improve performance for random reads. After that, more benefit is obtained by increasing the number of *index* buffers instead (use the BUFNI parm for that). You may wish to experiment with this parm if you have long–running, VSAM–intensive jobs.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  BUFND(3)
```

The above statement specifies that VSAM should allocate buffer space for three data control intervals when processing the EMPL–FILE.

BUFNI(nn)

VSAM files only. Specifies the number of "index buffers" that the VSAM access method should maintain when processing this input file. When this parm is not specified for a VSAM file, Spectrum Writer chooses a default number of index buffers to maintain.

> **Note:** According to the VSAM manual, increasing the number of index buffers (from VSAM's default of 1) should improve performance for random reads up to a certain point. At some point, excessive paging may cancel any benefit. Optimal performance is sometimes achieved by having one index buffer for each level of the file's index. You may wish to experiment with this parm if you have long–running, VSAM–intensive jobs.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  BUFND(3)  BUFNI(6)
```

The above statement specifies that VSAM should allocate buffers for three data control intervals and six index control intervals when processing the EMPL–FILE.

CLEAR(SPACES/ZEROS/NO)

When processing certain types of input files, Spectrum Writer clears the entire I/O area to blanks before each read. This is to ensure that when a short record is read, it is not followed by leftover data from a previous longer record. For certain record layouts such leftover data could cause misleading results. Specifying CLEAR(NO) suppresses this clearing, which may result in improved performance. You might want to specify CLEAR(NO) if you are certain that any leftover data in the I/O area will not adversely affect your run. Specifying CLEAR(ZEROS) causes Spectrum Writer to initialize the I/O area to hex zeros (rather than blanks) before each read.

> **Note:** You can also specify the CLEAR parm in the FILE statement to avoid having to put it in the READ statement each time. And, the NOCLEARIO parm in an OPTIONS statement can be used to prevent clearing of *all files* in a run.

```
Example: READ: PAYROLL-FILE  READKEY(EMPL-NUM)  CLEAR(NO)
```

The above statement names the PAYROLL–FILE as the input file for a run. Spectrum Writer will not clear its I/O area each time it reads a record from that file.

COPY(YES/NO)

Specifies whether control statements should be copied from the copy library before evaluating the file name. If the COPY parm is omitted and the file name has not been previously defined, the default is to attempt to perform a copy. Normally, the control statements that are copied will include the FILE and FIELD statements that describe the input file. This process is explained beginning on page 360.

If an attempt to copy records is unsuccessful (due to a missing copy library or missing member), that is *not* considered an error. Normal control statement processing continues, without any copy being performed.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  COPY(NO)
```

The above example specifies that no attempt should be made to copy records from the copy library.

DB2NAME('[qualifier.]name')

DB2 only. Specifies the name of the DB2 table or view that you wish to use as an auxiliary input for the run. For DB2 inputs, this parm is required (unless the filename was defined in an earlier FILE statement that included the DB2NAME parm.) The table name must be enclosed in quotation marks or apostrophes. Generally the table name will be qualified. If it is not explicitly qualified, DB2 will assume an implicit qualifier, which will be the DB2 Authorization ID of the job executing Spectrum Writer.

```
Example: READ: EMPLOYEE
               DB2NAME('DSN8230.EMP')
               WHERE(EMPNO = RESPEMP)
```

The above example specifies that the DB2 table named 'DSN8230.EMP' should be used as an auxiliary input "file" for the run. This input file has a Spectrum Writer file name of EMPLOYEE. That is, other Spectrum Writer control statements that refer to this input file will refer to EMPLOYEE (rather than to DSN8230.EMP).

DDNAME(ddname)

OS/390 only. Specifies an override DDNAME to use when reading the input file (for the current run only). If omitted, the DDNAME will be taken from the FILE statement that defined the file. A DDNAME parm *must be present* in either the FILE statement or the READ statement.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  DDNAME(TEMPDD)
```

The above example specifies that the TEMPDD DD statement in the JCL will be used to read the EMPL–FILE file, regardless of the DDNAME specified when the file was originally defined.

EXITPARM('text')

Most installations will not use exits, and will not need this parm. Specifies an override exit parm text. If this parm is omitted, the exit parm text (if any) will be taken from the FILE statement that defined the file. Exit parm text is passed to user data exit programs. Anytime a user data exit is called by Spectrum Writer for a field within this file, the text string specified in this parm will be passed to the exit. The use of this parm is discussed beginning on page 357.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  EXITPARM('12345')
```

The above example specifies that the text '12345' should be passed to user data exit programs involving this file, regardless of the EXITPARM specified when the file was originally defined.

GENERIC

VSAM and EXIT only. Specifies that the contents of the READKEY parm is a generic key rather than an entire key. That is, the length of the READKEY parm may be shorter than the key length in the VSAM file's definition. The first record in the file whose partial key matches the READKEY value will be read. If GENERIC is not specified, the READKEY value is assumed to be an entire key. The use of GENERIC keys is discussed in the section beginning on page 230.

```
Example: COMPUTE: SHORT-KEY = #SUBSTR(EMPL-NUM,1,2)
         READ:    EMPL-FILE  READKEY(SHORT-KEY)  GENERIC
```

The READ statement above uses a generic read key. The SHORT–KEY field is only two bytes long, while the defined key length for the EMPL–FILE file is three bytes. Thus, when performing the above read, the record read will be the first one where the first two bytes of its key equals the contents of SHORT–KEY.

IOEXIT('program' ['parm'] [,TRACE])

EXIT files only. Specifies override I/O Exit information for the input file. May also override the input file type (if it was something other than EXIT in the FILE statement). This parm provides the information necessary for Spectrum Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix I, "I/O Exits" (page 673).

> **OS/390 Note:** When this parm is present, a file type of EXIT is assumed and an explicit TYPE parm is not required.
>
> **VSE Note:** When this parm is present, an ATTR parm specifying a type of EXIT and a RECSIZE is required (in either this statement or the FILE statement).

'program' This parm is required. It specifies the name of the load module (OS/390) or phase (VSE) that Spectrum Writer will call in order to obtain records from the file.

'parm' This parm is optional. Each time the I/O Exit program is called by Spectrum Writer, the text specified in this parm will be passed to the exit program. Typically this text is used to provide the exit program with any special information it needs in order to process the file. This parm can be up to 255 bytes in length.

TRACE This parm is optional. When specified, Spectrum Writer prints trace information in the control listing before and after each call to the I/O Exit. This information can be useful when developing and debugging a new I/O Exit program. The TRACE parm is normally not used in production runs.

```
Example: READ: MASTER-FILE  READKEY(EMPL-NUM)  IOEXIT('MYEXIT')
```

The above example specifies that a program named MYEXIT should be called to read records from the auxiliary input file MASTER-FILE.

KGE

VSAM and EXIT only. Specifies that when reading this file, the first record should be returned whose key (or partial key, if GENERIC is also specified) is greater than or equal to the key (or partial key) in the READKEY parm. If KGE is not specified, only records that exactly equal

the READKEY value (or partial value) will be read. The use of the KGE parm is discussed in the section beginning on page 230.

> **Note:** The KGE parm may not be specified if the MULTI parm is also specified. Such a combination would result in reading every record in the file whose key was greater than or equal to the READKEY parm.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  KGE
```

When performing the above READ statement, a record is first sought whose key exactly matches the EMPL–NUM value. If none is found, the first record whose key is greater than the EMPL–NUM field will be read instead.

LIST(YES/NO)

Applies only if the COPY function is performed. The LIST parm specifies whether the copied control statements should be listed in the control listing. If no LIST parm is present, the default is to *not list* the copied statements.

> **Note:** If an error is detected in any of the copied control statements, that statement *will* be listed, along with the error message, regardless of the value of this parm.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  LIST(YES)
```

The above example specifies that the records copied from the copy library should be listed in the control listing.

LRECL(nnnnn)

OS/390 only. Specifies an override record length for the input file. This is the length of the largest record that might be found in the file. If this parm is omitted, the LRECL value (if any) from the FILE statement is used. If no LRECL parm is found in either the FILE or the READ statement, a default LRECL of 1000 is assumed.

> **Note:** Spectrum Writer uses this value only to determine the size of the I/O area that it allocates for use with the input file. Therefore it is not required that this value match the file's actual LRECL parm in the JCL or in the dataset's label information. In fact, if you suspect that a file's record size may grow in the future, you may want to specify a larger LRECL parm with some "growth" room in it. On the other hand, specifying an excessively large LRECL may result in higher CPU usage in some circumstances.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  LRECL(4000)
```

The above example specifies that a record as large as 4000 bytes long may be encountered in the EMPL–FILE file.

MULTI

For VSAM and EXIT files, specifies that when reading from this file, *all records* whose key (or partial key, if GENERIC is specified) matches the READKEY value should be read. If MULTI is not specified, only the first record whose key (or partial key) matches the READKEY value will be read.

For DB2 tables, specifies that when reading from this table, *all* records (rows) which pass the WHERE parm condition should be read. If MULTI is not specified, only the first record which passes the WHERE parm condition will be read.

The use of the MULTI parm is discussed in the section beginning on page 232.

Note: The MULTI parm may not be specified if the KGE parm is also specified. Such a combination would result in reading every record in the file whose key was greater than or equal to the READKEY parm.

```
Example: COMPUTE: SHORT-KEY = #SUBSTR(EMPL-NUM,1,2)
         READ:    EMPL-FILE  READKEY(SHORT-KEY)  GENERIC  MULTI
```

The READ statement above will read multiple records using a generic read key. The SHORT–KEY field is only two bytes long, while the defined key length for the EMPL–FILE file is three bytes. Thus, when performing the above read, all records will be read where the first two bytes of their key equals the contents of SHORT–KEY.

NONORMALIZE

Specifies that the auxiliary input file should not be normalized. Any NORMALIZE parms specified in the FILE statement (or in this READ statement) will be ignored.

```
Example: READ:  EMPL-FILE READKEY(EMPL-NUM) NONORMALIZE
```

The above example specifies that the EMPL-FILE should not be normalized for the current run, even though there may be a NORMALIZE parm in its FILE statement.

NORMALIZE(fieldname, num-expr [, ...]) ...

Specifies that the auxiliary input file should be normalized. (See "Using Normalization to Process Arrays" on page 237 for an explanation of file normalization.) The fieldname must be the name of a field that defines the entire first occurrence of the array that is to be normalized. The numeric expression specifies how many occurrences the array contains. (This numeric expression is evaluated individually for each input record to determine the number of occurrences in that record.)

Note: When normalizing an auxiliary input file, you should also specify the MULTI parm. Otherwise, only the first normalized record (which is the same as the physical record) will be used for a given read key.

The NORMALIZE parm may contain any number of fieldname-numeric-expression pairs. Each pair identifies one array to be normalized. When multiple arrays are specified within a single NORMALIZE parm, those arrays are normalized *in parallel* (see "Normalizing Multiple, Non-Nested Arrays" on page 245).

In addition, you may have any number of NORMALIZE parms in the READ statement. When multiple NORMALIZE parms are present, they represent *nested* arrays (see "Normalizing Nested Arrays" on page 244). The last NORMALIZE parm specifies the most deeply nested array, and is normalized first.Then the array specified in the next-to-last NORMALIZE parm is normalized, and so on.

Note: If only some of the records from the file should be normalized, use a NORMWHEN parm before the NORMALIZE parm.

```
Example: READ:  EMPL-FILE  READKEY(EMPL-NUM)  NORMALIZE(SALES-QTR1, 4)  MULTI
```

The above example specifies that the records read from the EMPL-FILE should be normalized. The first occurrence of the array being normalized is defined by SALES-QTR1. There are four occurrences of the array in each record.

NORMWHEN(conditional-expression)

Specifies which records from the input file should be normalized. (This parm is discussed in more detail in "Normalizing only Certain Records" on page 247.) When the conditional expression is true for a record, then all subsequent NORMALIZE parms (up until the next NORMWHEN parm) will be processed. If the conditional expression is false, the subsequent NORMALIZE parms will not be processed for that input record. (Any NORMALIZE parms that are not proceeded by a NORMWHEN parm are processed for every input record.)

```
Example:  READ: BATCH-FILE  READKEY(EMPL-NUM)  MULTI
                NORMWHEN(RECORD-TYPE = 'HDR')
                    NORMALIZE(STATUS-ARRAY, 5)
                NORMWHEN(RECORD-TYPE = 'DET')
                    NORMALIZE(CUSTOMER-ARRAY, 8)
```

The above statements tell Spectrum Writer to normalize the STATUS-ARRAY only for those records where the RECORD-TYPE field contains 'HDR'. And the CUSTOMER-ARRAY will be normalized only for those records where the RECORD-TYPE field contains 'DET'. Records with any other value in the RECORD-TYPE field will not be normalized at all.

ONIOERROR(DEFAULT/ERROR/STOP)

Specifies how I/O errors on this file should be treated. By default, when an I/O error occurs on an auxiliary input file, Spectrum Writer prints a warning message in the control listing and then continues the run without reading from that file again. (All records from that file are treated as "missing".)

Specify ERROR in this parm to change the control listing message from a warning to an error (which also sets the job completion code to 8).

Specify STOP in this parm to have Spectrum Writer halt the run immediately when an I/O error occurs on the file. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: You can also specify a ONIOERROR parm in an OPTIONS statement, if you want it to apply to *all* of the input files used in the run.

Note: "Missing" records are not considered I/O errors.

```
Example: READ:  EMPL-FILE  READKEY(EMPL-NUM)  ONIOERROR(ERROR)
```

The above example specifies that an I/O error on the EMPL-FILE should be considered an "error" condition rather than just a "warning."

ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)

Specifies how normalization errors in this file should be treated. (For examples of normalization errors, see "Normalization Errors" on page 248.)

By default, normalization error messages are treated as informational messages only. When a normalization error occurs, Spectrum Writer processes the physical record, and then skips the normalization in question for that record. If you want normalization errors to be treated as more serious errors, use the ONNORMERROR parm.

Specify WARNING in this parm to change the control listing message from "informational" to a "warning" (which also sets the job completion code to 4).

Specify ERROR in this parm to change the control listing message from "informational" to an "error" (which also sets the job completion code to 8).

Specify STOP in this parm to have Spectrum Writer halt the run immediately when a normalization error occurs. Spectrum Writer will print a message and then issue a "user ABEND" to terminate the run immediately.

Note: You can also specify an ONNORMERROR parm in the OPTIONS statement, if you want it to apply to *all* of the normalized files used in a run.

Note: When normalization errors occur, Spectrum Writer also prints a dump of the record in error. You can use the MAXNORMDUMP option (in an OPTIONS statement) to control how many (if any) such dumps appear in your control listing.

```
Example: READ:  EMPL-FILE  READKEY(EMPL-NUM)
                NORMALIZE(SALES-QTR1,4)  ONNORMERROR(ERROR)
```

The above example specifies that any normalization error in the EMPL-FILE should be treated as an "error".

ORDERBY(fieldname [ASC/DESC] [, ...])

DB2 only. This parm is optional. It is possible that more than one row will pass the search condition in your WHERE parm. If the MULTI parm is also specified, all of these rows will be passed to Spectrum Writer, one by one. If MULTI is not specified, Spectrum Writer accepts only the *first* row passed to it from DB2. Use this parm to specify the order in which the selected row(s) should be passed to Spectrum Writer. The contents of this parm is one or more column name from the DB2 table, optionally separated with commas. You may also include the DB2 keywords ASC or DESC after the column names.

```
Example: READ: EMPLOYEE
               DB2NAME('DSN8230.EMP')
               WHERE(EMPNO = RESPEMP)
               ORDERBY(LASTNAME)
```

The above statement specifies that DB2 should return rows from the employee table in LASTNAME order. Therefore, if multiple rows existed for a given RESPEMP number, DB2 would return the row whose LASTNAME came first alphabetically. If no ORDERBY parm is specified and multiple rows meet the WHERE condition, DB2 will return the rows in an "arbitrary" order. Since MULTI was not specified in this example, Spectrum Writer uses only the *first* row returned to it by DB2.

READKEY(fieldname)

This parm is required for VSAM and EXIT files. Identifies the field that will be used as the key when performing random reads to the file. The manner in which this key value is used

to locate an input record depends on two other parms which may be present in the READ statement, as shown in the following table:

READ STATEMENT PARM COMBINATIONS		
GENERIC PARM?	KGE PARM?	DESCRIPTION
No	**No**	The record will be read whose full key exactly matches the READKEY value. If no such record is found, the record will be "missing." The READKEY field should be the same length as the defined key length for the file. If MULTI is also specified, Spectrum Writer will read *all records* whose full key matches the READKEY value. If MULTI is not specified, only the first record with a matching key will be read.
Yes	**No**	The record will be read whose partial key matches the partial key in the READKEY value. The READKEY field may be any length less than or equal to the defined key length for the file. If MULTI is also specified, Spectrum Writer will read *all records* whose partial key matches the READKEY value. If MULTI is not specified, only the first record with a matching partial key will be read.
No	**Yes**	The record will be read whose full key matches the READKEY value. If no record matches the READKEY value, then the record with the next greater key value will be read instead. The READKEY field should be the same length as the defined key length for the file. The MULTI parm may not be specified when KGE is specified.
Yes	**Yes**	The record will be read whose partial key matches the partial key in the READKEY value. If no record matches the READKEY value, then the record with the next greater partial key value is read instead. The READKEY field may be any length less than or equal to the defined key length for the file. The MULTI parm may not be specified when KGE is specified.

The contents of the READKEY field is always used "*as is*" when performing the read. Therefore, the key field must have the same format as the file's key values. You may need to use a COMPUTE statement to build an acceptable READKEY field. (Only *character* type COMPUTE fields may be used as read keys. See page 79, as well as below, for examples of computing a read key.)

The READKEY field must be available at the time the READ statement is processed. Therefore, it must be either:

- a field from the **primary input file**
- a field from an **earlier auxiliary input file**
- a **character type computed field** (defined in a preceding COMPUTE statement). **Note:** if the key to an auxiliary input file contains packed or binary data, use the

#FORMAT function in a COMPUTE statement to build a character field containing the data in the PACKED or BINARY display format.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)
```

The above example specifies that the EMPL–NUM field will be used as the key when reading records from the EMPL–FILE file. The EMPL–NUM field must exist in a previously specified input file. For the read to be successful, an exact, full–key match must be found in the EMPL–FILE.

```
Example: COMPUTE: BINARY-DEPT-NUM = #FORMAT(DEPT-NUM,BINARY,2)
         READ:    DEPARTMENT-FILE  READKEY(BINARY-DEPT-NUM)
```

The above example illustrates how to create a **key in binary format**. Assume that the DEPARTMENT–FILE uses the department number formatted as a 2–byte binary field for its key. The regular DEPT–NUM field is defined as a NUMERIC type numeric field (see Appendix F, "Files Used in Examples" on page 648) and would not work as the READKEY in this case, since it is not in binary format. The COMPUTE statement above creates a new 2-byte character field to be used when reading records from the DEPARTMENT–FILE. The contents of the 2 bytes is the department number, formatted in binary format. That field can be used as the READKEY to the DEPARTMENT–FILE. Since neither KGE nor GENERIC is specified, an exact full–key match is again required for the read to be successful.

The following example is similar, but assumes that the DEPARTMENT–FILE requires a 4-byte **packed read key**:

```
COMPUTE: PACKED-KEY = #FORMAT(DEPT-NUM,PACKED,4)
READ:    DEPARTMENT-FILE  READKEY(PACKED-KEY)
```

RECNAME(name/filename)

Specifies a record name to use when referring to fields in this input file. This is especially useful when you will be reading multiple records from the same input file (by using multiple READ statements). The RECNAME parm (in each statement) can be used to assign unique names to each record read from the file. You may give the record any name you like, within the rules governing names given on page 446. The use of the RECNAME parm is discussed beginning on page 228.

If no RECNAME parm is specified, the *filename* is used as the record name.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  RECNAME(EMP)
```

The above example specifies that the records read from the EMPL–FILE file will be named EMP. Assume that a field named DATE exists in both this file and in some other input file. You can use the record name EMP to indicate that you are referring to the DATE field in the EMPL–FILE, like this:

```
COLUMNS: EMP.DATE
```

SHOWFLDS(YES/NO)

Specifies whether Spectrum Writer should print a list of all fields that have been defined for the file. (For DB2 inputs, the DB2 columns defined for the DB2 table are listed.) This list appears immediately after the READ statement in Spectrum Writer's control statement listing. The list will include the data type of each field (character, numeric, date, time or bit). Use this parm if you aren't sure of the names or spellings of the fields (or DB2 columns) in your input file.

```
Example: READ: EMPLOYEE
               DB2NAME('DSN8230.EMP')
               WHERE(EMPNO = RESPEMP)
               SHOWFLDS(YES)
```

The above statement causes a list to be printed showing each DB2 field defined for the DSN8230.EMP table.

TYPE(VSAM/DB2/EXIT)

OS/390 only. Specifies an override file type for the input file (for the current run only). If this parm is omitted, the file type will be taken from the FILE statement that defined the file. A complete list of file types is given under the FILE statement description, on page 536.

Note: SEQ type files (sequential or "flat" files) may *not* be specified in the READ statement.

```
Example: READ: EMPL-FILE  READKEY(EMPL-NUM)  TYPE(VSAM)
```

The above example specifies that the VSAM access method should be used when reading the EMPL–FILE file, regardless of the file type specified when the file was originally defined.

WHERE(search–condition)

This parm is required for DB2 inputs and not allowed for other inputs. It performs the same function that the READKEY parm performs for VSAM files. For each record read from the primary input file, Spectrum Writer will ask DB2 for one or more rows from this auxiliary input file. Use this parm to specify a "search condition" to instruct DB2 which row(s) from the DB2 table to pass to Spectrum Writer. The syntax of the search–condition is generally the same as DB2's syntax for the WHERE clause in a DB2 SELECT statement. The use of this parm in a READ statement is discussed in the section beginning on page 400. Its syntax is discussed in the section beginning on page 405.

```
Example: INPUT: PROJECT
                DB2NAME('DSN8230.PROJ')

          READ:  EMPLOYEE
                 DB2NAME('DSN8230.EMP')
                 WHERE(EMPNO = RESPEMP)
```

Here's how Spectrum Writer processes the above statements. The primary input to the report is the project DB2 table. So, Spectrum Writer will retrieve all rows from that DB2 table. After it fetches each row from the project table, Spectrum Writer will now also fetch one row from the employee table. The row from the employee table will be the one whose EMPNO field equals the RESPEMP field from the project table. If MULTI had also been specified in the READ statement, Spectrum Writer would fetch *all such rows* (in essence, a "one-to-many" joining of the tables). When MULTI is not specified, Spectrum Writer fetches just the first such row.

NOTES

How Auxiliary Input Files are Processed

The **primary** input file for a report is always read *sequentially* —usually from beginning to end. **Auxiliary** input files are handled differently. They are read randomly (or "directly") using either a "read key" or a WHERE expression to determine which record(s) to read.

This section explains in more detail how Spectrum Writer processes multiple input files.

Program Flow With No READ Statements

To understand how auxiliary input files are processed, let's first notice how Spectrum Writer produces a report when *no* auxiliary input files are used. In such a case, Spectrum Writer repeats the following steps over and over.

1. Read a record from the primary input file
2. Evaluate the INCLUDEIF statement using the data from this input record
3. If the record passes the INCLUDEIF tests, pass the record to Spectrum Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
4. If the record does not pass the INCLUDEIF tests, discard the record

The above steps are repeated until all records from the primary input file have been read. (Or, possibly earlier if a KEYRANGE or STOPWHEN parm has been specified.)

Program Flow with READ Statements

The flow described above remains basically the same when one or more auxiliary input files are added to the request. The only difference is in Step 1 above. Instead of simply reading records from the primary input file, Spectrum Writer now assembles "**logical input records**." A logical input record is a group of records consisting of one record from each input file.

The records from the primary input file are still read sequentially. The records from the auxiliary input files are read using a READKEY (or a WHERE clause). Once assembled, this group of records is then treated by Spectrum Writer as one, big logical input record containing all of the data fields from all of the input files. Steps 2 through 4 of the program flow remain the same — it's just that they are now performed on this logical record rather than on the primary input record alone.

1. Assemble a "logical input record" consisting of one record from each of the input files
2. Evaluate the INCLUDEIF statement using the data from this logical input record
3. If the logical input record passes the INCLUDEIF tests, pass the logical input record to Spectrum Writer's output phase (where it will be sorted and formatted into the desired report or PC file)

4. If the logical input record does not pass the INCLUDEIF tests, discard the logical input record

The specific way that Spectrum Writer assembles its logical input records (in Step 1) is different depending on whether any READ statements use the MULTI parm. The next two sections explain how Spectrum Writer assembles its logical records in each case.

Program Flow Without MULTI–type READ Statements

When none of the READ statements uses the MULTI parm, Spectrum Writer assembles one logical record for each record it reads from the primary input file. The primary input file is still read sequentially, from beginning to end. Each time Spectrum Writer reads a new record from the primary input file, it also reads a single record from each of the auxiliary input files. This group of related records, one from each input file, is treated as a logical input record.

Now the program flow can be described this way:

1. Read a record from the primary input file
2. Create one logical input record by also reading a single record from each auxiliary input file
3. Evaluate the INCLUDEIF statement using the data from this logical input record
4. If the logical input record passes the INCLUDEIF tests, pass the logical input record to Spectrum Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
5. If the logical input record does not pass the INCLUDEIF tests, discard the logical input record

The above steps are repeated until all records from the primary input file have been read. Note that when no MULTI parm is used, the number of logical records processed is the same as the number of primary input file records.

Note: The steps above describe what Spectrum Writer does *logically*. During actual processing, there may be cases where it is not necessary for Spectrum Writer to read a particular record from an auxiliary input file. For example, if the INCLUDEIF statement eliminates a primary input record without referring to fields from any auxiliary input files, it is not necessary to read the records from those files. The next primary input record can be read right away. For run–time efficiency, individual records are not read from auxiliary files when they are not actually needed to correctly process a request.

Program Flow With MULTI–type READ Statements

When one or more READ statements *with* a MULTI parm is used in a request, Spectrum Writer uses a different process to assemble logical records.

Let's consider a simple request that uses a single READ statement. Assume that the READ statement contains the MULTI parm. Rather than only reading a single record from the auxiliary input file each time, Spectrum Writer must now read *all* records that match the

READKEY value (or the WHERE clause). So now, each time a primary input file record is read, *all* of the qualifying auxiliary input file records must be read and, one at a time, combined with the primary input record to form multiple logical input records. Only after all of the qualifying auxiliary input file records have been processed can the next primary input file record be read.

You can see that when a MULTI–type READ statement is used, the number of logical input records processed can be far greater than the number of primary input file records.

When two (or more) READ statements have the MULTI parm, the process is similar to that just described. But now the number of record combinations that Spectrum Writer must assemble into logical records increases exponentially. For each primary input file record, Spectrum Writer must build one logical input record using every possible, unique combination of auxiliary input file records that are related to that primary input file record.

The program flow can now be described this way:

1. Read a record from the primary input file
2. Build as many logical input records as possible using this primary input record and all combinations of records read from the auxiliary input file(s)
3. For each logical input record, evaluate the INCLUDEIF statement using the data from that logical input record
4. If the logical input record passes the INCLUDEIF tests, pass the logical input record to Spectrum Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
5. If the logical input record does not pass the INCLUDEIF tests, discard the logical input record

The above steps are repeated until all records from the primary input file have been read.

Note: You may have a report request that uses some READ statements that have the MULTI parm and some READ statements that do *not* have it. In that case, the above flow is still used. When assembling logical records from the combinations of qualifying records from each file, the READ statements without the MULTI parm will always contribute only one qualifying record.

Note: Whenever an auxiliary input file does not have any qualifying records to contribute to the logical record, a single "missing record" from that file will be used in building the logical record combinations. This is true whether or not the MULTI parm is used in the READ statement.

Speed-Up Tip: READ statements with the MULTI parm are less efficient than regular READ statements. To reduce CPU and I/O usage, do not specify MULTI if you know that a file contains unique keys. (In other words, do not specify MULTI if you know the READKEY will only find one matching record in the file.)

Speed-Up Tip: When mixing READ statements with and without the MULTI parm, put the READ statements *without* the MULTI parm ahead of the READ statements with the MULTI parm whenever possible. This improves performance by reducing the amount of I/O required to assemble all of the possible record combinations.

Missing Records

Sometimes there will not be any record in an auxiliary file that matches READKEY value (or the WHERE expression). When this happens, Spectrum Writer assigns a default value to each of the fields in the missing record. The default value depends on the type of the field, as shown in the following table:

VALUE ASSIGNED TO FIELDS IN MISSING RECORDS	
FIELD TYPE	**DEFAULT VALUE**
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

See page 230 for a method you can use to determine whether a particular record is missing or not.

SORT Statement

PURPOSE

This statement specifies how Spectrum Writer should sort the included input file records before writing the report or PC file. A SORT statement is not required. If no SORT statement is found, no sort will be performed and the output will be in the original order of the input file.

Only one SORT statement is allowed, but it may contain as many sort fields as you like.

The SORT statement can also be used to specify control breaks.

FEATURES

Use the SORT statement to:

- specify the **sort fields** to be used for the report or PC file
- specify whether to sort each field into **ascending** or **descending** order
- specify that a **control break** should occur whenever the contents of a sort field changes
- specify the **control break spacing** to use at control breaks
- specify which **statistics lines**, if any, to print at control breaks

LEARNING MORE

The complete syntax of the SORT statement is shown on the following pages. In addition, the following parts of the manual relate to the SORT statement:

- a lesson on using the SORT statement in reports begins on page 62
- a lesson on using the SORT statement in PC files begins on page 105
- the use of the SORT statement to request control breaks is discussed beginning on page 177
- the OPTIONS statement also has several options that affect the sort process. These are listed on page 601.

SYNTAX

SORT STATEMENT SYNTAX

```
SORT: fieldname[(parms)] fieldname[(parms)] ...  [ #EQUALS ]
```

where **parms** can be one or more of the following (separated by commas or blanks):

```
ASC/DESC
AVERAGE
MAXIMUM
MINIMUM
n/PAGE/PAGE1/NEWSHEET/NEWSHEET1/ODDPAGE/ODDPAGE1
NZAVERAGE
NZMINIMUM
TOTAL/NOTOTAL
```

Standard Spelling	Alternate Spellings
#EQUALS	#EQUAL, #EQ
ASC	A
AVERAGE	AVER, AVG
DESC	D
MAXIMUM	MAX
MINIMUM	MIN
NOTOTAL	NOTOTALS, NOTOT, NOTOTS
NZAVERAGE	NZAVER, NZAVG
NZMINIMUM	NZMIN
PAGE	PG, P
SORT	SRT
TOTAL	TOTALS, TOT, TOTS

Only one or more fieldnames (or the #EQUALS parm) is required. All other parms are optional.

> **Note:** Use the AUTOSORT option (in an OPTIONS statement) if you want Spectrum Writer to automatically sort your report or PC file on its first five columns of data.

Specifying any parm other than ASC or DESC for a field makes that field a control break field. Specifically, the parms that cause a control break are:

- the TOTAL or NOTOTAL parm. (Specifying TOTAL results in a control break with totals; NOTOTAL results in a control break without totals.)
- a break spacing parm (such as PAGE, NEWSHEET, 3, etc.)
- a statistical parm (such as AVERAGE, MAXIMUM, etc.)

fieldname[(parms)]

Specifies a field on which the output is to be sorted, and optionally specifies additional processing information about the field. You are *not restricted* to sorting on fields that appear in the report. You may sort on a field which does not appear anywhere else in the

report. Of course, the field must be available to Spectrum Writer at the time the SORT statement is processed. That is, the field must be one of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)

No parms are required with the fieldname. If desired, specify one or more parms by placing them in parentheses immediately after the fieldname. (Do not leave a space before the parenthesis.) Separate the parms with a comma and/or blanks.

```
Example: SORT: REGION EMPL-NAME
```

The above example will cause the report to be sorted in REGION order and — within each region — in EMPL–NAME order.

#EQUALS

This parm can be used only as the *last item* (or only item) in a SORT statement. It specifies that, if after sorting on all of the preceding sort fields there are still some ties, the tie records should be left in the same relative order that they had in the input file. This is useful if the records in your input file are already in some special order, and you want to preserve that relative order.

```
Example: SORT: REGION #EQUALS
```

The above SORT statement causes the records to be sorted by REGION. However, within REGION, the records will not be sorted on any additional field. Instead, the #EQUALS parm specifies that the records within a region will be printed in the same relative order in which they appeared in the input file.

ASC/DESC

Specifies ascending or descending sort order. The default sort order is ascending.

```
Example: SORT: REGION(DESC) EMPL-NAME
```

The above example will cause the report to be sorted in *descending* REGION order. The last region (alphabetically) will print first, and the first region will print last. Within a region, the records will be further sorted on (ascending) employee name.

AVERAGE

Specifies that a control break should occur whenever the value of the sort field changes, and specifies that average values should be printed at the break. At the control break, a line will print showing each numeric column's average value in the control group just ended.

```
Example: SORT: REGION(AVERAGE) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. In addition to the totals line (which prints by default), an average line will print at the break.

MAXIMUM

Specifies that a control break should occur whenever the value of the sort field changes, and specifies that maximum values should be printed at the break. At the control break, a

line will print showing each accumulated column's maximum value in the control group just ended.

```
Example: SORT: REGION(MAXIMUM) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. In addition to the totals line (which prints by default), a maximum line will print at the break.

MINIMUM

Specifies that a control break should occur whenever the value of the sort field changes, and specifies that minimum values should be printed at the break. At the control break, a line will print showing each accumulated column's minimum value in the control group just ended.

```
Example: SORT: REGION(MINIMUM) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. In addition to the totals line (which prints by default), a minimum line will print at the break.

n/PAGE/PAGE1/NEWSHEET/NEWSHEET1/ODDPAGE/ODDPAGE1

Specifies that a control break should occur whenever the value of the sort field changes, and specifies the spacing to use at the control break. Unless overridden with the NOTOTAL parm, a line of totals will also print at the control break. After the totals line, the spacing specified with this parm will be performed.

A numeric value (n) specifies a number of *blank lines* to print at the break. All of the other parms cause the report to skip to a *new page* after the control break. For a description of each of these break spacing parms, see "How to Change the Control Break Spacing" (page 178).

```
Example: SORT: REGION(PAGE) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. After printing regions totals at the break, the report will skip to a new page.

NZAVERAGE

Specifies that a control break should occur whenever the value of the sort field changes, and specifies that non–zero average values should be printed at the break. At the control break, a line will print showing each accumulated column's average value (computed without considering any zero values) in the control group just ended.

```
Example: SORT: REGION(NZAVERAGE) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. In addition to the totals line (which prints by default), a non–zero average line will print at the break.

NZMINIMUM

Specifies that a control break should occur whenever the value of the sort field changes, and specifies that non–zero minimum values should be printed at the break. At the control

break, a line will print showing each accumulated column's minimum value (not considering zero values) in the control group just ended.

```
Example: SORT: REGION(NZMINIMUM) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. In addition to the totals line (which prints by default), a non–zero minimum line will print at the break.

TOTAL/NOTOTAL

Specifies that a control break should occur whenever the value of the sort field changes, and specifies whether or not to print totals at the control break.

The TOTAL parm specifies that totals are wanted at the control break. After the total line prints, the break spacing will be performed.

> **Note:** If a break spacing parm or any other statistical parm has been specified (indicating that a control break is desired), it is *not necessary* to also specify the TOTAL parm. The total line prints by default at all control breaks.

The NOTOTAL parm specifies that totals are not wanted at the break— only the break spacing is wanted. Unless overridden with a break spacing parm, two blank lines will print at the control break.

```
Example: SORT: REGION(TOTAL) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. Totals for the preceding region will print, followed by two blank lines.

```
Example: SORT: REGION(NOTOTAL) EMPL-NAME
```

The above example will cause the report to be sorted in region order, and then employee name order. A control break will occur every time a new region is about to print. However, a totals line will *not* print at the break. Only two blank lines will print.

NOTES

How Spectrum Writer Determines Sort Order

All data processed by Spectrum Writer falls into one of five general categories of data. The following table shows how each type of data is sorted:

HOW DIFFERENT TYPES OF DATA ARE SORTED	
DATA TYPE	**DESCRIPTION**
Character	Character fields are sorted into alphabetical order (based on their EBCDIC values). The letter "A" sorts before the letter "B", etc. Numerals ("1", "2", etc.) sort after the letter "Z". Special symbols such as parentheses, commas, dashes, etc. sort before the letter "A." Also, all lower case letters ("a" through "z") sort before the first upper case letter ("A"). If you want to sort on a field which contains mixed case letters, you may wish to first convert the field to all upper–case. That way, fields containing the same words will sort together, even if the words are capitalized differently. Use the #UCASE built–in function to create an all upper–case version of the desired field. Then, sort on that computed field. **Note:** The *full contents* of character fields are sorted, not just the portion that may appear in a report column. In other words, even if you truncate a character field to make it fit into a report column, the field's full value will still be used for sorting purposes. The field's full value is also used to determine when a control break occurs.
Numeric	The signed algebraic value of numeric fields are sorted. Thus, all minus numbers will sort before the first positive number. **Note:** The true internal value of a field is what is sorted, not the formatted value that may appear in the report. In other words, commas, dollar signs, etc. *are not* considered when sorting numeric fields. Also, if you rounded out some of the decimal digits when displaying the field, those decimal digits are still considered when performing the sort (and when determining breaks, if the field is a control break field).
Date	Dates are sorted in year, month, and day order, regardless of how the raw data may have been stored in the input file, and regardless of how the date may be formatted in the report.
Time	Times are sorted in hours, minutes and seconds order, regardless of how the raw data may have been stored in the input file, and regardless of how the time may be formatted in the report.

How Different Types of Data are Sorted (Continued)	
Data Type	**Description**
Bit	Bit fields are sorted as either an OFF or ON. They are *not sorted* according to the text used to *display* them in the report (that is, the ONTEXT and OFFTEXT values). Bit fields which are OFF ("0") will sort before bit fields which are ON ("1"). **Note:** Depending on what ONTEXT and OFFTEXT values are used, a sorted bit field column may or may not appear in alphabetical order. You can always reverse the order, if desired, by specifying the DESC parm when sorting a bit field.
Note: A field which is in *error* is treated as a very low value when sorting. Thus, fields containing invalid packed data, for example, and displayed with the ****I**** error indicator, will sort ahead of fields containing valid numeric values.	

OPTIONS statement Options that Affect the Sort

Certain technical aspects of the Sort process can be specified by options in the OPTIONS statement. For details on these options, see under the OPTIONS statement in this chapter. The following table lists the OPTIONS statements options related to the Sort.

OPTIONS Statement Options Related to the Sort	
Option	**Description**
AUTOSORT	Tells Spectrum Writer to sort on the first 5 fields of the report.
NOSORTSIZE	Prevents Spectrum Writer from passing a sort size parameter (MAINSIZE) to the Sort program. (May be useful if you want your shop's default sort size parm to be used.)
SORTDD	Specifies the 4-byte prefix of the ddnames that the Sort program should use for its work files.
SORTNAME	Specifies the name of the Sort program to call.
SORTOPT	Specifies any special options you want Spectrum Writer to pass to the Sort program.
SORTSIZE	Specifies the sort size parm (MAINSIZE) that Spectrum Writer should pass to the Sort program.

TITLE Statement

PURPOSE

This statement specifies a title that should print at the top of each page of the report. You may have as many TITLE statements as you like. Each TITLE statement results in one title line at the top of your report.

Another use of TITLE statements is to create your own column headings, when you do not want the ones automatically created.

TITLE statements are ignored when producing PC files.

FEATURES

Use the TITLE statement to:

- specify the **contents** of the report titles (which can include literal text, data from input files, and special items like the current page number, date, time, etc.)
- specify how to **left align, center** and **right align** different parts of the same title
- specify the desired **width, display format**, and **justification** for data fields that appear in a title

LEARNING MORE

The complete syntax of the TITLE statement is shown on the following pages. In addition, the following parts of the manual relate to the TITLE statement:

- a lesson on using the TITLE statement begins on page 53
- advanced examples of using the TITLE statement are shown beginning on page 161
- the use of TITLE statements to create column headings is discussed in "How to Produce Multi–Line Reports" (page 151)

SYNTAX

TITLE STATEMENT SYNTAX

```
TITLE:  print-expression [/ print-expression] [/ print-expression]
```

Note: the syntax for the print-expressions is shown on page 604.

Standard Spelling	Alternate Spellings
TITLE	TITL, TIT

The TITLE statement consists of from one to three print expressions, separated with slashes. If a TITLE statement has no slashes, the single print expression will be centered over the report. If there is one slash, the first print expression will be left aligned and the second print expression will be right aligned over the report. If there are two slashes, the first print expression will be left aligned, the second one will be centered, and the third one will be right aligned. It is okay for one or more of the print expressions to be empty. Examples of using various combinations of print expressions and slashes is illustrated in the section beginning on page 168.

You may also use empty TITLE statements. An empty TITLE statement results in one blank title line.

Note: Any title line that contains only spaces and underscore characters will be overprinted (that is, printed without advancing to the next line). Use this feature to underline column headings that you create with TITLE statements.

Note: Use the similar FOOTNOTE statement to print title lines at the bottom of each page of the report. The print expression syntax on the following pages applies to both FOOTNOTE and TITLE statements.

PRINT–EXPRESSION SYNTAX (IN TITLE AND FOOTNOTE STATEMENTS)

A **print–expression** consists of one or more **items**, optionally separated by numeric **spacing factors**:

```
TITLE:      [n] item [n] item [n] item ...
       [ / [n] item [n] item [n] item ... ]
       [ / [n] item [n] item [n] item ... ]
```

Each **item** can be either a **fieldname** or a **literal text**. Each item can optionally be followed by a parm list in parentheses:

```
fieldname[(  [   ASCII                    ]
             [   BIZ                      ]
             [   display-format           ]
             [   LEFT/CENTER/RIGHT        ]
             [   width                    ]   )]

'literal'[(      width                        )]
```

Standard Spelling	Alternate Spellings
CENTER	CJ
LEFT	LJ
RIGHT	RJ
TITLE	TITL, TIT

fieldname

Specifies that the title line should contain the contents of this field. The field's data will be taken from the *first detail record* on the new page. (In footnote lines, the field's data will be taken from the *last detail record* on the page.)

The field must be available to Spectrum Writer at the time the TITLE statement is processed. That is, the field name must be one of the following:

- a field from an **input** file. (An input file is a file named in the INPUT statement, or in an optional READ statement.)
- a **computed** field (defined in a preceding COMPUTE statement)
- a **built–in** field. (See Appendix C, "Built-In Fields" on page 624 for a complete list of built–in fields.)

Note that in addition to the standard built–in fields, there is one special built–in field that can be used only in the TITLE and FOOTNOTE statements. That is the #PAGENUM built–in field, which contains the current page number. By default, #PAGENUM shows four digits, in this format: ZZZ9. You can override this format by using any numeric display format of your choice (see page 605). This built-in field can also be abbreviated as #PAGE.

Example: `TITLE: #TODAY / 'ABC COMPANY' / 'PAGE' #PAGENUM`

The above example contains three print expressions. It will produce a title line which looks like this:

```
12/31/99                          ABC COMPANY                          PAGE nnnn
```

The literal texts ("ABC COMPANY", and "PAGE") print as specified. The contents of the built–in fields #TODAY and #PAGENUM also print, in default format. The first part of the title is left aligned; the second part is centered; the third part is right aligned.

'literal'

Specifies that the title line should contain this literal text. Enclose the literal text in either apostrophes or quotation marks.

Example: See the example above under the fieldname parm.

n

This is a numeric spacing factor. It specifies how many blank spaces to leave between two items in a title line. A spacing factor of zero is allowed. (It results in two items appearing in the title with no blank spaces between them.) If no spacing factor is given, the default is to leave one blank space between items.

Example:
```
TITLE: #TODAY  /  'ABC COMPANY'  /  'PAGE' 6 #PAGENUM
```

The above example specifies that six blank spaces should be left between the literal text "PAGE" and the contents of the #PAGENUM field. The title would now look like this:

```
12/31/99                          ABC COMPANY                          PAGE      nnnn
```

ASCII

Specifies that the final, formatted field should be translated from EBCDIC to ASCII in the print line. To specify your own EBCDIC-to-ASCII translation table, use the ASCIITABLE option in the OPTIONS statement (page 558). Otherwise, Spectrum Writer uses a default translation table. See page 143 for more information on creating ASCII output files.

Example:
```
TITLE: REGION(ASCII)
```

The above example causes the REGION field to be printed in ASCII.

BIZ

This “blank if zero” parm specifies that blanks should appear in the title for the field if it has a value of zero. This parm is allowed only for numeric, date and time fields. A date is considered to have a zero value if the month, day and last two digits of the year are all zeros (regardless of the value of the century part of the year).

Example:
```
TITLE: 'EMPLOYEES HIRED ON' HIRE-DATE(BIZ)
```

The above example causes the HIRE–DATE field in the title to be left blank whenever it contains a zero date.

display–format

Specifies how the contents of a field should be formatted in the title line. A complete list of display formats is found in Appendix B, "Display Formats" (page 617). If this parm is not specified, Spectrum Writer uses the display format from:

- the FIELD or COMPUTE statement that defined the field

- an OPTIONS statement FORMAT parm
- the default display format (page 618)

```
Example: TITLE: #TODAY(LONG1)  /  'ABC COMPANY'  /  'PAGE  #PAGENUM(PIC'999')
```

The above example specifies display formats for the #TODAY and the #PAGENUM fields. The LONG1 display format causes the month name in the date to be spelled out. The PICTURE display format (for #PAGENUM) specifies that three digits of the page number should be displayed, and that leading zeros should *not* be suppressed. The title line would now look like this:

```
DECEMBER 31, 1999                    ABC COMPANY                              PAGE 001
```

LEFT/CENTER/RIGHT

Specifies how a field's data should be justified *within* the space allocated for it in the title line. If none of these parms is specified, no justification is performed.

```
Example: TITLE: #TODAY(LONG1,CENTER)
```

The above example specifies a title line that simply contains the current date, displayed in LONG1 format. The LONG1 format causes 18 bytes to be reserved for the date in the title line. This is to allow enough room to print the biggest possible date (like "SEPTEMBER 31, 1999"). The 18–byte area reserved for the date will automatically be centered over the body of the report, since no slashes are used. But shorter dates (like "MAY 1, 1990") would not take up the entire 18–byte area, and thus would not appear to be centered correctly in the title. The CENTER parm is needed to cause these shorter dates to be *centered within* the 18–byte area in the title line. The title line produced by the above statement would look like this:

```
                                  MAY 1, 1990
```

A similar situation arises when you want to align a date with the *right* margin of a report. By using a slash you can cause the whole 18–byte area to be right aligned. But a small date ("MAY 1, 1990") would not use up the entire 18 bytes, and thus would not be flush with the right edge of your report. To solve that problem, use the RIGHT justification parm to right–justify the date *within* its 18–byte area, like this:

```
TITLE: 'ABC COMPANY'  /  #TODAY(LONG1,RIGHT)
```

The title line produced by the above statement would look like this:

```
ABC COMPANY                                                          MAY 1, 1990
```

width

This is a numeric parm that specifies the number of characters to reserve for an item in the title line. Use this parm if the default width is too large or too small.

```
Example: TITLE: 'PAGE' #PAGENUM(9)
```

The above example specifies that 9 characters (not digits) should be reserved to display the #PAGENUM field in the title line. The resulting title would look like this:

```
                                  PAGE n,nnn,nnn
```

Appendices

Appendices Table of Contents

Appendix A. Data Types 611
Character Data Types 611
Numeric Data Types 612
Date Data Types 613
Time Data Types 615
Bit Data Types 618

Appendix B. Display Formats 619
Default Display Formats 620
Display Formats for Any Type of Field 620
Numeric Display Formats 621
Date Display Formats 622
Time Display Formats 624

Appendix C. Built-In Fields 626
Character Built–In Fields 627
Numeric Built-In Fields 628
Date Built-In Fields 628
Time Built-In Fields 629

Appendix D. Built-In Functions 630
Functions that Return a Character Value 632
Functions that Return a Numeric Value 637
Functions that Return a Date Value 640
Functions that Return a Time Value 642
Functions that Return a Boolean (or Bit) Value 644

Appendix E. Error Indicators 646
Suppressing Error Indicators 648
Propagation of Error Indicators 649
Determining if a Field Is In Error 649

Appendix F. Files Used in Examples 650
Sample File Definitions 650
Sample Files' Raw Data 652

Appendix G. Speed-Up Tips 654
INCLUDEIF Statement 654
Conditional COMPUTE Statements 657
Compute Statements with RETAIN 658
Intermediate Computational Expressions 659
Intermediate Conditional Expressions 659
Read Statements with the MULTI parm 660

Use the STOPWHEN Parm for Non-Keyed Files 663
Replace an Auxiliary File with a “Table Lookup” 663
Clearing I/O Areas 664
Fine-Tuning the Sort 664
Development Cycle 665
Using Explicit Literals in Conditional Expressions 666

Appendix H. Sample Data Exit Programs 668
Sample Assembler Data Exit Program 668
Sample Cobol Data Exit Program 673

Appendix I. I/O Exits 675

Appendix A. Data Types

There are five general categories of data that Spectrum Writer recognizes. They are:

- character
- numeric
- date
- time
- bit

For each of these categories, there is more than one way that the data can actually be represented in an input record. A **data type** describes exactly how a particular field's data is stored within an input record. A field's data type is defined to Spectrum Writer with the TYPE parm in its FIELD statement.

Here are some examples of TYPE parms in FIELD statements:

```
FIELD: AMOUNT TYPE(NUM) LENGTH(6) DECIMAL(2)
FIELD: PACKED-SALARY TYPE(PACKED) LENGTH(4) DECIMAL(2)
FIELD: INDEX TYPE(BIN) LENGTH(1)
FIELD: SALES-DATE TYPE(YYMMSS)
FIELD: PACKED-JULIAN-START-DATE TYPE(P-YYDDD)
FIELD: SALES-TIME TYPE(HHMMSS)
```

The following charts show the data types that Spectrum Writer supports for each category of data. The charts also show the acceptable abbreviations and alternate spellings for the data types.

Character Data Types

DATA TYPES FOR CHARACTER FIELDS		
DATA TYPE	**DESCRIPTION**	**LENGTH ALLOWED**
CHARACTER **CHAR** **CH** **C**	Character data	1 to 32,767
CHAREXIT	Spectrum Writer will call a user–written exit program to obtain a character string.	1 to 32,767

Numeric Data Types

DATA TYPES FOR NUMERIC FIELDS			
DATA TYPE	**DESCRIPTION**	**PROGRAMMING LANGUAGE EQUIVALENTS**	**LENGTH ALLOWED** (See Note 1)

NUMERIC
NUM
DISPLAY
DISP
	Display numeric.
	Example: `C'1234', C'1234.0', C'+1234', C' 1234 ', C' $1,234'` are all 1,234.
	Example: `C'-1234'` is –1,234.
		`COBOL: USAGE DISPLAY`
		`PIC 9999`
		`PIC S9999 SIGN IS SEPARATE`
		`PL/1: PIC '9999'`
		`ASM: DS C`
			1–256

NUMERIC–SLD
NUM–SLD
	Numeric with Signed Last Digit.
	Example: `C'1234'` and `C'123D'` are 1,234.
	Example: `C'123M'` is –1,234.
		`COBOL: PIC S9999`
		`PL/1: PIC '999T'`
		`ASM: DS Z`
			1–256

NUMERIC–CD **NUM–CD**	Numeric with Comma for Decimal symbol. Example: `C'1.234.567,89'` and `C'1 234 567,8'` are valid values.	`(None)`	1–256

PACKED
PACK
P
COMP–3
	Packed decimal (signed).
	Example: `X'01234F', X'01234C'` are 1,234.
	Example: `X'01234D'` is –1,234.
		`COBOL: PIC S9999`
		`USAGE COMP-3`
		`PL/1: FIXED DECIMAL`
		`ASM: DS P`
			1–16

PACKEDUN **PACKUN** **PU**	Packed decimal unsigned (BCD). Example: `X'1234'` is 1,234.	`(None)`	1–16

BINARY
BIN
COMP
	Binary (signed).
	Example: `X'04D2'` is 1,234.
	Example: `X'FB2E'` is –1,234.
	Example: `X'FF'` is –1.
		`COBOL: PIC S9999`
		`USAGE COMP`
		`PL/1: FIXED BINARY`
		`ASM: DS H`
		`DS F`
			1–8

BINARYUN **BINUN** **BU**	Binary unsigned. Example: `X'04D2'` is 1,234. Example: `X'FB2E'` is 64,302. Example: `X'FF'` is 255.	`COBOL: PIC 9999 COMP` `ASM: DS A`	1–8

HALFWORD
HALF
	Same as BINARY but defaults to a length of 2 when no length is specified.
	Example: `X'04D2'` is 1,234.
	Example: `X'FB2E'` is –1,234.
		`COBOL: PIC S9(4) COMP`
		`PL/1: FIXED BIN(15)`
		`ASM: DS H`
			1–8

FULLWORD
FULL
	Same as BINARY but defaults to a length of 4 when no length is specified.
	Example: `X'000004D2'` is 1,234.
	Example: `X'FFFFFB2E'` is –1,234.
		`COBOL: PIC S9(8) COMP`
		`PL/1: FIXED BIN(31)`
		`ASM: DS F`
			1–8

NUMEXIT	Spectrum Writer will call a user–written exit program to obtain a numeric value. The exit program must return a 16–byte packed number (optionally containing decimal digits).	`COBOL: CALL`
		`PL/1: CALL`
		`ASM: GOTO`
			N/A

DATA TYPES FOR NUMERIC FIELDS (CONTINUED)			
DATA TYPE	DESCRIPTION	PROGRAMMING LANGUAGE EQUIVALENTS	LENGTH ALLOWED (See Note 1)

Notes:
(1) Lengths indicate the number of bytes occupied in the input record, not the number of digits. The maximum number of digits (including any decimal digits) allowed in any numeric field is 31.

Date Data Types

DATA TYPES FOR DATE FIELDS		
DATA TYPE	DESCRIPTION (See Note 1)	LENGTH
MM–DD–YY	MM/DD/YY date in character format (with slashes or other delimiters). (2) Leading zeros are optional in day and month. Example: `C'12/31/96'` and `C'12.31.96'` are Dec. 31, 1996. Example: `C'1/2/96 '` and `C' 1/2/96'` are Jan. 2, 1996.	8
MM–DD–YYYY	MM/DD/YYYY date in character format (with slashes or other delimiters). (2) Leading zeros are optional in day and month. Example: `C'12/31/1996'` and `C'12.31.1996'` are Dec. 31, 1996. Example: `C'1/2/1996 '` and `C' 1/2/1996'`. are Jan. 2, 1996.	10
MMDDYY	MMDDYY date in character format. Example: `C'123196'` is Dec. 31, 1996.	6
MMDDYYYY	MMDDYYYY date in character format. Example: `C'12311996'` is Dec. 31, 1996.	8
DD–MM–YY	DD/MM/YY date in character format (with slashes or other delimiters). (2) Leading zeros are optional in day and month. Example: `C'31/12/96'` and `C'31.12.96 '` are Dec. 31, 1996. Example: `C'2/1/96 '` and `C' 2/1/96'` are Jan. 2, 1996.	8
DD–MM–YYYY	DD/MM/YYYY date in character format (with slashes or other delimiters). (2) Leading zeros are optional in day and month. Example: `C'31/12/1996'` and `C'31.12.1996'` are Dec. 31, 1996. Example: `C'2/1/1996 '` and `C' 2/1/1996'` are Jan. 2, 1996.	10
DDMMYY	DDMMYY date in character format. Example: `C'311296'` is Dec. 31, 1996.	6
DDMMYYYY	DDMMYYYY date in character format. Example: `C'31121996'` is Dec. 31, 1996.	8
YYYY–MM–DD	YYYY/MM/DD date in character format (with slashes or other delimiters). (2) Example: `C'1996/12/31'` and `C'1996.12.31'` are Dec. 31, 1996. Example: `C'1/2/1996 '` and `C' 1/2/1996'` are Jan. 2, 1996.	10
YYMMDD	YYMMDD date in character format. Example: `C'961231'` is Dec. 31, 1996.	6

DATA TYPES FOR DATE FIELDS (CONTINUED)		
DATA TYPE	**DESCRIPTION** (See Note 1)	**LENGTH**
YYYYMMDD	YYYYMMDD date in character format. Example: `C'19961231'` is Dec. 31, 1996.	8
YYYY–DD–MM	YYYY/DD/MM date in character format (with slashes or other delimiters). (2) Example: `C'1996/31/12'` and `C'1996.31.12'` are Dec. 31, 1996. Example: `C'1996/2/1 '` and `C' 1996/2/1'` are Jan. 2, 1996.	10
YYDDD	YYDDD Julian date in character format. Example: `C'96366'` is Dec. 31, 1996.	5
YYYYDDD	YYYYDDD Julian date in character format. Example: `C'1996366'` is Dec. 31, 1996.	7
H–MMDDYY	MMDDYY date in hexadecimal (BCD) format. Example: `X'123196'` is Dec. 31, 1996.	3
H–MMDDYYYY	MMDDYYYY date in hexadecimal (BCD) format. Example: `X'12311996'` is Dec. 31, 1996.	4
H–DDMMYY	DDMMYY date in hexadecimal (BCD) format. Example: `X'311296'` is Dec. 31, 1996.	3
H–DDMMYYYY	DDMMYYYY date in hexadecimal (BCD) format. Example: `X'31121996'` is Dec. 31, 1996.	4
H–YYMMDD	YYMMDD date in hexadecimal (BCD) format. Example: `X'961231'` is Dec. 31, 1996.	3
H–YYYYMMDD	YYYYMMDD date in hexadecimal (BCD) format. Example: `X'19961231'` is Dec. 31, 1996.	4
H–YYDDD	YYDDD Julian date in hexadecimal (BCD) format. Example: `X'096366'` is Dec. 31, 1996.	3
H–YYYYDDD	YYYYDDD Julian date in hexadecimal (BCD) format. Example: `X'01996366'` is Dec. 31, 1996.	4
P–MMDDYY	MMDDYY date in packed format. Example: `X'0123196C'` is Dec. 31, 1996.	4
P–MMDDYYYY	MMDDYYYY date in packed format. Example: `X'012311996C'` is Dec. 31, 1996.	5
P–DDMMYY	DDMMYY date in packed format. Example: `X'0311296C'` is Dec. 31, 1996.	4
P–DDMMYYYY	DDMMYYYY date in packed format. Example: `X'031121996C'` is Dec. 31, 1996.	5
P–YYMMDD	YYMMDD date in packed format. Example: `X'0961231C'` is Dec. 31, 1996.	4
P–YYYYMMDD	YYYYMMDD date in packed format. Example: `X'019961231C'` is Dec. 31, 1996.	5

DATA TYPES FOR DATE FIELDS (CONTINUED)		
DATA TYPE	**DESCRIPTION** (See Note 1)	**LENGTH**
P–YYDDD	YYDDD Julian date in packed format. Example: `X'96366C'` is Dec. 31, 1996.	3
P–YYYYDDD	YYYYDDD Julian date in packed format. Example: `X'1996366C'` is Dec. 31, 1996.	4
P–CYYDDD	Packed Julian date with century digit (as used in SMF records). Example: `X'0096366C'` is Dec. 31, 1996. Example: `X'0196366C'` is Dec. 31, 2096.	4
STCKDATE	Spectrum Writer extracts the date portion of the date–time value stored by the IBM STCK machine instruction (CPU timer units since 00:00:00 1/1/1900 GMT). Spectrum Writer automatically converts the STCK value from GMT to local time. For more details, see the STCKADJ parm in the OPTIONS statement (page 575).	8
ABSDATE	Spectrum Writer extracts the date portion of a CICS ABSTIME date–time value (8-byte packed number of milliseconds since 00:00:00 1/1/1900).	8
DATEEXIT	Spectrum Writer will call a user–written exit program to obtain a date value. The exit program must return a 4–byte date in X'YYYYMMDD' format.	N/A

Notes:
(1) The CENTURY parm (in an OPTIONS statement) determines whether YY–type dates are 19YY or 20YY.
(2) Any non–numeric character is accepted as the delimiter character.

Time Data Types

DATA TYPES FOR TIME FIELDS			
DATA TYPE	**DESCRIPTION**	**DEFAULT LENGTH**	**LENGTH ALLOWED** (See Note 1)
HH–MM–SS	HH:MM:SS time in character format (with colons or other delimiters). (4) Decimal digits are allowed. Example: `C'12:34:56'` and `C'12.34.56'` are 12:34:56 Example: `C'12:34:56.7'` is 12:34:56.7 (6)	8	8–256 (2)
HHMMSS	HHMMSS time in character format (no delimiters). Decimal digits are allowed. Example: `C'123456'` is 12:34:56 Example: `C'1234567'` is 12:34:56.7 (6)	6	6–256 (2)
HH–MM	HH:MM time in character format (with a colon or other delimiter). (4) Decimal digits are *not* allowed. Example: `C'12:34'` and `C'12.34'` are 12:34	5	5
HHMM	HHMM time in character format. Decimal digits are *not* allowed. Example: `C'1234'` is 12:34	4	4

DATA TYPES FOR TIME FIELDS (CONTINUED)			
DATA TYPE	**DESCRIPTION**	**DEFAULT LENGTH**	**LENGTH ALLOWED** (See Note 1)
H–HHMMSS	HHMMSS time in hexadecimal (BCD) format. Decimal digits are allowed. Example: `X'123456'` is 12:34:56 Example: `X'01234567'` is 12:34:56.7 (6)	3	3–15 (2)
H–HHMM	HHMM in hexadecimal (BCD) format. Decimal digits are *not* allowed. Example: `X'1234'` is 12:34	2	2
P–HHMMSS	HHMMSS time in packed format. Decimal digits are allowed. Example: `X'0123456C'` is 12:34:56 Example: `X'1234567C'` is 12:34:56.7 (6)	4	4–16 (2)
P–HHMM	HHMM time in packed format. Decimal digits are *not* allowed. Example: `X'01234C'` is 12:34	3	3
SECS **SEC**	Seconds since midnight in character format. Decimal digits are allowed. Example: `C'45296'` is 12:34:56 (12*3600 + 34*60 + 56 = 45296.) Example: `C'452967'` is 12:34:56.7 (6)	N/A (5)	1–256 (3)
P–SECS	Seconds since midnight in packed format. Decimal digits are allowed. Example: `X'45296C'` is 12:34:56 Example: `X'0452967C'` is 12:34:56.7 (6)	N/A (5)	1–16 (3)
PU–SECS	Seconds since midnight in packed unsigned (BCD) format. Decimal digits are allowed. Example: `X'045296'` is 12:34:56 Example: `X'452967'` is 12:34:56.7 (6)	N/A (5)	1–16 (3)
B–SECS	Seconds since midnight in binary format. Decimal digits are allowed. Example: `X'0000B0F0'` is 12:34:56 (X'0000B0F0' = 45296 =12*3600 + 34*60 + 56.) Example: `X'0006E967'` is 12:34:56.7 (6)	N/A (5)	1–8 (3)
BU–SECS	Seconds since midnight in unsigned binary format. Decimal digits are allowed. Example: `X'B0F0'` is 12:34:56 Example: `X'0006E967'` is 12:34:56.7 (6)	N/A (5)	1–8 (3)
MINS	Minutes since midnight in character format. Decimal digits are allowed. Example: `C'120'` is 02:00:00 (2*60 =120) Example: `C'1205'` is 02:00:30.0 (6)	N/A (5)	1–256 (3)

DATA TYPES FOR TIME FIELDS (CONTINUED)			
DATA TYPE	**DESCRIPTION**	**DEFAULT LENGTH**	**LENGTH ALLOWED** (See Note 1)
P–MINS	Minutes since midnight in packed format. Decimal digits are allowed. Example: `X'120C'` is 02:00:00 Example: `X'01205C'` is 02:00:30.0 (6)	N/A (5)	1–16 (3)
PU–MINS	Minutes since midnight in packed unsigned (BCD) format. Decimal digits are allowed. Example: `X'0120'` is 02:00:00 Example: `X'1205'` is 02:00:30.0 (6)	N/A (5)	1–16 (3)
B–MINS	Minutes since midnight in binary format. Example: `X'0078'` is 02:00:00 (X'0078' = 120 = 2 * 60) Example: `X'04B5'` is 02:00:30.0 (6)	N/A (5)	1–8 (3)
BU–MINS	Minutes since midnight in binary unsigned format. Decimal digits are allowed. Example: `X'0078'` is 02:00:00 Example: `X'04B5'` is 02:00:30.0 (6)	N/A (5)	1–8 (3)
HOURS **HOUR** **HRS**	Hours since midnight in character format. Decimal digits are allowed. Example: `C'11'` is 11:00:00 Example: `C'1175'` is 11:45:00.00 (7)	N/A (5)	1–256 (3)
P–HOURS	Hours since midnight in packed format. Decimal digits are allowed. Example: `X'011C'` is 11:00:00 Example: `X'01175C'` is 11:45:00.00 (7)	N/A (5)	1–16 (3)
PU–HOURS	Hours since midnight in packed unsigned (BCD) format. Decimal digits are allowed. Example: `X'11'` is 11:00:00 Example: `X'1175'` is 11:45:00.00 (7)	N/A (5)	1–16 (3)
B–HOURS	Hours since midnight in binary format. Decimal digits are allowed. Example: `X'000B'` is 11:00:00 Example: `X'0497'` is 11:45:00.00 (7)	N/A (5)	1–8 (3)
BU–HOURS	Hours since midnight in binary unsigned format. Decimal digits are allowed. Example: `X'000B'` is 11:00:00 Example: `X'0497'` is 11:45:00.00 (7)	N/A (5)	1–8 (3)

DATA TYPES FOR TIME FIELDS (CONTINUED)			
DATA TYPE	**DESCRIPTION**	**DEFAULT LENGTH**	**LENGTH ALLOWED** (See Note 1)
STCKTIME	Spectrum Writer extracts the time portion of the date–time value stored by the IBM STCK machine instruction (CPU timer units since 00:00:00 1/1/1900 GMT). Spectrum Writer automatically converts the STCK value from GMT to local time. For more details, see the STCKADJ parm in the OPTIONS statement (page 575). STCKTIME fields always have 6 decimal digits.	8	8
ABSTIME	Spectrum Writer extracts the time portion of a CICS ABSTIME date–time value (8-byte packed number of milliseconds since 00:00:00 1/1/1900).	8	8
TIMEEXIT	Spectrum Writer will call a user–written exit program to obtain a time value. The exit program must return a 16-byte packed number of seconds since midnight (optionally including decimal digits).	N/A	N/A

Notes:
(1) Lengths refer to the number of bytes occupied in the input record.
(2) Field may contain no more than 15 numeric digits.
(3) Field may contain no more than 27 numeric digits.
(4) Any non–numeric character is accepted as the delimiter character.
(5) This data type has no default length. A LENGTH parm is always required for it in the FIELD statement.
(6) The FIELD statement would also need a DECIMAL(1) parm.
(7) The FIELD statement would also need a DECIMAL(2) parm.

Bit Data Types

DATA TYPES FOR BIT FIELDS		
DATA TYPE	**DESCRIPTION**	**LENGTH**
BIT	A single bit within a byte.	N/A
BITEXIT	Spectrum Writer will call a user–written exit program to obtain a bit value. The exit program must return either C'0' or C'1'.	N/A

Appendix B. Display Formats

Display formats can be used in various control statements to indicate how data should be formatted in a report or output file. When no display format is specified, Spectrum Writer formats data using a default display format (page 618). To override Spectrum Writer's default, use one of the display formats found in the following pages.

For example, you can specify a display format in the FORMAT parm of the FIELD statement:

```
FIELD: SOCIAL-SEC-NUM  TYPE(PACKED) LENGTH(5) FORMAT(PIC'999-99-9999')
```

The FIELD statement above includes a picture type of numeric display format. Specifying a FORMAT parm in the FIELD statement assigns a *default* format to use for that field whenever it appears in a report or output file.

You can also assign a default display format to computed fields in the COMPUTE statement:

```
COMPUTE:  PERCENT-GROWTH(PIC'ZZ9%') = (NEW - OLD) * 100 / OLD
```

You can also specify an *override* display format in any statement that describes a print line (such as the COLUMNS, TITLE, FOOTNOTE and BREAK statements.) For example:

```
COLUMNS:  HIRE-DATE(DD-MM-YYYY)
```

The above COLUMNS statement tells Spectrum Writer to format the HIRE–DATE field in "DD/MM/YYYY" format in the current report.

To change the default display format for all fields in a report, use the FORMAT parm of the OPTIONS statement:

```
OPTIONS: FORMAT(DOTSEP)
```

The above statement causes Spectrum Writer to format all numeric fields in the report with the DOTSEP format (e.g.: 1.234.567,89). You can still override any particular numeric field by using an override display format (for example, in the COLUMNS statement.)

For more information on exactly where and how to use a display format, see under the appropriate statement's description in Chapter 10, "Control Statement Syntax."

The display formats allowed for a particular field depend on the field's data type. For example, only *numeric* display formats may be used with *numeric* fields. You can not use a date or time display format with a numeric field.

The boxes on the following pages show the display formats available for each type of data.

> **Note:** There are no display formats for bit fields. A similar function is provided by the ONTEXT and OFFTEXT parms in the FIELD statement.

Default Display Formats

The following table shows Spectrum Writer's standard default display format for each type of data.

> **Note:** The default display formats are changed by certain OPTIONS statement options, including the FORMAT option, PC file options (such as EXCEL or LOTUS) and the MAINFRAME option.

DEFAULT DISPLAY FORMATS			
KIND OF DATA	DEFAULT DISPLAY FORMAT	DESCRIPTION	EXAMPLE
Character	CHARACTER	Data is displayed "as is," without any formatting	ABC
Numeric	NUMERIC	Leading zeros are suppressed; commas are used as separators; a floating negative sign precedes negative numbers.	-1,234.56
Date	MM–DD–YY	MM/DD/YY	12/31/96
Time	HH–MM–SS	HH:MM:SS (Decimal portions of seconds, if any, are also shown.)	13:45:59 17:30:00.12
Bit	none	There are no display formats for bit fields. Bit fields are always displayed using their ONTEXT or OFFTEXT value. See page 347.	FIELDNAME NOT FIELDNAME

Display Formats for Any Type of Field

DISPLAY FORMATS ALLOWED FOR ANY FIELD		
DISPLAY FORMAT	DESCRIPTION	EXAMPLE
CHARACTER **CHAR**	No formatting is done— data is printed "as is". This is normally used only for *character* fields, but is allowed for any type of field. This is the **default display format** for character fields.	ABC
QCHAR	The data is enclosed within quotation marks. Other than that, the data is not reformatted at all. This format is useful for formatting character fields for use in PC files. (You can use the QCHAR parm of the OPTIONS statement to choose a character other than the double quotation mark to use as the delimiter with this display format. See page 573.)	"ABC"
HEX	Each byte of data is expanded into two bytes to show the hexadecimal representation of the data. This format is useful when investigating fields that contain invalid data, such as hex zeros.	C1C2C3
BITS	Each byte of data is expanded into an 8-byte character string (of 0's and 1's) showing the individual bits within the data.	11000001

Numeric Display Formats

DISPLAY FORMATS FOR NUMERIC FIELDS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Formats Normally Used in Reports		
NUMERIC **NUM**	This is the **default display format** for all numeric fields, regardless of their data type. Formatting includes suppression of leading zeros and the use of commas as separators. A floating negative sign precedes negative numbers.	`1,234.56` `-1,234.56`

BARGRAPH
BAR
	A bar graph is printed. A number of asterisks equal to the rounded value of the numeric field will print (up to the total width of the column). Bar graphs are discussed on page 154.	`******`
		`********`
		`****`

DISPLAY **DISP**	Numbers are displayed without any punctuation (other than a decimal point, if necessary). Leading zeros are *not* suppressed. The "zone" portion of the last digit contains the sign.	`0001234.567` `0001234.56P`
DOLLAR	Same as NUMERIC, but a floating dollar sign will precede the first significant digit.	`$1,234.56` `-$1,234.56`

DOTSEP	Same as NUMERIC, but uses dots rather than commas as separators. Also uses a comma as the decimal indicator, rather than a dot. This format is widely used outside the USA.	`1.234,56`
		`-1.234,56`
		`12.345.678,9`

NOCOMMAS **NOCOMMA**	Same as NUMERIC, except that commas are not inserted among the digits. This format is useful for formatting numeric fields for use in PC files.	`1234.56` `-1234.56`
PICTURE'...' **PICT'...'** **PIC'...'** **P'...**	A "picture" is used to describe how the numeric value should be formatted. This is useful for formatting special purpose numbers, such as telephone numbers and social security numbers. The rules governing PICTUREs are given on page 451.	`(800) 555-1212` `123-45-6789`
Formats Normally Used in Output Files		
BINARY **BIN** **COMP**	Numbers are converted into binary representation (called COMP in COBOL, and FIXED BINARY in PL/I). The default width for data in BINARY format is 4 bytes.	`X'0001E240'` `X'FFFF1DC0'`
FULLWORD **FULL**	Same as BINARY, with an implied width of 4 bytes.	`X'0001E240'`
HALFWORD **HALF**	Same as BINARY, but with an implied width of 2 bytes.	`X'04D2'`

DISPLAY FORMATS FOR NUMERIC FIELDS (CONTINUED)		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
BINARYUN **BINUN** **BU**	Numbers are converted into an unsigned binary format (which has no equivalent in COBOL or in PL/I). It is similar to BINARY, except that the high order bit is not used as a sign, but as another binary digit. The default width for data in the BINARYUN format is 4 bytes. Negative numbers can *not* be formatted with this display format.	X'0001E240'
PACKED **PACK** **COMP–3**	Numbers are converted into packed decimal format (called COMP–3 in COBOL, and FIXED DECIMAL in PL/I). The default width for data in PACKED format is 8 bytes.	X'000000000123456C' X'000000000123456D'
PACKEDUN **PACKUN** **PU**	Numbers are converted into an unsigned packed decimal format, sometimes called BCD. (There is no equivalent in COBOL or in PL/I.) It is similar to PACKED, except that the last byte contains two numeric digits (like the other bytes), rather than a single digit and a sign. The default width for data in the PACKEDUN format is 8 bytes. Negative numbers can *not* be formatted with this display format.	X'0000000000123456'

Date Display Formats

DISPLAY FORMATS FOR DATE FIELDS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Formats Normally Used in Reports		
MM–DD–YY	MM/DD/YY This is the **default display format** for all date fields, regardless of their data type. (1)	12/31/96 12-31-96 12.31.96
MM–DD–YYYY	MM/DD/YYYY (1)	12/31/1996 12-31-1996 12.31.1996
MMDDYY	MMDDYY	123196
MMDDYYYY	MMDDYYYY	12311996
DD–MM–YY	DD/MM/YY (1)	31/12/96 31-12-96 31.12.96
DD–MM–YYYY	DD–MM–YYYY (1)	31/12/1996 31-12-1996 31.12.1996
DDMMYY	DDMMYY	311296
DDMMYYYY	DDMMYYYY	31121996
YYYY–MM–DD	YYYY-MM-DD (1)	1996/12/31 1996-12-31 1996.12.31

DISPLAY FORMATS FOR DATE FIELDS (CONTINUED)		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
YYMMDD	YYMMDD	961231
YYYYMMDD	YYYYMMDD	19961231
YYDDD	YYDDD (Julian date)	96366
YYYYDDD	YYYYDDD (Julian date)	1996366
SHORT1	MMM DD, YYYY	DEC 31, 1996
SHORT2	DD MMM YYYY	31 DEC 1996
SHORT3	DD MMM YY	31 DEC 96
LONG1	MMMMMMMMMMM DD, YYYY	DECEMBER 31, 1996
LONG2	DD MMMMMMMMMMM YYYY	31 DECEMBER 1996
LONG3	DD MMMMMMMMMMM YY	31 DECEMBER 96
Formats Normally Used in Output Files		
Q–MM–DD–YY	"MM/DD/YY" date in quotation marks. (1) (2)	"12/31/96"
Q–MM–DD–YYYY	"MM/DD/YYYY" date in quotation marks. (1) (2)	"12/31/1996"
Q–MMDDYYYY	"MMDDYYYY" date in quotation marks. (2)	"12311996"
Q–DD–MM–YYYY	"DD/MM/YYYY" date in quotation marks. (1) (2)	"31/12/1996"
Q–DDMMYYYY	"DDMMYYYY" date in quotation marks. (2)	"31121996"
Q–YYMMDD	"YYMMDD" date in quotation marks. (2)	"961231"
Q–YYYY–MM–DD	"YYYY/MM/DD" date in quotation marks. (1) (2)	"1996/12/31"
Q–YYYYMMDD	"YYYYMMDD" date in quotation marks. (2)	"19961231"
H–MMDDYY	MMDDYY (hex)	X'123196'
H–MMDDYYYY	MMDDYYYY (hex)	X'12311996'
H–DDMMYY	DDMMYY (hex)	X'311296'
H–DDMMYYYY	DDMMYYYY (hex)	X'31121996'
H–YYMMDD	YYMMDD (hex)	X'961231'
H–YYYYMMDD	YYYYMMDD (hex)	X'19961231'
H–YYDDD	YYDDD (hex, Julian date)	X'096366'
H–YYYYDDD	YYYYDDD (hex, Julian date)	X'01996366'
P–MMDDYY	MMDDYY (packed)	X'0123196C'
P–MMDDYYYY	MMDDYYYY (packed)	X'012311996C'
P–DDMMYY	DDMMYY (packed)	X'0311296C'
P–DDMMYYYY	DDMMYYYY (packed)	X'031121996C'
P–YYMMDD	YYMMDD (packed)	X'0961231C'

DISPLAY FORMATS FOR DATE FIELDS (CONTINUED)		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
P–YYYYMMDD	YYYYMMDD (packed)	X'019961231C'
P–YYDDD	YYDDD (packed, Julian date)	X'96366C'
P–YYYYDDD	YYYYDDD (packed, Julian date)	X'1996366C'
P–CYYDDD	Packed CYYDDD date (Julian date with century indicator, as used in SMF records)	X'096366C' X'196366C'
Notes:: (1) Use the DATEDELIM parm in the OPTIONS statement (page 560) to specify a delimiter other than the slash (/). (2) Use the QCHAR parm in the OPTIONS statement (page 573) to specify a delimiter other than the double quotation mark (").		

Time Display Formats

DISPLAY FORMATS FOR TIME FIELDS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Formats Normally Used in Reports		
HH–MM–SS	HH:MM:SS[.NNN...] This is the **default display format** for all time fields that include seconds. (1)	13:30:45 13:30:45.5 13.30.45
HH–MM–SS–AMPM	HH:MM:SS[.NNN...] AM/PM (1) The time is shown in 12-hour format including either AM or PM	1:30:45 AM 1:30:45.5 AM 1.30.45 PM
HHMMSS	HHMMSS	133045
HH–MM	HH:MM This is the **default display format** for all time fields that do not include seconds. (1)	13:31 13.31
HH–MM–AMPM	HH:MM AM/PM (1) The time is shown in 12-hour format including either AM or PM. Seconds are not shown and the time is rounded to the nearest minutes.	1:31 AM 1.31 PM
HHMM	HHMM	1331
TPICTURE'...' **TPICT'...'** **TPIC'...'** **TP'...'**	User defined "time picture." (Time pictures are discussed on page 458.) Example: TPIC'Z9–99–99' might result in " 8–25–59".	8-25-59
SECS **SEC**	Number of seconds since midnight. (13 hours, 30 minutes and 45 seconds is 48,645 seconds.)	48,645
MINS	Number of minutes since midnight. (13 hours, 30 minutes and 45 seconds is 810.75 minutes.)	810.75
HOURS **HOUR** **HRS**	Number of hours since midnight. (13 hours, 30 minutes and 45 seconds is 13.513 hours.)	13.513

DISPLAY FORMATS FOR TIME FIELDS (CONTINUED)		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Formats Normally Used in Output Files		
Q–HH–MM–SS	"HH:MM:SS" time in double quotation marks. This format is useful for formatting time fields for use in PC files. (1) (2)	`"13:30:45"`
Q–HH–MM	"HH:MM" time in double quotation marks. (1) (2)	`"13:31"`
H–HHMMSS	HHMMSS (hex)	`X'133045'`
H–HHMM	HHMM (hex)	`X'1331'`
P–HHMMSS	HHMMSS (packed)	`X'0133045C'`
P–HHMM	HHMM (packed)	`X'01331C'`
SECS–NC **SEC–NC**	Number of seconds since midnight, formatted with "no commas" (for use in PC files).	`48645`
MINS–NC	Number of minutes since midnight, formatted with "no commas" (for use in PC files).	`810.75`
HOURS–NC **HOUR–NC** **HRS–NC**	Number of hours since midnight, formatted with "no commas" (for use in PC files).	`13.513`

Notes:

(1) Use the TIMEDELIM parm in the OPTIONS statement (page 576) to specify a delimiter other than the colon (:).

(2) Use the QCHAR parm in the OPTIONS statement (page 573) to specify a delimiter other than the double quotation mark (").

Appendix C. Built-In Fields

Spectrum Writer has a number of "built–in" fields that are available for use. You may refer to these fields regardless of what input file(s) you use. Built–in fields are easily distinguished from most other fields because all built–in field names begin with the pound character (#).

The following table lists the Spectrum Writer built–in fields. Following the table, each field is discussed in more detail.

SPECTRUM WRITER BUILT-IN FIELDS	
FIELD NAME	**DESCRIPTION**
Character Built-In Fields	
#DAYNAME	Name of the current day of the week ("MONDAY")
#ITEM–ENDING	The correct plural or singular ending for the word "item(s)" at a control break. (Allowed only in the BREAK statement.)
#JOBNAME	Jobname under which Spectrum Writer is currently executing.
#TIME	Character field containing the formatted system time (when program began execution). Format uses AM or PM ("12:45 PM").
#TIME24	Character field containing the formatted system time (when program began execution). Formatted in 24–hour format ("13:45").
Numeric Built-In Fields	
#COUNTER **#COUNT**	The cumulative number of items in the report. (Allowed only in the BREAK statement.)
#ITEMS **#ITEM**	The number of items in the current control group. (Allowed only in the BREAK statement.)
#ITEM1 **through** **#ITEM9**	The item number currently being printed. The 9 different built–in fields are reset at 9 different levels of control breaks. (Allowed only in the COLUMNS statement.)
#PAGENUM **#PAGE**	The current page number of the report. (Allowed only in the TITLE and FOOTNOTE statements.)
Date Built-In Fields	
#COMDATE	(VSE only) The date set by the // DATE JCL statement.
#TODAY	The system date (at the time program began execution).

SPECTRUM WRITER BUILT-IN FIELDS (CONTINUED)	
FIELD NAME	**DESCRIPTION**
Time Built-In Fields	
#HHMMSS	The system time (when program began execution).

Character Built–In Fields

#DAYNAME

Allowed in any control statement. A 9–byte field containing the name of the day of the week in which the program began execution. The value of this built–in field does not change during the run. The use of this field is discussed on page 163.

Sample value: `WEDNESDAY`

#ITEM–ENDING

Allowed only in the BREAK statement. A 1–byte character field that contains either the letter "S" or a blank, depending on the value of the built–in field #ITEMS.

When #ITEMS is equal to 1 (that is, when the current control group contains only a single record), #ITEM–ENDING will contain a blank space. Otherwise (when the control group contains more than one record) #ITEM–ENDING will contain an "S". Append this field to words like "ITEM" to form the proper plural or singular ending. (For example, "1 ITEM " versus "2 ITEMS".) The use of this field is discussed on page 198.

#JOBNAME

Allowed in any control statement. An 8–byte character field containing the name of the job that is executing Spectrum Writer.

#TIME

Allowed in any control statement. An 8–byte character field containing the system time (when the program began execution). The value of this built–in field does not change during the run. The time is in 12–hour format and includes either AM or PM. The use of this field is discussed on page 163.

Sample value: `12:31 PM`

#TIME24

Allowed in any control statement. A 5–byte character field containing the system time (when the program began execution). The value of this built–in field does not change during the run. The time is in 24 hour format. The use of this field is discussed on page 198.

Sample value: `14:55`

Numeric Built-In Fields

#COUNTER
#COUNT

Allowed only in the BREAK statement. A numeric field that contains the number of items processed in the report through the current break. Similar to #ITEMS but is not reset to zero at each control break. By default it displays with a ZZZ,ZZ9 picture format. The use of this field is discussed on page 198.

#ITEMS
#ITEM

Allowed only in the BREAK statement. A numeric field that contains the number of items in the control group being processed. By default it displays with a ZZZ,ZZ9 picture format. The use of this field is discussed on page 198.

#ITEM/#ITEM1/#ITEM2/#ITEM3/#ITEM4/
#ITEM5/#ITEM6/#ITEM7/#ITEM8/#ITEM9

Allowed only in the COLUMNS statement. These nine built–in fields all show the item number (within a given level of control group) of the line currently being printed. #ITEM1 contains the item number within the lowest level control group. #ITEM1 is reset to zero at every control break. (#ITEM1 can also be abbreviated #ITEM.) #ITEM2 contains the item number within the second lowest level control group. #ITEM2 is not reset to zero at the lowest level control break, but is reset at the second lowest level control break. #ITEM3 through #ITEM9 work similarly for the third through ninth lowest level control breaks. All are numeric fields which display by default with a ZZ,ZZ9 picture format. The use of these fields is discussed on page 211. For a discussion of "control group levels", see page 204.

#PAGENUM
#PAGE

Allowed only in TITLE and FOOTNOTE statements. A numeric field containing the current page number. By default, it displays with a ZZZ9 picture format. The use of this field is discussed on page 163.

Date Built-In Fields

#COMDATE

Allowed in any control statement— VSE only. Contains the "comm area" date. This is the date set by the // DATE JCL statement. If not set in the JCL, or if used under OS/390, #COMDATE will be the same as #TODAY. By default, it is formatted using the default date display format that is in effect (normally MM–DD–YY).

Sample value: `12/01/99`

#TODAY

Allowed in any control statement. Contains the system date on which the job began. The value of this built–in field does not change during execution. By default, it is formatted using the run's default date display format (usually MM–DD–YY). The use of this field is discussed on page 163.

Sample value: `12/01/99`

Time Built-In Fields

#HHMMSS

Allowed in any control statement. Contains the system time (when the program began execution). The value of this built–in field does not change during the run. By default, it is formatted using the run's default time display format (normally HH-MM-SS).

Sample value: `12:34:56`

Appendix D. Built-In Functions

A number of built–in functions are available for use within computational expressions. Computational expressions are used in COMPUTE statements. These built–in functions are listed on the following pages, according to the type of data *returned* by the function (character, numeric, date, time or boolean).

The arguments to a function will not necessarily be of the same data type as the result. The data type expected for each argument is indicated in a function's syntax. For example, "char" means that a character argument is expected. Except where otherwise indicated, an argument may be any of the following:

- a literal value
- the name of a field from any input file
- the name of a computed field (from any previous COMPUTE statement)
- a computational expression (which may itself involve other built–in functions)

Separate the arguments with blanks and/or commas.

The following table lists the Spectrum Writer built–in functions. After the table, each of the functions is discussed in more detail.

SPECTRUM WRITER BUILT-IN FUNCTIONS		
FUNCTION	**DESCRIPTION**	**PAGE**
Functions that Return a Character Value		
#AND	returns the result of ANDing two character strings	page 630
#ASCII	returns the ASCII equivalent of an EBCDIC string	page 631
#COMPRESS	concatenates multiple fields and compresses out extra blanks	page 631
#DAY	returns the day of the week for a given date	page 631
#EBCDIC	returns the EBCDIC equivalent of an ASCII string	page 631
#FORMAT	converts a numeric, date or time value to a character value	page 632
#LCASE	returns the lower–case value of a character string	page 632
#LEFT	returns the leftmost *n* characters of a character string	page 632
#MONTH	returns the month name pertaining to a given date	page 633
#OR	returns the result of ORing two character strings	page 633
#PARSE	returns one individual word parsed out of a character string	page 633
#RIGHT	returns the rightmost *n* characters of a character string	page 634
#SUBSTR	returns a substring from a character string	page 634

SPECTRUM WRITER BUILT-IN FUNCTIONS (CONTINUED)		
FUNCTION	**DESCRIPTION**	**PAGE**
#TRANSLATE	translates one set of characters within a character string to another set of characters	page 634
#UCASE	returns the upper–case value of a character string	page 634
#XOR	returns the result of XORing two character strings	page 635
#YEAR	returns the 4–byte year pertaining to a given date	page 635
Functions that Return a Numeric Value		
#ABS	returns the absolute value of a number	page 635
#DAYNUM	returns the day of the month (1–31) for a given date	page 635
#DOWNUM	returns a number representing the day of the week of a given date	page 635
#HOURNUM	returns the numeric value of the hours portion of a time	page 635
#INDEX	returns the starting column of a substring	page 635
#INT	returns the integer portion of a number	page 635
#MAKENUM	converts a character, date or time value to a numeric value	page 636
#MAX	returns the greater of two or more values	page 637
#MIN	returns the smaller of two or more values	page 637
#MINUTENUM	returns the numeric value of the minutes portion of a time	page 637
#MOD	returns the remainder left after division ("modulus")	page 637
#MONTHNUM	returns the month number (1–12) for a given date	page 637
#NUMWORDS	returns the number of words within a character string	page 638
#ROUND	returns the rounded value of a number	page 638
#SECONDNUM	returns the numeric value of the seconds portion of a time	page 638
#YEARNUM	returns the 4–digit year for a given date	page 638
Functions that Return a Date Value		
#BEGMONTH	returns the first day of the month in which a date occurs	page 638
#BEGWEEK	returns the first day of the week in which a date occurs	page 638
#BEGYEAR	returns the first day of the year in which a date occurs	page 639
#ENDMONTH	returns the last day of the month in which a date occurs	page 639
#ENDWEEK	returns the last day of the week in which a date occurs	page 639
#ENDYEAR	returns the last day of the year in which a date occurs	page 639
#INCDATE	increments a date by a number of days, weeks, months or years	page 639
#INCDATETIME	increments a date/time by a number of seconds, minutes or hours	page 639

SPECTRUM WRITER BUILT-IN FUNCTIONS (CONTINUED)		
FUNCTION	**DESCRIPTION**	**PAGE**
#MAKEDATE	converts a character or numeric value to a date	page 640
#YMD **#MDY** **#DMY**	creates a date from three numeric parms	page 640
Functions that Return a Time Value		
#INCDURATION	increments a time duration by a number of seconds, minutes or hours	page 640
#INCTIME	increments a time of day by a number of seconds, minutes or hours	page 641
#MAKETIME	converts a character or numeric value to a time	page 641
Functions that Return a Boolean Value		
#ERROR	returns "true" if the argument field is "in error"	page 642
#ISNUM	returns "true" if the character argument is numeric	page 642
#LEAPYEAR	returns "true" if the date argument occurs in a leap year	page 642
#MISSING	returns "true" if the argument field is "missing"	page 642
#OFF	returns "false"	page 642
#ON	returns "true"	page 643
#REALDATE	returns "true" if the date argument is a valid, calendar date	page 643

Functions that Return a Character Value

#AND(char1,char2)

Performs the logical AND operation on the two character arguments and returns the result. (An AND operation results in a 1 bit if the corresponding bit of *both* operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right-padded with hex zeros before performing the AND operation. The size of the result is the size of the larger operand.

```
Example: COMPUTE: A = #AND(X'01FF',X'035E')   results in   A=X'015E'
```

Here is an example of using the #AND built–in function to test individual bits within a status flag. Say that we want to include records in our report if the X'80' and the X'20' bits of the STATUS field are both on, regardless of the value of the other bits in that byte.

```
Example: COMPUTE: TEMP = #AND(STATUS, X'A0')
         COMPUTE: BOTH-BITS-ARE-ON = WHEN(TEMP = X'A0') ASSIGN(#ON)
         INCLUDEIF: BOTH-BITS-ARE-ON
```

Note: You can use the #AND function to change a packed numeric field's sign from the common, but non-standard, F to the standard C. For example, assume that

PACKED-NUMBER is a 5-byte packed field that has an F in the zone portion of its last byte (X'000000123F')

```
Example: COMPUTE: PACKED-C = #AND(PACKED-NUMBER,X'FFFFFFFFFC'
                  results in PACKED-C = X'000000123C'
```

#ASCII(char)

Returns the ASCII equivalent of the EBCDIC character argument. The size of the value returned by this function is the size of the character argument.

```
Example: COMPUTE: A = #ASCII(X'F1F2F3')  results in  A=X'313233'
```

Note: The three characters "123" are represented with X'F1F2F3' in EBCDIC and with X'313233' in ASCII.

Note: If you want to specify your own EBCDIC-to-ASCII translation table, use the ASCIITABLE option in the OPTIONS statement (page 558). Otherwise, Spectrum Writer uses a default translation table.

Note: For information on creating ASCII output files, see page 143.

#COMPRESS([n,] char [,n] ,char ...)

Concatenates the char arguments (any number), but compresses out all but 1 of the blanks between each argument The optional override number "n" specifies how many blanks to leave between the two char arguments (if a number other than 1 is desired). You may specify 0 if no blanks are wanted between two arguments. The size of the returned string is the sum of the sizes of all arguments, plus spacing bytes.

```
Example: COMPUTE: NAME=#COMPRESS(LAST-NAME, 0, "," , FIRST-NAME)
                  might result in NAME="BAKER, VIVIAN                "

         COMPUTE: ADDR=#COMPRESS(CITY, 0, ",", STATE ZIP-CODE)
                  might result in ADDR="DALLAS, TX 75230              "
```

Note: The #COMPRESS function does not remove any *leading* blanks that might be in the character arguments. If your arguments could contain leading (as well as trailing) blanks, you should first left-justify those arguments to remove the leading blanks. Like this:

```
Example: COMPUTE: LJ-LAST-NAME  = #FORMAT(LAST-NAME, LEFT)
         COMPUTE: LJ-FIRST-NAME = #FORMAT(FIRST-NAME, LEFT)
         COMPUTE: NAME = #COMPRESS(LJ-LAST-NAME, 0 "," LJ-FIRST-NAME)
```

#DAY[(date)]

Returns the day of the week pertaining to the date argument, as a 9–byte character field. If specified, the date argument must be a valid date in either the twentieth or twenty-first century. If no date argument is present, the system date is used.

```
Example: COMPUTE: A = #DAY(HIRE-DATE)  might result in  A="TUESDAY  "
```

#EBCDIC(char)

Returns the EBCDIC equivalent of the ASCII character argument. The size of the value returned by this function is the size of the character argument.

```
Example: COMPUTE: A = #EBCDIC(X'313233')  results in  A=X'F1F2F3'
```

Note: The three characters "123" are represented with `X'F1F2F3'` in EBCDIC and with `X'313233'` in ASCII.

Note: If you want to specify your own ASCII-to-EBCDIC translation table, use the EBCDICTABLE option in the OPTIONS statement (page 562). Otherwise, Spectrum Writer uses a default translation table.

#FORMAT(fieldname [,display–format] [,width] [,BIZ] [,LEFT/CENTER/RIGHT] [ASCII])

Returns a character string containing the contents of the field (any data type) after formatting it as specified by the other parms. Only fieldname is required. It must be the first argument. It must also be the name of an actual field-- expressions are not allowed for this argument. The other parms may appear in any combination and in any order. The display format, if specified, must be valid for the specified field's data type. For an explanation of each of the parms, see the syntax of the COLUMNS statement, which uses the same parms (page 498).

Example: `COMPUTE: A = #FORMAT(#TODAY)` might result in `A='03/31/07`

`COMPUTE: A = #FORMAT(#TODAY, LONG1, CENTER)`
might result in `A='  MARCH 31, 2007  '`

`COMPUTE: A = #FORMAT(SALES-DATE, BIZ)`
results in `A='03/31/07'` when `SALES-DATE = 03/31/2007`
or `A='        '` (when `SALES-DATE = 00/00/0000`

`COMPUTE: A = #FORMAT(TOTAL-SALES)` might result in `A='      92,125.89'`

`COMPUTE: A = #FORMAT(TOTAL-SALES,10)` might result in `A=' 92,125.89'`

`COMPUTE: A = #FORMAT(TELEPHONE, PIC'(999) 999-9999'))`
might result in `A='(415) 555-1209'`

`COMPUTE: A = #FORMAT(TOTAL-SALES, BIZ)`
results in `A='      92,125.89'` when `TOTAL-SALES = 92,125.89`
or `A='               '` when `TOTAL-SALES = 0`

#LCASE(char)

Returns the character argument's value after translating any of its upper–case alphabetic characters to the corresponding lower–case character. All other printable characters remain unchanged. (The effect of this function on non-printable characters is not defined.) The size of the value returned by this function is the size of the character argument.

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)
`COMPUTE: A = #LCASE(DESC)` results in `A="this is a description".`

#LEFT(char,num1)

Returns a substring of the char argument, starting with the first column, for a length of "num1" bytes. Num1 may be either a literal value or a numeric expression. When num1 is a literal value, the size of the value returned by this function is num1. When num1 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

Example: `COMPUTE: A = #LEFT('ABCDEFG',4)` results in `A='ABCD'`

#MONTH[(date)]

Returns the name of the month pertaining to the date argument, as a 9–byte character field. If no date argument is present, the system date is used.

```
Example: COMPUTE: A = #MONTH(HIRE-DATE)   might result in   A="MARCH     "
```

#OR(char1,char2)

Performs the logical OR operation on the two character arguments and returns the result. (An OR operation results in a 1 bit if the corresponding bit of *either (or both)* operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right–padded with hex zeros before performing the OR operation. The size of the result is the size of the larger operand.

```
Example: COMPUTE: A = #OR(X'8024',X'0756')  results in  A=X'8776'
```

Note: You can use the #OR function to create packed numeric fields that have a sign of F (rather than the standard sign of C). For example, assume that AMOUNT has a value of 123 (in any format).

```
Example: COMPUTE: PACKED   = #FORMAT(AMOUNT,PACKED,2)
         COMPUTE: PACKED-F = #OR(PACKED,X'000F')  results in  PACKED = X'123C'  and
                                                              PACKED-F = X'123F'
```

#PARSE(char,num)

Returns a single word parsed from the character argument. Internally, the character argument is first parsed into individual words, each delimited by one or more spaces. The numeric argument specifies which of the parsed words should be returned by the function. A numeric argument of 1 indicates that the first word should be returned; an argument of 2 means return the second word, etc. Negative numbers may also be used. A negative number indicates the word to return counting backwards from the last word parsed. A numeric argument of –1 means return the last word parsed; an argument of –2 means return the second to last word, etc. If the word indicated by the numeric argument doesn't exist, blanks are returned by this function. The size of the value returned by this function is the size of the character argument.

Note: You can use the related #NUMWORDS built–in function to find out how many words a character string contains.

```
Example: (Assume that DESC = "THIS IS A DESCRIPTION")
         COMPUTE: A = #PARSE(DESC,1)  results in  A="THIS            "
         COMPUTE: A = #PARSE(DESC,2)  results in  A="IS              "
         COMPUTE: A = #PARSE(DESC,-1) results in  A="DESCRIPTION     "
         COMPUTE: A = #PARSE(DESC,5)  results in  A="                "
```

Note: To parse a text using a delimiter other than blanks, try using the #TRANSLATE built-in function to first translate the desired delimiter characters into blanks. For example, you could parse an IP address (which is delimited with dots) this way:

```
Example: COMPUTE: A = #PARSE(#TRANSLATE(IPADDR,"."," "),2)
```

Assume that `IPADDR = "12.345.67.8"`. The above statement results in `A = "345         "`

Note that using #TRANSLATE with #PARSE may not work if the original string contains *multiple consecutive* delimiters.

#RIGHT(char,num1)

Returns a substring of the char argument consisting of the last "num1" bytes. Num1 may be either a literal value or a numeric expression. When num1 is a literal value, the size of the value returned by this function is num1. When num1 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

```
Example: COMPUTE: A = #RIGHT('ABCDEFG',4)  results in  A='DEFG'
```

#SUBSTR(char,num1,num2)

Returns a substring of the char argument, starting at column "num1" for a length of "num2" bytes. Num1 and num2 may be literal values or numeric expressions. When num2 is a literal value, the size of the value returned by this function is num2. When num2 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

```
Example: COMPUTE: A = #SUBSTR('ABCDEFG',2,3)  results in  A='BCD'
```

#TRANSLATE(char1,char2,char3)

Returns the char1 string after translating any of its characters found in the char2 argument into the corresponding character of the char3 argument. (Translated characters in the char1 argument are *not* then re–evaluated for additional translation.) The size of the value returned by this function is the size of the char1 argument.

```
Example:  (Assume that DESC = "THIS IS A DESCRIPTION")
          COMPUTE: A = #TRANSLATE(DESC,"TA","XY")  would result in
                                                   A="XHIS IS Y DESCRIPXION".
```

Note: Normally the char2 and char3 arguments are character or hex literals. However, character fields may also be used for those arguments. If character fields are used, their contents will be examined only once by Spectrum Writer. This occurs the first time the results of the #TRANSLATE function are actually required during the run. (This may or may not correspond to the first input record.) After that, subsequent executions of the #TRANSLATE function do not re–examine the contents of the char2 and char3 fields. The contents of those arguments from their first examination is used for the entire run.

#UCASE(char)

Returns the character argument's value after translating any of its lower–case alphabetic characters to the corresponding upper–case character. All other printable characters remain unchanged. (The effect of this function on non-printable characters is not defined.) The size of the value returned by this function is the size of the character argument.

```
Example:  (Assume that NAME = "Smith     ")
          COMPUTE: SORT-NAME = #UCASE(NAME)  results in  SORT-NAME = "SMITH     "
```

Note: This function may be useful when sorting a report on a field that contains mixed–case text. For example, in order to ensure that the names "SMITH", "Smith", and "smith" all sort together, you could compute a new field that contains the upper–case value of the mixed–case name field. By sorting on this new upper–case field, rather than the original mixed–case field, the three names above would sort together. Of course, you can still choose to *print* the original, mixed–case names in your report, even though sorting on the upper–case names.

#XOR(char1,char2)

Performs the logical XOR operation on the two character arguments and returns the result. (An XOR operation results in a 1 bit if the corresponding bit of either (but *not* both) operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right-padded with hex zeros before performing the XOR operation. The size of the result is the size of the larger operand.

Example: `COMPUTE: A = #XOR(X'5766',X'5744')` results in `A=X'0022'`

#YEAR[(date)]

Returns the year portion of the date argument as a 4–byte character field. Note 2

Example: `COMPUTE: A = #YEAR(HIRE-DATE)` might result in `A="2001"`

Functions that Return a Numeric Value

#ABS(num)

Returns the absolute value of the numeric argument.

Example: `COMPUTE: A = #ABS(-4)` results in `A= 4`

#DAYNUM[(date)]

Returns the numeric value of the day portion of the date argument. Note 2

Example: `COMPUTE: A = #DAYNUM(3/31/2007)` results in `A=31`

#DOWNUM[(date)]

Returns a number from 1 to 7 representing the day of the week of the argument date. (1 means Sunday, 2 means Monday, ... 7 means Saturday.) Notes 2, 3

Example: `COMPUTE: A = #DOWNUM(3/31/2007)` results in `A=7`

#HOURNUM[(time)]

Returns the numeric value of the hours portion of the time argument. Note 1

Example: `COMPUTE: A = #HOURNUM(12:30:59)` results in `A=12`

#INDEX(char1,char2)

If the second argument appears somewhere within the first argument, #INDEX returns the byte number in char1 where the char2 text begins. If char1 does not contain char2, #INDEX returns zero.

Example: `COMPUTE: A = #INDEX('ABCDEF', 'CDE')` results in `A=3`

#INT(num)

Returns the integer portion of the numeric argument. The argument's decimal digits, if any, are simply dropped, regardless of the sign of the argument.

Example: `COMPUTE: A = #INT(12.345)` results in `A= 12`
`COMPUTE: A = #INT(-12.345)` results in `A= -12`

#MAKENUM(char/date/time)

For **character arguments**, converts the string of numeric characters into a numeric value. No decimal point, commas, or any other non–numeric character is allowed in the string. The only exception is that leading blanks are allowed. An all–blank string returns the value zero.

```
Example: COMPUTE: A = #MAKENUM('  125')  results in  A=125
```

Note: You can use the #ISNUM function to determine whether a given character text can successfully be converted into a number by #MAKENUM.

```
Example: COMPUTE X-IS-NUMERIC = #ISNUM(X)
         COMPUTE: NUMERIC-X = WHEN(X-IS-NUMERIC) ASSIGN(#MAKENUM(X))
                              ELSE               ASSIGN(99999999)
```

The above example results in NUMERIC-X being equal to the numeric value in the character field X, or being 99999999 when the character field X does not contain a valid numeric value.

For **date arguments**, #MAKENUM converts the date into a numeric "day in century" value. January 1, 1900 corresponds to day 1, and December 31, 2099 is day 73,049. The date argument must be a valid date in either the 20th or 21st century. (You can use the #MAKEDATE function to convert a numeric day in century back into a date.)

```
Example: COMPUTE: A = #MAKENUM(12/31/2007)  results in  A=39446
         COMPUTE: A = #MAKENUM(1/1/2008)    results in  A=39447
```

```
Example: (of computing the number of days between two dates):
         COMPUTE: NUM-DAYS = #MAKENUM(END-DATE) - #MAKENUM(START-DATE)
```

If END-DATE = 4/2/2007 and START-DATE = 3/28/2007, then the above example would result in NUM-DAYS = 5.

For **time arguments**, #MAKENUM converts the time value into its equivalent number of seconds since midnight.

```
Example: COMPUTE: A = #MAKENUM(01:29:59)  results in  A=5399
```

(One hour = 3600 seconds; 29 minutes is another 1740 seconds, plus 59 seconds equals 5399.)

```
Example: (of computing the number of seconds between two times):
         COMPUTE: NUM-SECS = #MAKENUM(END-TIME) - #MAKENUM(START-TIME)
```

If END-TIME = 13:05:07 and START-TIME = 13:04:56, then the above example would result in NUM-SECS = 11.

If the start and end times might occur on different days, you can convert the starting and ending *dates* into seconds as well, and use those in the computation. (There are 86400 seconds in a 24–hour day).

```
Example: COMPUTE: NUM-SECS = ((#MAKENUM(END-DATE) * 86400) + #MAKENUM(END-TIME))
                           - ((#MAKENUM(START-DATE) * 86400) + #MAKENUM(START-TIME))
```

To convert the resulting interval (in seconds) back into a time field, you would use this statement:

```
Example: COMPUTE: DURATION = #MAKETIME(NUM-SECS)
```

If `NUM-SECS = 11` then the above example would result in `DURATION = 00:00:11`.

#MAX(num1,num2,num3,...)

Returns the largest of the numeric arguments. Any number of arguments is allowed.

```
Example: COMPUTE: A = #MAX(12, 25, -3)  results in  A=25
```

You can also use this function to determine the largest of several **time fields**. First convert the times to numeric values for use with #MAX. Then convert the result back to a time:

```
Example: COMPUTE: LAST-TIME = #MAKETIME(#MAX(#MAKENUM(TIME1), #MAKENUM(TIME2)))
```

You can also use this function to determine the largest (latest) of several **date fields**. First convert the dates to numeric values for use with #MAX. Then convert the result back to a date:

```
Example: COMPUTE: LAST-DATE = #MAKEDATE(#MAX(#MAKENUM(DATE1), #MAKENUM(DATE2)))
```

#MIN(num1,num2,num3,...)

Returns the smallest of the numeric arguments. Any number of arguments is allowed.

```
Example: COMPUTE: A = #MIN(12, 25, -3)  results in  A=-3
```

You can also use this function to determine the smallest of several **time fields**. First convert the times to numeric values for use with #MIN. Then convert the result back to a time:

```
Example: COMPUTE: FIRST-TIME = #MAKETIME(#MIN(#MAKENUM(TIME1), #MAKENUM(TIME2)))
```

You can also use this function to determine the smallest (earliest) of several **date fields**. First convert the dates to numeric values for use with #MIN. Then convert the result back to a date:

```
Example: COMPUTE: FIRST-DATE = #MAKEDATE(#MIN(#MAKENUM(DATE1), #MAKENUM(DATE2)))
```

#MINUTENUM[(time)]

Returns the numeric value of the minutes portion of the time argument. (May also be abbreviated #MINNUM.) Note 1

```
Example: COMPUTE: A = #MINUTENUM(12:30:59)  results in  A=30
```

#MOD(num1,num2)

Returns the remainder left when the first argument is divided by the second argument.

```
Example: COMPUTE: A = #MOD(45, 4)   results in  A=  1
         COMPUTE: A = #MOD(-45, 4)  results in  A= -1
         COMPUTE: A = #MOD(1.5, .2) results in  A= .1
```

#MONTHNUM[(date)]

Returns the numeric value of the month portion of the date argument. If no date argument is present, the system date is used.

```
Example: COMPUTE: A = #MONTHNUM(3/31/2007)  results in  A=3
```

#NUMWORDS(char)

Returns the number of words found within the character argument. The words are parsed in the manner described under the #PARSE built–in function (page 633).

```
Example: (Assume that DESC = "THIS IS A DESCRIPTION")
         COMPUTE: A = #NUMWORDS(DESC)  results in  A = 4.
```

Note: This function may be useful when you want to assign a value to a computed field differently depending on how many, if any, words are in some other field. For example, the following example assigns the second word from the DESC field to the result. However, if the DESC field contains only 1 (or no) words, the text "*NONE*" is assigned instead:

```
Example: COMPUTE: A = WHEN(#NUMWORDS(DESC) >= 2) ASSIGN(#PARSE(DESC,2))
                      ELSE                       ASSIGN("*NONE*")
```

#ROUND(num1,num2)

Returns the first numeric argument, rounded to the precision specified by the second numeric argument. Num2 is the number of decimal places that num1 should be rounded to. Rounding of negative numbers is performed as if they were positive. Num2 must be a literal integer (not an expression). The *number* of decimal digits returned by this function is the same as the number of decimal digits in the num1 argument.

Num2 can also be a **negative** number. Use this feature to round to a digit position on the *left* side of the decimal point. Use -1 to round to tens, -2 to round to hundreds, and so on.

```
Example: COMPUTE: A = #ROUND(12345.678, 2)  results in  A= 12345.680
         COMPUTE: A = #ROUND(-12345.678, 2) results in  A=-12345.680
         COMPUTE: A = #ROUND(12345.678, 0)  results in  A= 12346.000
         COMPUTE: A = #ROUND(12345.678, -2) results in  A= 12300.000
```

#SECONDNUM[(time)]

Returns the numeric value of the seconds portion of the time argument. (May also be abbreviated #SECNUM.) Note 1

```
Example: COMPUTE: A = #SECONDNUM(12:30:59)  results in  A=59
```

#YEARNUM[(date)]

Returns the 4-digit numeric value of the year portion of the date argument. If no date argument is present, the system date is used.

```
Example: COMPUTE: A = #YEARNUM(3/31/07)  results in  A=2007
```

Functions that Return a Date Value

#BEGMONTH[(date)]

Returns the first day of the month in which the date argument occurs. Notes 2, 3

```
Example: COMPUTE: A = #BEGMONTH(5/15/2007)  results in  A=5/1/2007
```

#BEGWEEK[(date)]

Returns the Sunday of the calendar week in which the date argument occurs. Notes 2, 3

```
Example: COMPUTE: A = #BEGWEEK(5/15/2007)  results in  A=5/13/2007
```

Note: You can also use this function to return *any* particular day of a given week (Monday, Tuesday, etc.). Just use it in combination with an #INCDATE function that adds the appropriate number of days to the result . Add 1 to get Monday, 2 to get Tuesday, and so on. The following example returns the Wednesday of the week that SALES-DATE falls within.

```
Example: WED-SALES-DATE = #INCDATE(#BEGWEEK(SALES-DATE), 3, DAYS)
```

#BEGYEAR[(date)]

Returns the first day of the year in which the date argument occurs. Notes 2, 3

Example: `COMPUTE: A = #BEGYEAR(5/15/2007)` results in `A=1/1/2007`

#ENDMONTH[(date)]

Returns the last day of the month in which the date argument occurs. Notes 2, 3

Example: `COMPUTE: A = #ENDMONTH(5/15/2007)` results in `A=5/31/2007`

#ENDWEEK[(date)]

Returns the Saturday of the calendar week in which the date argument occurs. Notes 2, 3

Example: `COMPUTE: A = #ENDWEEK(5/15/2007)` results in `A=5/19/2007`

#ENDYEAR[(date)]

Returns the last day of the year in which the date argument occurs. Notes 2, 3

Example: `COMPUTE: A = #ENDYEAR(5/15/2007)` results in `A=12/31/2007`

#INCDATE([date,] number, units)

Returns the date obtained by incrementing the argument date by the given number of units. Units can be any of these keywords or abbreviations: Notes 2, 3

- DAYS, DAY, D
- WEEKS, WEEK, WKS, WK, W
- MONTHS, MONTH, MONS, MON, M
- YEARS, YEAR, YRS, YR, Y

Example: `COMPUTE: A = #INCDATE(5/15/2007, 3, WEEKS)` results in `A = 6/5/2007`
`COMPUTE: YESTERDAY = #INCDATE( -1, DAYS)` results in `YESTERDAY` being the date before the system date.

Note: When incrementing by months or years, the day portion of the resulting date is sometimes changed to the last day of the month, in order to return a valid calendar date.

`COMPUTE: A = #INCDATE(5/31/2007, 1, MONTH)` results in `A = 6/30/2007` (not `6/31/2007` which is not a valid date)
`COMPUTE: B = #INCDATE(2/29/2008, 1, YEAR)` results in `B = 2/28/2009` (not `2/29/2009` which is not a valid date)

#INCDATETIME([date,] [time,] number, units) or
#INCDATETIME([date,] [time,] time)

Returns the date obtained by incrementing the date and time arguments by the given number of units, or by a time value. Units can be any of these keywords or abbreviations: Notes1, 2, 3

- SECONDS, SECOND, SECS, SEC, S

- MINUTES, MINUTE, MINS, MIN, M
- HOURS, HOUR, HRS, HR, H

Examples:

```
COMPUTE: A= #INCDATETIME(1/1/2008, 23:45:00, 12, MINUTES)  results in  A = 1/1/2008
COMPUTE: B= #INCDATETIME(1/1/2008, 23:45:00, 16, MINUTES)  results in  A = 1/2/2008
```

Note: This function is often used in conjunction with #INCTIME. Together, they let you add a time interval to a starting date and time and get the resulting date and time. For example, to compute an "expiration" date and time that is 12 hours after SALES-DATE and SALES-TIME, you could use the following:

```
Example: COMPUTE: EXPIRE-DATE = #INCDATETIME(SALES-DATE, SALES-TIME, 12, HOURS)
         COMPUTE: EXPIRE-TIME = #INCTIME(SALES-TIME, 12, HOURS)
```

Note: Transitions to or from Daylight Savings Time are not taken into account by this function.

#MAKEDATE(char/num)

For **character arguments**, converts the YYMMDD or YYYYMMDD character string into the corresponding date. The character argument must be either 6 or 8 bytes in length. When a YYMMDD argument is used, Spectrum Writer assigns the century based on the CENTURY option in effect (page 559):

```
Example: COMPUTE: A = #MAKEDATE('20070331')  results in  A=3/31/2007
```

For **numeric arguments**, the argument is treated as a "day in century" value. The numeric argument must between be 1 (corresponding to January 1, 1900) and 73,049 (corresponding to December 31, 2099) inclusive. The function returns the date corresponding to the numeric day from the start of the 20th century. (Use this function to change the results of the #MAKENUM(date) function back into a date.)

```
Example: COMPUTE: A = #MAKEDATE(39446)  results in  A=12/31/2007
```

#YMD(num, num, num)
#MDY(num, num, num)
#DMY(num, num, num)

Returns a date value based on the three numeric arguments (representing month, day and year in the order indicated by the function name.) The resulting date is not validity-checked to see if it is an actual calendar date. (You can use the #REALDATE function to find out.) The numeric argument representing the year can be any 1 to 4 digit number, and the month and day arguments can be any 1 or 2 digit number.

```
Example: COMPUTE: A = #MDY(12,31,2007)  results in  A=12/31/2007
         COMPUTE: B = #YMD(9999,99,99)  results in  B=99/99/9999
```

Functions that Return a Time Value

#INCDURATION([time,] number, units) or
#INCDURATION([time,] time)

Returns the time duration (not necessarily a time of day) obtained by incrementing the time argument by the given number of units, or by another time value. The result is treated as a

time duration (interval) and is not converted to a proper time of day. Units can be any of these keywords or abbreviations: Note1

- SECONDS, SECOND, SECS, SEC, S
- MINUTES, MINUTE, MINS, MIN, M
- HOURS, HOUR, HRS, HR, H

```
Example: COMPUTE: A=#INCDURATION(23:00:00, 2, HOURS)  results in  A = 25:00:00
         COMPUTE: B=#INCDURATION(23:00:00, 05:12:34)  results in  B = 28:12:34
```

Note: See also the related #INCTIME built-in function.

#INCTIME([time,] number, units) or
#INCTIME([time,] time)

Returns the time of day obtained by incrementing the time argument by the given number of units, or by another time value. The result will always be a proper time of day (that is, it will be in the range 00:00:00 to 23:59:59). Units can be any of these keywords or abbreviations: Note1

- SECONDS, SECOND, SECS, SEC, S
- MINUTES, MINUTE, MINS, MIN, M
- HOURS, HOUR, HRS, HR, H

```
Example: COMPUTE: A=#INCTIME(23:00:00, 2, HOURS)  results in  A = 01:00:00
         COMPUTE: B=#INCTIME(23:00:00, 05:12:34)  results in  B = 04:12:34
```

Note: See also the related #INCDURATION built-in function.

Note: This function is often used in conjunction with #INCDATETIME. Together, they let you add a time to a starting date and time and get the resulting date and time. For example, to compute an "expiration" date and time that is 12 hours after SALES-DATE and SALES-TIME, you could use the following:

```
Example: COMPUTE: EXPIRE-DATE = #INCDATETIME(SALES-DATE, SALES-TIME, 12, HOURS)
         COMPUTE: EXPIRE-TIME = #INCTIME(SALES-TIME, 12, HOURS)
```

Note: Transitions to or from Daylight Savings Time are not taken into account by this function.

Note: You can use this function to convert an improper time-of-day to a proper time-of-day.

```
Example: COMPUTE: PROPER-TIME = #INCTIME(29:12:34, 0, SECS) results in
                                                  PROPER-TIME = 05:12:34
```

#MAKETIME(char/num)

For **character arguments**, converts the HHMMSS character string into the corresponding time. The character argument must be exactly 6 bytes long.

```
Example: COMPUTE: A = #MAKETIME('135959')  results in  A = 13:59:59
```

For **numeric arguments**, the argument is treated as a number of seconds. The number of seconds is converted into the corresponding number of hours, minutes and seconds. (Use this function to change the results of a #MAKENUM(time) function back into a time.)

```
Example: COMPUTE: A = #MAKETIME(3600)  results in  A = 01:00:00
```

Functions that Return a Boolean (or Bit) Value

#ERROR(fieldname)

Returns "true" if the argument field is "in error." Otherwise, it returns "false."

Fields which are in error appear in a report with error indicators (like ***I***, ***Z***, ***V***, etc.). Examples of fields in error are fields containing invalid packed data and compute fields where a divide-by-zero or an overflow occurred. (Missing fields are not in error. Use the #MISSING built-in function (page 642) to check for that.)

```
Example: COMPUTE: A=#ERROR(AMOUNT)   results in A being "true," if the AMOUNT field in
                                     the input record does not contain a valid numeric
                                     value; otherwise A will be "false."
         COMPUTE: X = 3 / 0
         COMPUTE: B=#ERROR(X)        results in B being "true"
```

#ISNUM(char)

Returns "true" if the character argument contains only numeric characters (optionally preceded by leading blanks.) Otherwise, it returns "false." All-blank fields and fields with decimal points or commas, etc. return "false."

```
Example: COMPUTE: A=#ISNUM(' 123')    results in A being "true"
         COMPUTE: B=#ISNUM('123.45')  results in B being "false"
         COMPUTE: C=#ISNUM('JONES')   results in C being "false"
```

Note: A character field that returns "true" for this function can be converted to a numeric field with the #MAKENUM function.

#LEAPYEAR[(date)]

Returns "true" if the year portion of the argument date is a leap year. Otherwise, it returns "false." (The month and day portions of the argument date are not examined for validity.) (This function may also be abbreviated #LEAP.) Note 2

```
Example: COMPUTE: A=#LEAPYEAR(5/15/2007) results in A being "false"
         COMPUTE: B=#LEAPYEAR(5/15/2008) results in B being "true"
```

#MISSING(fieldname)

Returns "true" if the argument field is in a "missing" auxiliary input record. Otherwise, it returns "false."

Fields in the primary input file (specified by the INPUT statement) are never missing. Fields in records read from an auxiliary input file (specified in a READ statement) are missing when no record is found that matches a particular "read key." Fields from DB2 tables used as auxiliary inputs are missing when no row meets the conditions specified in the WHERE parm.

```
Example: INPUT: SALES-FILE
         READ:  EMPL-FILE  READKEY(EMPL-NUM)
         COMPUTE: A=#MISSING(FIRST-NAME)
```

Since FIRST-NAME is a field in the EMPL-FILE, the example above results in A being "true" if there is no record in the EMPL-FILE whose key matches the EMPL-NUM in the current SALES-FILE record. If such a record does exist, A will be "false."

#OFF

Always returns a "false" value.

```
Example: COMPUTE: A = #OFF  results in A being "false"
         COMPUTE: SALES-AWARD = WHEN(TOTAL-SALES > 50000) ASSIGN(#ON)
                                ELSE                      ASSIGN(#OFF)
```

The second example above results in SALES–AWARD being "true" if total sales are greater than 50,000; otherwise, it results in SALES–AWARD being "false".

#ON

Always returns a "true" value.

```
Example: COMPUTE: A = #ON  results in A being "true"
         COMPUTE: SALES-AWARD = WHEN(TOTAL-SALES > 50000) ASSIGN(#ON)
                                ELSE                      ASSIGN(#OFF)
```

The second example above results in SALES–AWARD being "true" if total sales are greater than 50,000; otherwise, SALES–AWARD will be "false".

#REALDATE(date)

Returns "true" if the date argument contains a real calendar date. Otherwise, it returns "false."

```
Example: COMPUTE: A=#REALDATE(6/30/2007)   results in A being "true"
         COMPUTE: B=#REALDATE(6/31/2007)   results in B being "false"
         COMPUTE: C=#REALDATE(2/29/2007)   results in C being "false"
         COMPUTE: D=#REALDATE(99/99/9999)  results in D being "false"
```

Function Notes:

1. If the time argument is omitted, the system time is used.
2. If the date argument is omitted, the system date is used.
3. If the date argument is not a valid calendar date, the function returns an "invalid data" error (***I***).

Note: You can use #REALDATE to determine whether a given date is valid or not.

```
Example: COMPUTE X-IS-REALDATE = #REALDATE(X)
         COMPUTE: BEGIN-X-MONTH =
                    WHEN(X-IS-REALDATE) ASSIGN(#BEGMONTH(X))
                    ELSE                ASSIGN(99/99/9999)
```

The above example results in BEGIN-X-MONTH being the first day of the month in which X occurs *if* X is a valid calendar date. Otherwise, BEGIN-X-MONTH will contain 99/99/9999.

Appendix E. Error Indicators

Sometimes an error prevents Spectrum Writer from displaying the desired data in a report or PC file. When that happens a number of asterisks are printed where that data should have appeared. A single letter is imbedded within the asterisks. That letter is an **error code** which tells you exactly what kind of error has occurred. The following table lists the error codes:

MEANING OF ERROR INDICATORS	
ERROR CODE	**MEANING**
****A****	**Ambiguous reference**. You asked to print a certain field here, but there is more than one field by that name in the input file(s). Use a record name to indicate exactly which field you mean (page 228).
****E****	**Error in definition**. You asked to print a certain field here, but that field was defined in error. Look for error messages concerning the FIELD or COMPUTE statement used to define the field. Correct those errors.
****F****	**Offset error occurred**. You asked to print a field here, but an error occurred while trying to compute the field's location within the input record. Offset errors occur when the sum of the OFFSET value and the COLUMN/DISP value (or the default value used) are not within the I/O area reserved for the input record. (The size of this I/O area is determined by the record size specified in the FILE, INPUT or READ statement.) Offset errors also occur when a computation error arises while computing the OFFSET value. This includes division by zero, overflow, or any reference to another field that is in error. If desired, you can use the MISSOFFSET option to ignore this error condition (page 566).
****I****	**Invalid data**. You asked to print a certain field here, but that field contained invalid data. For example, the field was supposed to contain packed data and instead it contained spaces. Or, a field that was supposed to contain a date actually contained alphabetic characters. Correct the data in the input file. To help you identify the problem, Spectrum Writer prints a hex listing of the record containing the invalid data and identifies the field that is in error. Look for this in the Spectrum Writer control listing. If desired, you can use the ZEROINVDATA (or just ZEROINV) option to ignore this error condition. (page 577) A MAXINVSHOW option is also available to let you control how many records with invalid data are printed in the control listing (page 566).

MEANING OF ERROR INDICATORS	
ERROR CODE	**MEANING**
******S******	**Size error**. You asked to print a numeric field here, but there was not enough room to show all of its significant digits (and a minus sign, if the number was negative). Use a **width parm** to increase the number of characters reserved to print this field. (See "How to Change the Width of a Column" on page 135.) As an example, the following statement reserves 20 characters to print the TOTAL–SALES field: `COLUMNS: EMPL-NAME  TOTAL-SALES(20)`
******U******	**Undefined field**. You asked to print a certain field here, but that field is not defined in any input file for the current run. You may have just misspelled the field name. Or, the field may belong to a file that is not an input file to the current report. **Tip:** To see a list of all field names available for a file, add the SHOWFLDS(YES) parm to your INPUT and READ statements.
******V******	**Overflow occurred**. You asked to print a computed numeric field here, but an overflow error occurred while trying to compute its value. This may happen when two very large numbers are multiplied together. It can also happen when a very large number is divided by a very small number (like .000000001). Try requesting that fewer decimal places be kept in the computed result. Also try splitting complex COMPUTE statements into several simpler COMPUTE statements. Spectrum Writer can maintain a maximum of 31 digits (including decimal digits) in computed fields. (This also applies to any intermediate results used to compute the final result.) If desired, you can use the ZEROOVERFLOW (or just ZEROOVER) option to ignore this error condition (page 577).

******Z******	**Divide by zero occurred.** You asked to print a computed numeric field here, but a division by zero was attempted while trying to compute its value. You may be able to use a conditional COMPUTE statement to prevent division by zero, like this:
	`COMPUTE: RATIO = WHEN(B ¬= 0)   ASSIGN(A/B)`
	`                ELSE           ASSIGN(0)`
	
	If desired, you can use the ZERODIVZERO (or just ZERODIVZ) option to ignore this error condition (page 577).

Suppressing Error Indicators

In some situations, certain error conditions may not be critical to your report. In such cases, you can suppress the asterisk error indicators by using one or more of the following OPTIONS statement options.

OPTIONS THAT SUPPRESS ERROR INDICATORS	
OPTION	**MEANING**
ZEROINVDATA	Treat fields containing invalid data as if they contained zeros instead. This suppresses the ***I*** indicator. May also be abbreviated ZEROINV.
ZEROOVERFLOW	Assign a value of zero to COMPUTE fields that have overflow errors. This suppresses the ***V*** indicator. May also be abbreviated ZEROOVER.
ZERODIVBYZERO	Assign a value of zero to COMPUTE fields when a division by zero is attempted. This suppresses the ***Z*** indicator. May also be abbreviated ZERODIVZERO and ZERODIVZ.
MISSOFFSET	Treat fields that have OFFSET parm errors as if the field was "missing." (Missing fields are assigned zeros for numeric, date and time fields, blanks for character fields, and OFF for bit fields.) This suppresses the ***F*** indicator.

The above options tell Spectrum Writer to treat fields that have the specified error as if they contained a zero (or missing) value. This means you'll see zeros in your output, rather than the asterisk error indicators. (For character fields with the offset error, you'll see blanks instead of the error indicator.) It also prevents fields from propagating their error conditions to other fields that reference them. (See discussion below.)

If you want invalid numeric items (along with other zero values) to appear as blanks (rather than zeros) in your output, use a PICTURE that specifies suppression of all leading zeros, like this:

```
OPTIONS:  ZEROINVDATA
COLUMNS:  AMOUNT(PIC'ZZZ,ZZZ')
```

Alternatively, you could use the BIZ ("blank if zero) parm:

```
OPTIONS:  ZEROINVDATA
COLUMNS:  AMOUNT(BIZ)
```

Propagation of Error Indicators

When a field which has an error is used as an operand in a COMPUTE statement, its error code is normally passed on to the result field. Consider the following statement:

```
COMPUTE: B = A + 1
```

Assume that A is defined as a packed field. If a certain record contains invalid packed data for field A, then **I** will appear in the report where the contents of A should have appeared. *In addition*, you will also see **I** anywhere that field B should have printed. That is because field A, which is needed to compute field B, passes its error condition on to field B.

Determining if a Field Is In Error

You may want to determine if a field is in error *before* trying to use its value. That allows you to use alternate processing for invalid fields to avoid propagating the error further.

You can use the #ERROR built-in function (page 642) to do that. That function accepts any fieldname as a parm and returnd "true" or "false" whether the field is in error. Examples of fields in error are fields containing invalid packed data and compute fields where a divide-by-zero or an overflow occurred. (Missing fields are not in error. Use the #MISSING built-in function (page 642) to check for that.)

```
Example: COMPUTE: INV-AMOUNT = #ERROR(AMOUNT)
         COMPUTE: TAX-PERCENT = WHEN(INV-AMOUNT) ASSIGN(0.08)         /* DEFAULT */
                                ELSE             ASSIGN(TAX / AMOUNT) /* ACTUAL  */
```

Appendix F. Files Used in Examples

The sample reports used in this manual were created using actual files. The boxes on the following pages show the definition statements (that is the FILE and FIELD statements) that were used to define these files. The contents of the SWALIAS member of the sample copy library is also shown. Finally, each file's raw unformatted data is also shown.

Sample File Definitions

STATEMENTS STORED IN SWALIAS MEMBER OF COPY LIBRARY

```
SALES-FILE    =  SALES
EMPL-FILE     =  EMPLFILE
PRODUCT-FILE  =  PRODFILE
STATE-FILE    =  STATE
```

DEFINITION STATEMENTS FOR SALES–FILE

```
*****************************************************************
*                                                               *
*  SPECTRUM WRITER FILE DEFINITION FOR SALES-FILE               *
*                                                               *
*****************************************************************
FILE:  SALES-FILE      DDNAME(SALEFILE)   LRECL(80)
*
FIELD: EMPL-NAME       LENGTH(10)
FIELD: EMPL-NUM        LENGTH(3)
FIELD: BACKUP-EMPL-NUM LENGTH(3)
FIELD: REGION          LENGTH(5)
FIELD: AMOUNT          LENGTH(6)  TYPE(NUM)    DECIMAL(2)
FIELD: TAX             LENGTH(4)  TYPE(NUM)    DECIMAL(2)
FIELD: COMMISSION-RATE LENGTH(4)  TYPE(NUM)    DECIMAL(3)
FIELD: SALES-DATE                 TYPE(YYMMDD)
FIELD: SALES-TIME                 TYPE(HHMMSS)
FIELD: CUSTOMER        LENGTH(15)
FIELD: TELEPHONE       LENGTH(10) TYPE(NUM)
FIELD: TIME-ON-PHONE   LENGTH(4)  TYPE(SECS)   DECIMAL(1)
FIELD: PRODUCT-CODE    LENGTH(3)
```

Remarks:

- these statements are stored in the SALES member of the copy library
- for VSE, the following FILE statement is used instead:

```
FILE: SALES-FILE  ATTR(DASD,'SALEFIL',80,160)
```

DEFINITION STATEMENTS FOR EMPL–FILE

```
************************************************************
*                                                          *
*  SPECTRUM WRITER FILE DEFINITION FOR EMPL-FILE           *
*                                                          *
************************************************************
FILE:  EMPL-FILE TYPE(VSAM) DDNAME(EMPLFILE)  LRECL(150)
*
FIELD: EMPL-NUM       LEN(3)
FIELD: LAST-NAME      LEN(15)
FIELD: FIRST-NAME     LEN(15)
FIELD: HIRE-DATE              TYPE(YYMMDD)
FIELD: DEPT-NUM       LEN(1)  TYPE(NUM) NOACCUM
FIELD: SEX            LEN(1)
FIELD: STATUS-BYTE    LEN(1)
FIELD: FULL-TIME      COL(STATUS-BYTE) BIT(1)
FIELD: SOCIAL-SEC-NUM COL(*+1) LEN(9)  TYPE(NUM)
                               FORMAT(PIC'999-99-9999')
FIELD: NUM-ACCOUNTS   LEN(4)  TYPE(NUM)
FIELD: TOTAL-SALES    LEN(7)  TYPE(NUM) DEC(2)
FIELD: SALES-QTR1     LEN(7)  TYPE(NUM) DEC(2)
FIELD: SALES-QTR2     LEN(7)  TYPE(NUM) DEC(2)
FIELD: SALES-QTR3     LEN(7)  TYPE(NUM) DEC(2)
FIELD: SALES-QTR4     LEN(7)  TYPE(NUM) DEC(2)
FIELD: ADDRESS        LEN(20)
FIELD: CITY           LEN(15)
FIELD: STATE          LEN(2)
FIELD: ZIP            LEN(5)
FIELD: TELEPHONE      LEN(10) TYPE(NUM)
                              FORMAT(PIC'(999) 999-9999')
```

Remarks:

- these statements are stored in the EMPLFILE member of the copy library
- for VSE, the following FILE statement is used instead:

```
FILE: EMPL–FILE ATTR(VSAM,'EMPLFIL',150)
```

DEFINITION STATEMENTS FOR PRODUCT–FILE

```
************************************************************
*                                                          *
*  SPECTRUM WRITER FILE DEFINITION FOR PRODUCT-FILE        *
*                                                          *
************************************************************
FILE: PRODUCT-FILE  DDNAME(PRODFILE)  TYPE(VSAM)  LRECL(22)
*
FIELD: PRODUCT-STATUS  LEN(3)
FIELD: PRODUCT-KEY     LEN(4)
FIELD: PRODUCT-DESC    LEN(15)
```

Remarks:

- these statements are stored in the PRODFILE member of the copy library
- For VSE, the following FILE statement is used instead:

```
FILE: PROD–FILE ATTR(VSAM,'PRODFIL',22)
```

DEFINITION STATEMENTS FOR STATE–FILE

```
************************************************************
*                                                          *
*  SPECTRUM WRITER FILE DEFINITION FOR STATE-FILE          *
*                                                          *
************************************************************
FILE:   STATE-FILE TYPE(VSAM)  DDNAME(STATFILE)  LRECL(20)
*
FIELD:  STATE-CODE  LEN(2)
FIELD:  STATE-NAME  LEN(10)
```

Remarks:

- these statements are stored in the STATE member of the copy library
- for VSE, the following FILE statement is used instead:

```
FILE: STATE-FILE ATTR(VSAM,'STATFIL',20)
```

Sample Files' Raw Data

CONTENTS OF SALES–FILE (UNFORMATTED)

```
JOHNSON   037041SOUTH0101380609035095031210250OACE ELECTRICAL 21355598710079952
BAKER     044045WEST 013700082203609503261209O9JACKS CAFE     21455511240102978
MORRISON  042036EAST 004435026603609503291530220STAR MARKET    40855576540599907
MORRISON  042045EAST 002965017803609503301905410 PHOTOGRAPHY 40855577860600919
SIMPSON   041039EAST 001499009003609504010817570EUROPEAN DELI  40855565430150916
JOHNSON   039036NORTH023445140703709504011702470VILLA HOTEL    41555576300929926
JOHNSON   039044NORTH000998006003709504051433100MARYS ANTIQUES 41555512560000997
BAKER     044037WEST 013575081503609504121431120JACKS CAFE     21455511240231916
THOMAS    045037WEST 000998006003609504141541380YOGURT CITY    21455517895421997
JONES     036042NORTH001025006203709504150758320EZ GROCERY     41555548720810977
JONES     036039NORTH012176073103709504150801590TOY TOWN       41555515001200907
JONES     036039NORTH001025006203709504151352410TOY TOWN       41555515000523977
JOHNSON   037042SOUTH050000300003509504161148330ACME BUILDING  21355521211025976
SIMPSON   041042EAST 002387014303609504301530210J & S LUMBER   40855523212451916
```

CONTENTS OF EMPL–FILE (UNFORMATTED)

```
036JONES          JERRY          8001312MA012098765007842509890995601105115608698071333425125 MAIN S
TREET     SAN FRANCISCO  CA940124155557653
037JOHNSON        THOMAS         7506211MA91204033401288699924215601521350211997010241187840000 LINDA
 VISTA    SCOTTSDALE     AZ900126025556654
039JOHNSON        LINDA          7911252FA00477998101047502355145903417220102010008231131212 LINCOLN
 DRIVE    SANTA ROSA     CA954124155556785
040MACDONALD      RICHARD        8207042M 889790013000602560980054850006871300599250072610525 FOOTHI
LL DRIVE  PLEASANTON     CA945684155559887
041SIMPSON        TIMOTHY        8212013MA11205045600160872388012875805109030099812013291589876 WEST
 53 STREETARCADIA        CA910068185551887
042MORRISON       MICHAEL        7911303MA900120556015498054992501419261122128010091891850 98 SOUTH L
AKESIDE DRGLENDALE       CA912028185554748
043CHRISTOPHERSON MELISSA        8108151FA41509076100654766531138072216549010805007092590161752 TIMB
ERIDGE RD PHOENIX        AZ905026025554556
044BAKER          VIVIAN         8206044FA878190156014792125892133610249990224001332178944667 CRESTH
AVEN BLVD WALNUT CREEK   CA945984155551209
045THOMAS         MARTIN         8206044MA776838221011860193491488907180450514250121300925778 12 S. H
UNTINGTON CONCORD        CA945194155551152
```

CONTENTS OF PRODUCT–FILE (UNFORMATTED)

```
NEWP907INKPADS
NEWP916RED PENS
NEWP919GREEN PENS
OLDP926DESK CALENDARS
NEWP952PENCILS (NO. 1)
OLDP976CHAIRS
OLDP977PAPER CLIPS
NEWP978HOLE PUNCHERS
OLDP997MAILING LABELS
```

CONTENTS OF STATE–FILE (UNFORMATTED)

```
AZARIZONA
CACALIFORNIA
OROREGON
WAWASHINGTON
```

Appendix G. Speed-Up Tips

Because Spectrum Writer is written entirely in fast, efficient Assembly language, it runs faster than any other 4GL report writer we know of. This Appendix lists some techniques you can use when writing your queries to allow Spectrum Writer to run at its fastest. You may want to review these items if you have large, long–running jobs where minimizing CPU use is especially important.

INCLUDEIF Statement

The INCLUDEIF statement is perhaps the single most important factor that affects how long your job will run. By considering the following suggestions when writing your INCLUDEIF statements, you can help Spectrum Writer run at its fastest.

The INCLUDEIF statement simply consists of a conditional expression. Spectrum Writer always *stops processing* a conditional expression as soon as it knows that the entire expression is either definitely true or definitely false. That means that Spectrum Writer may not always need to perform every test in a conditional expression. By writing your conditional expressions so that Spectrum Writer can make a definite determination as soon as possible, you can help eliminate unnecessary processing. That reduces CPU usage.

> **Speed-Up Tip:** Put tests that definitely include or definitely exclude the majority of input file records early in your INCLUDEIF statement.

We will now illustrate this tip in detail, both for conditional expressions that use AND and for conditional expressions that use OR.

Order of ANDed Tests

As an example, assume that we are processing a large database of people. We want to include all records where *both* of the following conditions are true:

- SEX = 'F'
- NAME = 'JOSEPHSON'

Note that one of these conditions (SEX = 'F') should be true in about half of the input records. (We are assuming that the database is representative of the population at large.) The other condition (NAME = 'JOSEPHSON') will probably be true for only a tiny fraction of the database— far less than 1%.

We could write the necessary INCLUDEIF statement either of two ways.:

```
1. INCLUDEIF: SEX = 'F' AND NAME = 'JOSEPHSON'
2. INCLUDEIF: NAME = 'JOSEPHSON' AND SEX = 'F'
```

If we use the first statement above, Spectrum Writer will have to perform *both tests* on approximately 50% of the input records. That is because the first test (SEX = 'F') will be true for about half of the input file. For that half of the file, the second test will then have to be performed as well (NAME = 'JOSEPHSON'). (When this second test is performed, most of the records will fail it and will thus fail the entire INCLUDEIF statement.)

Now consider the second (and much better) way that we would write our INCLUDEIF statement:

```
INCLUDEIF: NAME = 'JOSEPHSON'  AND  SEX = 'F'                <--best choice
```

The above statement results in exactly the same records being included in the report, but it is **much more efficient** in terms of CPU use. In this case, 99% of the input file records will fail the first test. For those records, the second test will not need to be performed at all. Spectrum Writer can definitely exclude the input record with just a single test 99% of the time. It will only need to perform the second test (SEX = 'F') on less than 1% of the input records.

To compare the two methods, let's assume that our database contains one million people. Using the first INCLUDEIF statement discussed above, Spectrum Writer would have to perform about 1,500,000 tests to evaluate the INCLUDEIF statement for the entire file. (1,000,000 SEX tests, plus 500,000 NAME tests.) Using the second INCLUDEIF statement discussed above, Spectrum Writer would have to perform less than 1,010,000 tests. (1,000,000 NAME tests, plus less than 10,000 SEX tests.) You can see that the second INCLUDEIF statement would use almost 33% less CPU than the first one.

> **Speed-Up Rule:** When using multiple tests separated with AND, put the most difficult test to pass first. Put the next–most–difficult test second, and so on. By "most difficult test," we mean the test that the most input file records will fail. By "next–most–difficult test," we mean the test that will be failed most often by those records that have passed the first test.

Order of ORed Tests

Now let's consider conditional expressions that use OR. Assume now that we want to include all the people in our database where *either* of the following conditions are true:

- SEX = 'F'
- NAME = 'JOSEPHSON'

Again, we can assume that about 50% of the records will pass the first test shown above, and less than 1% will pass the second test.

Here is the best way to write the INCLUDEIF statement:

```
INCLUDEIF: SEX = 'F'  OR  NAME = 'JOSEPHSON'                 <--best choice
```

By using the above statement, Spectrum Writer will definitely include about 50% of the file after evaluating only the first test. It will only have to perform the second test on the other 50% of the file.

On the other hand, consider if we had written the statement this way:

```
INCLUDEIF: NAME='JOSEPHSON'  OR  SEX='F'
```

If we used the above statement, the first test would not be true over 99% of the time. That means that Spectrum Writer would have to go on to perform the second test on 99% of the input file. While both statements would include the same records in your report, the above statement would require almost twice as much CPU time to process as the earlier statement.

As you can see, the rule is reversed when using multiple conditions that are separated with OR.

Speed-Up Rule: When using multiple tests separated with OR, put the easiest test to pass first. Put the next–easiest test second, and so on. By "easiest test," we mean the test that the most input file records will pass. By "next–easiest test," we mean the test that will be passed most often by those records which did not pass the first test.

One common way that this rule comes up is when you are including records where a certain field is equal to any one of a number of values.

Example

```
INCLUDEIF:  DEPT-NUM = 2 OR 3 OR 4
```

You will improve performance in such a case if you put the most common value first. For example, if more people in the input file are in department 4 than are in department 2 or 3, you should put 4 first:

```
INCLUDEIF:  DEPT-NUM = 4 OR 2 OR 3
```

Fields from Auxiliary Input Files

So far, we have assumed that all fields referred to in an INCLUDEIF statement come from one file. When the INCLUDEIF statement refers to fields from two or more files, there is another factor to consider. As we mentioned earlier, Spectrum Writer stops processing a conditional expression as soon as it knows that the entire expression is either definitely true or definitely false. That means that if Spectrum Writer can definitely exclude a record based only on tests from the primary input file, it will not have to perform any subsequent tests that involve the auxiliary input file(s). In most cases, Spectrum Writer does not read an auxiliary input record until data from that record is actually needed for processing. Thus, if you can exclude a large percentage of records based solely on primary input file tests, Spectrum Writer will not have to read their auxiliary records at all and you will save a large amount of I/O. Since I/O is relatively slow, it is always desirable to avoid unnecessary I/O whenever possible.

Let's consider an example using our large database of people. Assume that it contains an ID number for each person that can be used as the key to another file that contains birth date information. Assume that we want to include people in our report if both of the following conditions are true:

- NAME = 'JOSEPHSON'
- BIRTHDATE = 1/1/1965

The best way to write the INCLUDEIF statement is:

```
INCLUDEIF: NAME='JOSEPHSON' AND BIRTHDATE = 1/1/1965
```

In the above statement, 99% of the input file will be definitely excluded based on the first test alone. That means that 99% of the time the "read" to the auxiliary input file containing the BIRTHDATE field will not be necessary. This method reduces the amount of I/O performed by almost half (compared with writing the statement with the BIRTHDATE test first). When the BIRTHDATE test is written first, the auxiliary record has to be read 100% of the time.

If we had an OR–type INCLUDEIF statement, we would probably still want to put the primary input file test first:

```
INCLUDEIF: NAME='JOSEPHSON' OR  BIRTHDATE = 1/1/1965
```

In the above case, only a small percentage of the input records would pass the first test, meaning that the auxiliary record would then have to be read in order to perform the second test. Still, reading the second file 99% of the time is slightly better than reading it 100% of the time, as would be the case if the BIRTHDATE test were the first test.

> **Speed–Up Tip:** When the INCLUDEIF statement involves tests using fields from auxiliary input files, try to make the auxiliary file tests the last ones.

Of course, there will be times when your inclusion requirements prevent you from doing this. Or, you may have a conflict between the rules specified earlier (involving easy–to–pass and difficult–to–pass tests) and the rule regarding tests from auxiliary input files. In such cases, you may want to experiment with the INCLUDEIF statement on test runs until you find the most efficient way to write it for your situation. For regularly scheduled, long running jobs, it may be worth the effort to do that.

Intermediate Conditional Expressions

If your INCLUDEIF statement uses the same tests in multiple places, you may be able to improve performance by assigning the result of those tests to an intermediate bit field. This technique is discussed on page 657.

Conditional COMPUTE Statements

When writing conditional COMPUTE statements, there are two considerations that affect performance:

- the order of the tests within each WHEN parm
- the order of the WHEN parms themselves

The contents of a WHEN parm is simply a conditional expression. The INCLUDEIF statement also consists of a conditional expression. Therefore, carefully read the above tips regarding the INCLUDEIF statement. Follow those same suggestions when writing the conditional expressions within your WHEN parms.

For example, consider the following WHEN parm:

```
COMPUTE: A = WHEN(SEX='F' OR NAME='JOSEPHSON') ASSIGN(...)        <--best choice
```

The above WHEN parm is more efficient than writing it the following way (even though both ways yield the same final result):

```
COMPUTE: A = WHEN(NAME='JOSEPHSON'  OR  SEX='F')  ASSIGN(...)
```

If you don't know why the first statement above is better, read the earlier section titled "INCLUDEIF Statement" (page 652).

The second consideration when writing conditional COMPUTE statements is the order of the WHEN parms themselves. Remember that when evaluating a conditional COMPUTE statement, Spectrum Writer stops evaluating the WHEN parms as soon as it finds a WHEN expression that is true. Thus, you will want to put the WHEN parms that are most likely to

be true as early as possible. That lets Spectrum Writer stop its WHEN parm processing as early as possible in the maximum number of cases.

> **Speed–Up Tip:** Put the WHEN parm that is most likely to be true first. Next, put the WHEN parm that is most likely to be true (considering only those records that failed the first WHEN parm), and so on.

Consider the following statement:

```
COMPUTE: STATE=NAME = WHEN(STATE = 'CA')   ASSIGN('CALIFORNIA')
                      WHEN(STATE = 'NY')   ASSIGN('NEW YORK')
                      ...
                      WHEN(STATE = 'WY')   ASSIGN('WYOMING')
```

Notice that the WHEN parms are not in alphabetical state order like you might expect. Instead, they appear in order of *decreasing state population*. Thus (again assuming that our database is representative of the US population as a whole) the WHEN parm most likely to be true for the entire file (STATE = 'CA') comes first. For about 12% of the input records, Spectrum Writer will only have to evaluate this one WHEN parm (since about 12% of the population live in California).

Next, considering only those records that are not in California, the most records will be in New York. Therefore, we checked for STATE='NY' second. This allows another 7% of the input file to have only two WHEN parms evaluated. And so on through the rest of the states. Spectrum Writer would only have to evaluate all 50 WHEN parms for 0.2% of the input records (for Wyoming).

Putting the WHEN parms in the above order ensures that Spectrum Writer performs the fewest total number of WHEN parm evaluations, thus ensuring the best performance.

Of course, your COMPUTE statements will involve different conditions. It may be hard for you to guess which of your WHEN parms are the most likely to be true. But, even if you can only identify the one or two most common WHEN parms, just putting those first can result in a significant benefit.

Compute Statements with RETAIN

COMPUTE statements that use the RETAIN keyword can be much slower than COMPUTE statements that do not use it. The reason is this: if an input record will not be included in the run (because it fails the INCLUDEIF tests), Spectrum Writer does not normally have to compute the value of all the COMPUTE statements for that record. However, it *does* have to compute the value of all RETAIN–type COMPUTE statements for those records. This is because, even though a specific record may not be included in the report, the value assigned to the COMPUTE field for that record might need to be retained and then used in conjunction with later input records.

RETAIN–type COMPUTEs are especially slow when they refer to fields from auxiliary input records. The reason: since RETAIN–type COMPUTEs must be computed for *every* input file record, that means that the auxiliary input file record needed for the COMPUTE must also be read for every input file record— even those records that won't be included in the report. That can add a lot of I/O time to a run, since direct reads to auxiliary input files are relatively slow.

Tip: If you have a RETAIN–type COMPUTE statement that refers to a field from an auxiliary input file, see if you can replace it with a non–RETAIN–type COMPUTE statement. Sometimes you can accomplish this by using a RETAIN–type COMPUTE statement to retain *just the key* needed to read the auxiliary input file record. Then the COMPUTE statement that actually refers to fields in the auxiliary input file should not need to use RETAIN. When the COMPUTE field is actually needed, the retained key will be enough to cause the correct record to be read for the COMPUTE statement.

Intermediate Computational Expressions

If your request uses a common computational expression in multiple statements, you may be able to improve performance by using an intermediate computation. Assign the value of the common part of the expression to an intermediate field. Then refer to that intermediate field name in each place where the common expression is needed. That way Spectrum Writer only has to compute the value of that expression once. It can then use that one result as many times as needed.

For example, assume that your request contains these three COMPUTE statements:

```
COMPUTE:  X = ((B - C) * 100) / C + 0.02
COMPUTE:  Y = ((B - C) * 100) / C + 0.09
COMPUTE:  Z = ((B - C) * 100) / C + 1.57
```

You may be able to improve performance by computing the common part of the expressions just once and saving the result in an intermediate field, like this:

```
COMPUTE:  TEMP = ((B - C) * 100) / C
COMPUTE:  X = TEMP + 0.02
COMPUTE:  Y = TEMP + 0.09
COMPUTE:  Z = TEMP + 1.57
```

Intermediate Conditional Expressions

If your request uses a common conditional expression in multiple places, you may be able to improve performance by using an intermediate expression. Assign the value of the common part of the expression to an intermediate bit field. Then use that intermediate field name in each place where the expression is needed. That way Spectrum Writer only has to compute the value of that expression once. It can then use that one result as many times as needed.

For example, assume that your request contains this conditional COMPUTE statement:

```
COMPUTE:  X = WHEN((A = B OR C > D) AND E = 1)   ASSIGN(1.23)
              WHEN((A = B OR C > D) AND E = 2)   ASSIGN(8.45)
              WHEN((A = B OR C > D) AND E = 3)   ASSIGN(0.29)
```

You may be able to improve performance by evaluating the common part of the conditional expressions (in the WHEN parms) just once and saving the result in an intermediate bit field, like this:

```
COMPUTE:  TEMP = WHEN(A = B OR C > D)  ASSIGN(#ON)
COMPUTE:  X = WHEN(TEMP AND E = 1)  ASSIGN(1.23)
              WHEN(TEMP AND E = 2)  ASSIGN(8.45)
              WHEN(TEMP AND E = 3)  ASSIGN(0.29)
```

Read Statements with the MULTI parm

In other parts of this manual, we discussed two speed–up tips involving READ statements that use the MULTI parm. We repeat them here:

> **Speed-Up Tip:** If you *know* that there will only be one qualifying record in an auxiliary input file for each READKEY value, do not specify the MULTI parm in your READ statement. Runs that use the MULTI parm are slower than runs that do not use it.

> **Speed-Up Tip:** If you have some READ statements that use the MULTI parm and some that do not, put the READ statement(s) *without* the MULTI parm ahead of the other READ statements (when possible). This may reduce the overall amount of I/O that Spectrum Writer has to perform.

For a detailed description of the program flow when MULTI-type READ statements are used, see page 592.

VSAM I/O

Direct (random) reads to VSAM files are inherently slow. A single random read may involve multiple EXCPs (to read different levels of index blocks and then data blocks). Since many 4GL report writers do not support direct reads to VSAM files at all, many users do not have a good standard to compare Spectrum Writer's VSAM I/O performance with.

When you write a Spectrum Writer job that does perform extensive random reads, it will run slower than a similar job that does not perform direct VSAM I/O. The inherent slowness of direct VSAM I/O is the cause, however, and not any additional overhead added by Spectrum Writer.

Here are some tips to make your VSAM jobs run as quickly as possible.

VSAM Buffers

When reading from VSAM files, you may be able to improve performance by increasing the number of VSAM buffers. This can increase the chances that VSAM will find a needed record already in one of its buffers, thus eliminating the need for a disk access.

Spectrum Writer provides parms that let you specify VSAM buffers right in your control statements (thus saving you from having to modify the execution JCL). Use the BUFND and BUFNI parms in your INPUT and READ statements to specify the number of buffers that VSAM should use.

The BUFND parm specifies the number of "data buffers" that the VSAM access method should maintain when processing the file. The BUFNI parm specifies the number of "index buffers" that the VSAM access method should maintain when processing the file. When these parms are not specified for a VSAM file, Spectrum Writer chooses a default number of data and index buffers for VSAM to maintain.

Different values for these parms are recommended for use in the INPUT statement and the READ statement. You may wish to experiment with these parms if you have long–running, VSAM–intensive jobs.

READ Statement Buffers

According to IBM's VSAM manual:

- Increasing the number of **data buffers** by 1 or 2 (from VSAM's default of 2) may improve performance for random reads. After that, more benefit is obtained by increasing the number of *index* buffers.
- Increasing the number of **index buffers** (from VSAM's default of 1) should improve performance for random reads up to a certain point. At some point, excessive paging may cancel any benefit. Optimal performance is sometimes achieved by having one index buffer for each level of the file's index.

Example

```
READ: EMPL-FILE  READKEY(EMPL-NUM)  BUFND(3)  BUFNI(6)
```

The above statement specifies that VSAM should allocate buffers for 3 data control intervals and 6 index control intervals when processing the EMPL–FILE.

INPUT Statement Buffers

According to IBM's VSAM manual:

- Increasing the number of data buffers to 4 or 5 (from VSAM's default of 2) may improve performance for sequential reads. At some point after that, excessive paging may cancel any benefit.
- Increasing the number of index buffers (from VSAM's default of 1) does not normally improve performance for sequential reads.

Example

```
INPUT: EMPL-FILE  BUFND(5)
```

The above statement specifies that VSAM should allocate buffer space for 5 data control intervals when processing the EMPL–FILE.

Pre–Sorting the Input File

Sometimes a vast improvement in performance can be achieved by pre–sorting the primary input file to Spectrum Writer. For example, assume we have a job that uses the SALES–FILE as the primary input file. Its records are in chronological order. Assume that we also use a READ statement to read an auxiliary input record from the EMPL–FILE. The READKEY is the EMPL–NUM from the SALES–FILE:

```
INPUT: SALES-FILE
READ:  EMPL-FILE  READKEY(EMPL-NUM)
```

Since the SALES–FILE is in chronological order, the EMPL–NUMs within it are presumably distributed randomly. Thus, Spectrum Writer may first have to read the EMPL–FILE record for key 036, then read a record for key 044, then read another record for key 036, etc. Since the reads are in random order, the odds are not good that VSAM will have the desired record already sitting in one of its buffers. Thus, it will have to perform real EXCP I/O to the VSAM file to get the desired record each time.

Now consider what would happen if we pre–sorted the SALES–FILE into EMPL–NUM order *before* having Spectrum Writer process it. The first SALES–FILE record might be for EMPL-NUM 036, for example. Spectrum Writer would then perform a read for key 036 to the EMPL–FILE. Then, the next SALES–FILE record would also be for key 036. That means VSAM

would find that record already in its buffer and would not have to perform any EXCPs to obtain it. All of the SALES–FILE records for EMPL–NUM 036 could be processed without any additional I/O to the EMPL–FILE. Then, when the SALES–FILE record for the next EMPL–NUM is read, the same thing would happen for it. VSAM might have to perform one I/O to get the correct EMPL–FILE record the first time, but then would not need to perform any more I/O for all the other SALES–FILE records with that same EMPL–NUM. The total number of slow, direct VSAM reads would be dramatically decreased.

Of course, pre–sorting the input file does add overhead to the overall job. Various factors, including the sizes of the primary input file and the auxiliary input file will determine whether the pre–sort saves you net execution time in the end. In many cases, it is worth the pre–sort. By the way, you can use a separate Spectrum Writer step to perform the pre–sort, if you like. This is explained on page 283.

KEYRANGE Parm

If the primary input file is a KSDS (keyed) VSAM file, you may be able to use the KEYRANGE parm in your INPUT statement to reduce the I/O required for the run. The KEYRANGE parm tells Spectrum Writer to read only those records within a certain range of keys, rather than reading through the entire VSAM file.

For example, assume that the input file for a run is a large KSDS customer file. The key for this file is a 2-byte state code followed by a 10-byte customer number. Assume we want a report that lists all of the male customers in New York. Normally, we might write:

```
INPUT:     CUSTOMER
INCLUDEIF: STATE = 'NY'   AND   SEX = 'M'
```

In the above example, Spectrum Writer must read through the entire CUSTOMER file, testing the STATE field and the SEX field in each record to determine which records to include in the report.

However, since the key to this file begins with the state code, we could write the following instead:

```
INPUT:     CUSTOMER   KEYRANGE('NY')
INCLUDEIF: SEX='M'
```

The above statements result in the very same report, but run much faster. Instead of having to read every record in the CUSTOMER file, Spectrum Writer can now jump in right at the first record whose key begins with NY. It then starts reading records sequentially from that point. And, after reading the last record whose key begins with NY, it stops reading the file altogether. This run is much faster because Spectrum Writer does not have to read the CUSTOMER records for all of the other states and perform the INCLUDEIF tests on them.

Notice that in the second run we also dropped the STATE='NY' test from the INCLUDEIF statement. Since the KEYRANGE parm guarantees that only records with NY in the STATE field are read, there is no need to test for that in the INCLUDEIF statement. Dropping this test provides an additional improvement in performance.

The syntax of the KEYRANGE parm is shown on page 547.

Use the STOPWHEN Parm for Non-Keyed Files

If your input file is not keyed, the KEYRANGE parm just discussed can't be used. In some cases, however, you may still be able to reduce unnecessary I/O by using the STOPWHEN parm. As long as your input file is sorted in a specific order, the STOPWHEN parm may be useful.

For example, assume that we have the same customer file described in the previous section, except that it is a sequential tape file (not a KSDS VSAM file). Assume that this tape file is already sorted by state code and customer number. To produce the same report described in the previous section, you could use these statements:

```
INPUT:     CUSTOMER  STOPWHEN(STATE > 'NY')
INCLUDEIF: SEX = 'M'
```

In the above example, Spectrum Writer knows that it can stop reading the input file as soon as it gets to a record whose state is greater than 'NY'. It will not have to read through the remainder of the input file. (Unlike with the KEYRANGE parm, however, Spectrum Writer *will* still have to read through all the records *before* 'NY'.)

The syntax of the STOPWHEN parm is discussed on page 551.

Replace an Auxiliary File with a "Table Lookup"

Since random I/O to auxiliary input files is slow, consider whether you can use a "table lookup" instead of reading a file. For example, assume that your primary input file contains 2–byte state codes. You want to print the entire state name in your report. One approach may be to write a READ statement that uses the state code as the read key for a STATE–FILE:

```
INPUT:   EMPL-FILE
READ:    STATE-FILE  READKEY(STATE)
COLUMNS: LAST-NAME  ADDR  CITY  STATE-FILE.STATE-NAME  ZIP
```

However, it will often be much faster to use a conditional COMPUTE statement to "look up" the state name (instead of reading a VSAM file):

```
INPUT:   EMPL-FILE
COMPUTE: NAME-OF-STATE =WHEN(STATE = 'CA')  ASSIGN('CALIFORNIA')
                        WHEN(STATE = 'NY')  ASSIGN('NEW YORK')
                        ...
                        WHEN(STATE = 'WY')  ASSIGN('WYOMING')
                        ELSE                ASSIGN(STATE)
COLUMNS: LAST-NAME  ADDR  CITY  NAME-OF-STATE  ZIP
```

The conditional COMPUTE statement above functions as a table lookup routine and eliminates the need for a READ statement.

In some cases, there will be too many potential lookup values for such a COMPUTE statement to be practical. Or, the number of entries may be constantly changing. In that case, you might still consider a combination of 1) a COMPUTE statement (to efficiently satisfy the most common cases), and 2) a READ statement to cover any cases missed by the COMPUTE statement:

```
INPUT:   EMPL-FILE
READ:    STATE-FILE  READKEY(STATE)
COMPUTE: NAME-OF-STATE =  WHEN(STATE = 'CA')  ASSIGN('CALIFORNIA')
```

```
                              WHEN(STATE = 'NY')  ASSIGN('NEW YORK')
                              ...
                              WHEN(STATE = 'WY')  ASSIGN('WYOMING')
                              ELSE                ASSIGN(STATE-FILE.STATE-NAME)
   COLUMNS: LAST-NAME  ADDR  CITY  NAME-OF-STATE ZIP
```

In the above example, whenever the STATE value is one that is covered by a WHEN condition, no read will be performed on the STATE–FILE. (That is because, even though a READ statement exists, no data from that file would actually be needed, and Spectrum Writer would not perform the read.) However, if a STATE is encountered which is *not* covered by any of the WHEN parms, the ELSE clause would assign the STATE–NAME field from the STATE–FILE. In that case (and only in that case) Spectrum Writer would need to perform the read to the VSAM file.

Clearing I/O Areas

When processing certain types of files, Spectrum Writer normally clears the entire I/O area to blanks before each read. This is to ensure that when a short record is read, it is not followed by leftover data from a previous longer record. For certain record layouts, such leftover data could cause misleading results. Specifying CLEAR(NO) (in the INPUT or READ statement) suppresses this clearing, which results in improved performance. You might want to specify CLEAR(NO) if you are certain that any leftover data in the I/O area will not affect your run.

Example

```
INPUT: PAYROLL-FILE  CLEAR(NO)
```

The above statement names the PAYROLL–FILE as the primary input file for the run. Spectrum Writer will not clear its I/O area each time it reads a record from that file.

> **Note:** You can also specify the CLEAR parm in the FILE statement to avoid having to put it in the INPUT and READ statements each time. The NOCLEARIO parm in the OPTIONS statement can be used to prevent clearing of *all files* in a run.

Fine-Tuning the Sort

For runs that involve sorting a large number of records, the sort process itself may account for a significant portion of the CPU usage. In such cases, you may be able to speed up your run by "fine-tuning" the sort process.

Spectrum Writer does not perform the sort logic itself. It simply calls your shop's standard Sort program (or the program named with the optional SORTNAME option). Check the manual for your Sort program to see if there are optional parms that you can specify to speed up the sort.

Here is a specific example. Large sorts run faster when the sort program knows ahead of time the approximate number of records it will be sorting. If you know the approximate number of records normally sorted in a particular run, try passing that information to the

sort program. Under OS/390, you can pass this information to programs like Syncsort by providing a special $ORTPARM DD in your JCL, like this:

```
//$ORTPARM DD *
FILSZ=E100000
```

The above parm tells Syncsort that it will be sorting approximately 100,000 records. (The "E" stands for estimated, and should be used unless you happen to know the exact number of records that will be sorted.)

Verify that your parm is being successfully processed by the Sort program by scanning the SYSOUT output. (Verify that your parm is listed and does not have any error messages associated with it.)

> **Note:** Another way to pass options to most sort programs is with Spectrum Writer's SORTOPT parm (in an OPTIONS statement, see page 574):
>
> ```
> OPTIONS: SORTOPT('FILSZ=E100000')
> ```

Another factor that can affect sort time involves the temporary work datasets used by the Sort program. You may be able to speed up large sorts by specifying more and/or larger work datasets in your JCL. In OS/390, this is usually done via SORTWKnn DD statements (see page 412). Again, check the manual for your Sort program for the specifics on how to do this.

Development Cycle

The process of developing new requests often entails making minor changes and re–running the request many times. If the input file you are using contains a million records, this can obviously take some time. The following options are available to help speed up your development runs. Once you are satisfied with your request, just remove the option to obtain your full production results.

OPTIONS TO SPEED UP DEVELOPMENT

OPTION	DESCRIPTION
MAXINPUT(nnnn)	Tells Spectrum Writer to *read* only the specified number of records from the input file. After reading that many records, Spectrum Writer acts as if it has hit EOF (end of file) on the input file and produces the final report or PC file. For example: `OPTIONS: MAXINPUT(500)`

OPTIONS TO SPEED UP DEVELOPMENT (CONTINUED)	
OPTION	**DESCRIPTION**
MAXINCLUDE(nnnn)	Tells Spectrum Writer to *include* only the specified number of records in the run. Here is how this option differs from the MAXINPUT option just described. You might specify MAXINPUT(500) and find that your report has no records in it at all. That may be because the records that pass your INCLUDEIF statement are not among the first 500 records in the file — they occur further along in the file. The MAXINCLUDE option tells Spectrum Writer to read as many records as necessary until it finds the specified number of records that can be included in the report. For example: `OPTIONS: MAXINCLUDE(500)`
MAXPAGES(nnnn) **MAXPRINT(nnnn)**	Tells Spectrum Writer to print only the specified number of pages or lines in the report and then stop. This option prevents you from getting a million page report by accident as you develop your report. For example: `OPTIONS: MAXPAGES(500)` If you use either of these options, also see the NOCHECK option (page 567).
DETAIL(nnnn)	Tells Spectrum Writer to print only the specified number of detail records *per control break*. Use this option to limit the size of your output, while still letting you verify the control break processing. For example: `OPTIONS: DETAIL(10)`

Using Explicit Literals in Conditional Expressions

> **Caution:** We do *not* recommend routine use of this technique. It sacrifices ease–of–use for improved performance. Therefore it makes it easier to introduce errors into your queries. It also makes them more difficult to maintain. Use this technique only if runtime speed is of paramount importance for a particular job.

Using explicit literals in your INCLUDEIF statement (or in your WHEN parm expressions) when testing non–character type fields may improve performance. That is because it saves Spectrum Writer from having to perform data conversion. Here are some drawbacks to this technique:

- You must know both the length and the exact format in which a field is stored in your input record in order to correctly write the explicit literal.
- If a subsequent modification to the record layout changes the field's length or type and you fail to correctly update the INCLUDEIF statement, you might unknowingly obtain wrong results.

- You may not be able to use the "greater than" and "less than" comparisons (as opposed to "equal" and "not equal" comparisons). That is because Spectrum Writer performs a byte–by–byte comparison of the EBCDIC contents of a field whenever it is compared to an explicit literal. Thus, a negative packed number (X'123D') would be considered greater than the hex literal X'123C', which is a positive packed number. Had the two fields been compared as packed fields, the opposite would be true (X'123C' would be greater than X'123D').

Consider the following INCLUDEIF statement:

```
INCLUDEIF: SALARY = 2000 AND BIRTHDATE = 12/31/1975 AND BEGIN-TIME = 14:00:00
```

If you use the above statement, you do not need to know how long each field is or how it is stored in the input record. Spectrum Writer automatically performs all conversions necessary to make the literals compatible with the data fields.

If you want to write the same INCLUDEIF statement using explicit literals, you would need to know that information. Let's assume the following:

- SALARY is a 4–byte packed field
- BIRTHDATE is a 3–byte packed Julian date
- BEGIN–TIME is stored as a fullword containing hundredths of seconds since midnight in binary format

Given the above, you could write the same INCLUDEIF statement using explicit literals as follows:

```
INCLUDEIF: SALARY = X'0002000C' AND BIRTHDATE = X'75365C' AND BEGIN-TIME = X'004CE780'
```

The above statement would execute more efficiently than the earlier INCLUDEIF statement that did not use explicit literals.

Again, using explicit literals like these defeats a prime feature of Spectrum Writer— it's ease of use. Thus, we don't recommend using this technique in routine cases.

Appendix H. Sample Data Exit Programs

Spectrum Writer has an exit "hook" available for calling user–written routines for fields that require specialized processing. Using these routines, called "data exit programs," is discussed in "How to Define a Field Created by a Data Exit" on page 357.

Data exit programs can written in Cobol, PL/1 or Assembler (or any other language that allows you to pass parms to Spectrum Writer in the manner it expects).

Following are two sample data exit programs, one written in Assembler and one in Cobol-II.

> **Note:** Spectrum Writer always runs in 24-bit addressing mode. Therefore, your data exit program will be called in 24-bit address mode and must return to Spectrum Writer in the same mode. Generally, that means that you should link-edit your exit program with the AMODE=24 and RMODE=24 parms.

Sample Assembler Data Exit Program

A sample data exit program written in Assembly language appears in **Figure 73** (page 668). This sample program performs five simple functions in order to illustrate data exit calls for each of the five types of data. The DXPARM value (from the FIELD statement) tells the exit program which function is desired when it is called. Use this sample program as a model for writing your own Assembler data exit programs.

> **Note:** If you would like a copy of this sample exit program, just call or e-mail us with your request. We will be happy to e-mail you a copy.

Note the $DX DSECT located near the end of the program. That DSECT shows the complete parm list that Spectrum Writer passes to all data exit programs. Specifically, when a data exit program is called by Spectrum Writer, register 1 will point to a fullword. That fullword will contain the address of the $DX DSECT parm list.

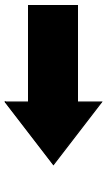
Figure 72 shows an actual run that uses this sample data exit program. In that run, five fields are defined as data exit fields. Notice the FIELD statements used to define those fields. Each statement has a TYPE parm that defines the field as a data exit type field (NUMEXIT, for example.) In each case, the name of the data exit program (the DXPROG parm) is the same. It is SWDEXIT, the name we chose for this sample exit program.

When processing a report request, Spectrum Writer will call SWDEXIT each time that it needs to process any of the 5 fields defined as data exit fields. Notice that each field has a different DXPARM value. The appropriate DXPARM value is passed to the exit program as part of the parm list whenever it is called (see $DXFLDPA.) That parm value tells SWDEXIT what function to perform, and thus, what value to return to Spectrum Writer.

> **Note:** there is no guarantee as to precisely when, in what order, or even *if*, Spectrum Writer will call the data exit for a given field in an input record. For example, if Spectrum Writer determines that a record will not be included in the report (and

These Control Statements:

```
INPUT: EMPL-FILE

* FOLLOWING STMTS DEFINE ADDITIONAL "EXIT" TYPE FIELDS FOR EMPL-FILE

FIELD: TESTNUM TYPE(NUMEXIT)   DXPROG('SWDEXIT') DXPARM('N') DXRETDEC(0)
FIELD: TESTDATE TYPE(DATEEXIT) DXPROG('SWDEXIT') DXPARM('D')
FIELD: TESTTIME TYPE(TIMEEXIT) DXPROG('SWDEXIT') DXPARM('T') DXRETDEC(0)
FIELD: TESTCHAR TYPE(CHAREXIT) DXPROG('SWDEXIT') DXPARM('R')
       COLUMN(LAST-NAME) LENGTH(15) DXRETLEN(15)
FIELD: TESTBIT  TYPE(BITEXIT)  DXPROG('SWDEXIT') DXPARM('B')
       COLUMN(DEPT-NUM)

COLUMNS: EMPL-NUM LAST-NAME TESTCHAR
         TESTNUM(7) TESTDATE TESTTIME DEPT-NUM(4) TESTBIT
```



Produce this Report:

```
THU  09/21/95   8:21 AM          DATA FROM EMPL-FILE                PAGE     1

EMPL      LAST                                                  DEPT
NUM       NAME          TESTCHAR      TESTNUM TESTDATE TESTTIME NUM   TESTBIT

036  JONES                    SENOJ       0 12/31/96 12:34:56    2 NOT TESTBIT
037  JOHNSON                NOSNHOJ       1 12/31/96 12:34:56    1 TESTBIT
039  JOHNSON                NOSNHOJ       2 12/31/96 12:34:56    2 NOT TESTBIT
040  MACDONALD             DLANODCAM      3 12/31/96 12:34:56    2 NOT TESTBIT
041  SIMPSON                  NOSPMIS     4 12/31/96 12:34:56    3 TESTBIT
042  MORRISON               NOSIRROM      5 12/31/96 12:34:56    3 TESTBIT
043  CHRISTOPHERSON   NOSREHPOTSIRHC      6 12/31/96 12:34:56    1 TESTBIT
044  BAKER                     REKAB      7 12/31/96 12:34:56    4 NOT TESTBIT
045  THOMAS                   SAMOHT      8 12/31/96 12:34:56    4 NOT TESTBIT



*** GRAND TOTAL (     9 ITEMS)          36
```

Remarks:

- This report uses five fields that are created by the data exit program named SWDEXIT.

Figure 72. A report that uses a data exit program

determines this without testing the data exit field), it may not call the data exit program at all for that particular record.

Note: In this example, we chose to write a single data exit program to support five different functions (and thus five different fields.) We could also have written five separate data exit programs— one for each field. Then, each FIELD statement would have named a different exit program (in the DXPROG parm). In that case, the DXPARM parm in the FIELD statement would not be needed. Each program would always perform its one single function. You can use whichever of these approaches you prefer.

.

```
SWDEXIT  TITLE '- SAMPLE SPECTRUM WRITER DATA EXIT'
*************************************************************************
*                     SAMPLE DATAEXIT PROGRAM                           *
*                                                                       *
*  ENTRY: R1  -- POINTS TO A FULLWORD WHICH CONTAINS THE ADDRESS        *
*                OF THE $DX DSECT                                       *
*  ENTRY: R13 -- POINTS TO A 18-FULLWORD SAVEAREA IN CALLERS PROGRAM    *
*  ENTRY: R14 -- RETURN ADDRESS WITHIN CALLER'S PROGRAM                 *
*  ENTRY: R15 -- CONTAINS THE STARTING ADDRESS OF THIS EXIT PROGRAM     *
*                                                                       *
*  THE VALUE OF THE FIELD STATEMENT'S DXPARM() PARM DETERMINES WHAT     *
*  VALUE THIS PROGRAM RETURNS WHEN IT IS CALLED.                        *
*                                                                       *
*  DXPARM: N =  RETURN A NUMERIC COUNTER VALUE                          *
*          B =  RETURN BIT VALUE OF THE LOW-ORDER BIT IN RECORD FIELD*
*          D =  RETURN A CONSTANT DATE (12/31/1996)                     *
*          R =  RETURN THE "REVERSED" CONTENTS OF A CHARACTER FIELD     *
*          T =  RETURN A CONSTANT TIME (12:34:56)                       *
*                                                                       *
*************************************************************************
SWDEXIT  START 0
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*
         STM   R14,R12,12(R13)     SAVE CALLERS REGS
         LR    R10,R15             USE R10 AS BASE REGISTER FOR EXIT
         USING SWDEXIT,R10         SET ADDRESSIBILITY FOR THIS EXIT
*
         ST    R13,OURSAVE+4       POINT OUR SAVE AREA TO CALLER'S SA
         LA    R15,OURSAVE         POINT TO OUR SAVEAREA
         ST    R15,8(R13)          POINT CALLER'S SAVEAREA TO OURS
         LR    R13,R15             LEAVE R13 POINTING TO OUR SAVEAREA
*
         L     R1,0(R1)            L R1 WITH ADDR OF PARM DSECT
         USING $DX,R1              ADDRESS CALLER'S PARM DSECT
*
         L     R2,$DXFLDPA         R2 -> DXPARM VALUE FROM FIELD STMT
```

Figure 73. Sample Data Exit Program Written in Assembly Language

```
*
*********************************************************************
* FOLLOWING LOGIC IS EXECUTED FOR FIELDS WITH DXPARM('N'), SUCH AS:
*
*   FIELD: TESTNUM  TYPE(NUMEXIT) DXPROG('SWDEXIT') DXPARM('N')
*          DXRETDEC(0)
*
* THIS SAMPLE EXIT PROGRAM SIMPLY RETURNS AN ASCENDING COUNTER VALUE.
*********************************************************************
*
         CLI   0(R2),C'N'          IS DXPARM 'N'?  (NUMERIC EXAMPLE)
         BNE   NOTNUM              B IF NOT 'N'
         L     R2,$DXRESAD         POINT R2 TO AREA TO PLACE RESULT
         ZAP   0(16,R2),COUNTER    RETURN THIS 16-BYTE PACKED NUMBER
         AP    COUNTER,=P'1'       INCREMENT COUNTER FOR NEXT CALL
         B     RETURN              FUNCTION 'N' HAS BEEN PERFORMED
*
*********************************************************************
* FOLLOWING LOGIC IS EXECUTED FOR FIELDS WITH DXPARM('D'), SUCH AS:
*
*   FIELD: TESTDATE TYPE(DATEEXIT) DXPROG('SWDEXIT') DXPARM('D')
*
* THIS SAMPLE EXIT PROGRAM SIMPLY RETURNS THE CONSTANT DATE 12/31/1996
*********************************************************************
NOTNUM   EQU   *
         CLI   0(R2),C'D'          IS DXPARM 'D'? (DATE EXAMPLE)
         BNE   NOTDATE             B IF NOT DXPARM('D')
         L     R2,$DXRESAD         POINT R2 TO AREA TO PLACE RESULT
         MVC   0(4,R2),=X'19961231' RETURN THIS 4-BYTE X'YYYYMMDD' DATE
         B     RETURN
*
*********************************************************************
* FOLLOWING LOGIC IS EXECUTED FOR FIELDS WITH DXPARM('B'), SUCH AS:
*
*   FIELD: TESTBIT TYPE(BITEXIT) DXPROG('SWDEXIT') DXPARM('B')
*          COLUMN(14)
*
* THIS SAMPLE EXIT PROGRAM SIMPLY RETURNS THE VALUE OF THE LAST BIT
*   IN THE BYTE IDENTIFIED BY THE FIELD STATEMENT'S COLUMN() PARM.
*   (IN THIS EXAMPLE, THAT'S THE LAST BIT OF THE BYTE IN COLUMN 14.)
*********************************************************************
NOTDATE  EQU   *
         CLI   0(R2),C'B'          IS DXPARM 'B'?  (BIT EXAMPLE)
         BNE   NOTBIT
         L     R2,$DXRESAD         POINT R2 TO AREA TO PLACE RESULT
         L     R3,$DXFLDAD         POINT R3 TO RAW DATA IN INPUT RECORD
         TM    0(R3),X'01'         IS THE LOWORDER BIT ON?
         BZ    BITOFF              NO - RETURN AN "OFF" VALUE
         MVI   0(R2),C'1'          YES - RETURN AN "ON" VALUE
         B     RETURN
BITOFF   EQU   *
         MVI   0(R2),C'0'          RETURN AN "OFF" VALUE TO SPECTRUM
         B     RETURN
*
*********************************************************************
* FOLLOWING LOGIC IS EXECUTED FOR FIELDS WITH DXPARM('R'), SUCH AS:
*
*   FIELD: TESTCHAR TYPE(CHAREXIT) DXPROG('SWDEXIT') DXPARM('R')
*          COLUMN(1) LENGTH(10) DXRETLEN(10)
*
* THIS SAMPLE EXIT PROGRAM REVERSES THE CHARACTERS IN A CHARACTER FIELD
*   IN THE RECORD.  IT USES THE FIELD STATEMENT'S LENGTH(NNN) PARM
*   TO KNOW HOW MANY BYTES TO REVERSE.
*********************************************************************
NOTBIT   EQU   *
         CLI   0(R2),C'R'          IS DXPARM 'R' (REVERSE CHAR EXAMPLE)
         BNE   NOTREVER
         L     R2,$DXRESAD         POINT R2 TO AREA TO PLACE RESULT
         L     R3,$DXFLDAD         POINT R3 TO RAW DATA IN INPUT RECORD
         LH    R4,$DXFLDLN         LENGTH OF FIELD TO REVERSE.
         AR    R3,R4               POINT R3 PAST CHAR FIELD IN RECORD
*
REVLOOP  EQU   *                   LOOP THRU FIELD BACKWARDS
         BCTR  R3,0                BACKUP 1 BYTE (POINTER TO REC FIELD)
         MVC   0(1,R2),0(R3)       MOVE 1 REVERSED BYTE TO RESULT AREA
         LA    R2,1(R2)            INCREMENT POINTER IN RESULT AREA
         BCT   R4,REVLOOP          CONTINUE THROUGH ALL BYTES
*
         B     RETURN
NOTREVER EQU   *
```

Sample Data Exit Program Written in Assembly Language (Continued)

```
*********************************************************************
* FOLLOWING LOGIC IS EXECUTED FOR FIELDS WITH DXPARM('T'), SUCH AS:
*
*   FIELD: TESTTIME TYPE(TIMEEXIT) DXPROG('SWDEXIT') DXPARM('T')
*          DXRETDEC(0)
*
* THIS SAMPLE EXIT PROGRAM SIMPLY RETURNS THE CONSTANT TIME 12:34:56
* (12*3600 PLUS 34*60 PLUS 56 = 45296 SECONDS)
*********************************************************************
         CLI   0(R2),C'T'          IS DXPARM 'T'? (TIME EXAMPLE)
         BNE   NOTTIME
         L     R2,$DXRESAD         -> RESULT AREA
         ZAP   0(16,R2),=P'45296'  RETURN 12:34:56 AS PL16'SECONDS'
         B     RETURN
*
NOTTIME  EQU   *
*
*********************************************************************
*   NOW RETURN TO SPECTRUM WRITER                                    *
*********************************************************************
RETURN   EQU   *
         L     R13,OURSAVE+4       RESTORE CALLER'S R13 (SAVE AREA PTR)
         LM    R14,R12,12(R13)     RESTORE CALLER'S REGS FROM HIS SA
         BR    R14                 RETURN TO report WRITER
*
*
OURSAVE  DC    18F'0'              OUR SAVE AREA
COUNTER  DC    PL4'0'              COUNTER IS 0 ON FIRST CALL.
*
*********************************************************************
*                                                                   *
*        $DX --  PARM DSECT FOR CALLING USER DATA EXITS.            *
*                                                                   *
*********************************************************************
$DX      DSECT ,                   DATA EXIT PARM DSECT
*
$DXNAME  DC    CL4'DATA'           NAME OF EXIT
$DXLEVEL DC    CL4'0001'           LEVEL NUMBER
$DXFUNC  DC    CL4'CONV'           FUNCTION
$DXFLDNM DS    CL50                FIELDNAME BEING PROCESSED
$DXFILNM DS    CL50                FILENAME OF FIELD BEING PROC'ED
$DXFLDAD DS    A                   ADDR OF FIELD'S DATA IN INPUT RECRD
$DXRECAD DS    A                   ADDR OF BEGINNING OF INPUT RECORD
$DXFLDPA DS    A                   ADDR OF FIELD'S DXPARM() TEXT
$DXFILPA DS    A                   ADDR OF FILE'S EXITPARM() TEXT
$DXRESAD DS    A                   ADDR WHERE EXIT SHOULD PUT RESULT
$DXFLDLN DS    AL2                 VALUE OF FIELD'S LENGTH(NNN) PARM
$DXFLDDP DS    AL2                 VALUE OF FIELD'S DEC(NNN) PARM
$DXFLDPL DS    AL2                 LENGTH OF $DXFLDPA PARM'S TEXT
$DXFILPL DS    AL2                 LENGTH OF $DXFILPA PARM'S TEXT
$DXRESLN DS    AL2                 VALUE OF FIELD'S DXRETLEN(NNN) PARM
$DXRESDP DS    AL2                 VALUE OF FIELD'S DXRETDEC(NN) PARM
*
*
*
         END   SWDEXIT
```

Sample Data Exit Program Written in Assembly Language (Continued)

Sample Cobol Data Exit Program

A sample data exit program written in Cobol-II appears below. This sample program performs a single function — it reverses the order of the bytes of a 5-byte field passed to it.

Note: If you would like a copy of this sample exit program, just call or e-mail us with your request. We will be happy to e-mail you a copy.

.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  COBDEXIT.
 AUTHOR.
*****************************************************************
*                                                               *
*  SAMPLE SPECTRUM WRITER DATA EXIT PROGRAM IN COBOL-II.        *
*  THIS EXIT REVERSES THE BYTES OF A 5-BYTE CHARACTER FIELD IN* 
*  THE INPUT RECORD.  IT PASSES THE RESULT BACK TO SPECTRUM     *
*  WRITER IN ITS RESULT AREA.                                   *
*                                                               *
*  NOTE: THIS PROGRAM MUST BE LKED'ED IN 24-BIT ADDRESS MODE    *
*        AND LOADED BELOW THE 16-MEG LINE TO WORK WITH          *
*        SPECTRUM WRITER.                                       *
*                                                               *
*****************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  IBM-370.
 OBJECT-COMPUTER.  IBM-370.
 INPUT-OUTPUT SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.

*****************************************************************
*  THIS IS THE WORKING STORAGE FOR THIS PARTICULAR SAMPLE       *
*  PROGRAM.  YOUR PROGRAM WILL PROBABLY USE DIFFERENT ITEMS.    *
*****************************************************************
 77 J             PIC S9(4) COMP.
 77 DATA-INDEX    PIC S9(4) COMP.
 77 RESULT-INDEX  PIC S9(4) COMP.


 LINKAGE SECTION.

*****************************************************************
*  COPY THIS STRUCTURE EXACTLY AS IT APPEARS BELOW.             *
*  THIS SHOULD BE THE FIRST ITEM IN THE LINKAGE SECTION.        *
*****************************************************************
 01 DXPARMS.
    05 DXNAME  PIC X(4).
    05 DXLEVEL PIC X(4).
    05 DXFUNC  PIC X(4).
    05 DXFLDNM PIC X(50).
    05 DXFILNM PIC X(50).
    05 DXFLDAD POINTER.
    05 DXRECAD POINTER.
    05 DXFLDPA POINTER.
    05 DXFILPA POINTER.
    05 DXRESAD POINTER.
    05 DXFLDLN PIC S9(4) COMP.
    05 DXFLDDP PIC S9(4) COMP.
    05 DXFLDPL PIC S9(4) COMP.
    05 DXFILPL PIC S9(4) COMP.
    05 DXRESLN PIC S9(4) COMP.
    05 DXRESDP PIC S9(4) COMP.
```

Sample Cobol Data Exit Program

```
      ****************************************************************
      *  FOR THIS SAMPLE PROGRAM, WE NEED TO ADDRESS THE FIELD IN    *
      *  THE INPUT RECORD AND THE RESULT AREA THAT SPECTRUM WILL     *
      *  LOOK IN TO FIND THE DATA WE PREPARE.                        *
      ****************************************************************
       01 INPUT-FIELD     PIC X(5).
       01 RESULT-AREA     PIC X(5).


      ****************************************************************
      *  IF DESIRED, YOU CAN ALSO PUT CHARACTER FIELDS HERE          *
      *  REPRESENTING: THE INPUT RECORD, THE FIELD'S EXITPARM TEXT,  *
      *  AND THE FILE'S EXITPARM TEXT.  USE DXRECAD, DXFLDPA, AND    *
      *  DXFILPA TO ADDRESS THEM (IN THE PROCEDURE DIVISION.)        *
      *01 INPUT-RECORD    PIC X(100).                                *
      *01 FIELD-EXITPARM  PIC X(10).                                 *
      *01 FILE-EXITPARM   PIC X(10).                                 *
      ****************************************************************


       PROCEDURE DIVISION USING DXPARMS.

      ****************************************************************
      *  ADDRESS THE 5-BYTE CHARACTER FIELD IN THE INPUT RECORD.     *
      ****************************************************************
           SET ADDRESS OF INPUT-FIELD TO DXFLDAD.

      ****************************************************************
      *  ADDRESS THE AREA WHERE SPECTRUM EXPECTS THE 5-BYTE RESULT.  *
      ****************************************************************
           SET ADDRESS OF RESULT-AREA TO DXRESAD.

      ****************************************************************
      *  NOW BUILD THE RESULT BY REVERSING THE BYTES OF THE FIELD.   *
      ****************************************************************
           MOVE 0 TO J.
           PERFORM 5 TIMES
               COMPUTE DATA-INDEX   = 5 - J
               COMPUTE RESULT-INDEX = 1 + J
               MOVE INPUT-FIELD (DATA-INDEX:1) TO
                    RESULT-AREA (RESULT-INDEX:1)
               ADD +1 TO J
           END-PERFORM.


      ****************************************************************
      *  THE RESULT IS READY FOR SPECTRUM WRITER TO USE.             *
      *  NOW EXIT BACK TO SPECTRUM WRITER.                           *
      ****************************************************************
           GOBACK.
```

Appendix I. I/O Exits

Spectrum Writer has an exit "hook" available for calling user-written I/O routines. Such "I/O exits" are useful for input files that require specialized processing. Examples of such files are:

- files that use a proprietary access method
- files whose records are encrypted

Spectrum Writer passes your I/O exit program all of the information it needs to be able to handle:

- sequential or keyed reads
- "multiple" (one-to-many) reads
- KGE and/or GENERIC keys
- KEYRANGE values
- DDNAME/DLBL value to use

Thus, if you code your exit program to handle all of these possibilities, your users will be able to use the exit-type file just like any other file with Spectrum Writer. That is, they can successfully use the KEYRANGE, MULTI, KGE, GENERIC and DDNAME/DLBL parms in the normal way within their INPUT and READ statements. To the end-users, your exit-type files will look just like any other file.

Spectrum Writer also passes your exit program an optional, user-defined parm containing up to 255 bytes of whatever information you choose. You can use this parm information to tell your exit program, for example, the kind of special processing it should perform.

How to Define an I/O Exit File

Use the IOEXIT parm in the FILE statement to define a file that will be handled in an I/O exit.

```
FILE: MY-FILE  IOEXIT('program' [,'parm'] [,TRACE])  LRECL(750)
```

Only a program name is required in the IOEXIT parm. The "parm" text is optional. Use it to pass constant parm information to your I/O exit. Use the TRACE parm when developing new I/O exits to see useful debug information in the control listing.

Besides the IOEXIT parm, the only other item required to define an I/O exit file is a maximum record length. In OS/390, you can specify this with a LRECL parm (as shown above) or omit it and use Spectrum Writer's default length. In VSE, you must use the ATTR parm, like this:

```
FILE: MY-FILE IOEXIT('program' [,'parm'] [,TRACE]) ATTR(EXIT,750)
```

When Is the I/O Exit Loaded?

The I/O exit for an input is loaded the first time that Spectrum Writer needs a record from that input. That same copy of the program is then called for all subsequent requests for that

input record. If Spectrum Writer never needs a record from a given input, the I/O exit for that input will not be loaded at all.

A separate copy of the exit program is loaded for each input. That means that if you use the same exit program for more than one input in a run (for example, in the INPUT statement and in a READ statement), Spectrum Writer loads two copies of the exit program.

When Is the I/O Exit Called?

The I/O exit for an input is called each time Spectrum Writer needs to obtain a record from that input. In other words, the exit is called at the same times that Spectrum Writer would, for a non-exit type input, issue its own I/O request. In addition, Spectrum Writer calls the I/O exit once at end-of-job time to allow the exit to perform any close processing it desires. Note that there is no separate call to the exit to perform "open file" processing. The exit should perform any required open logic the first time that Spectrum Writer calls it to obtain a record.

> **Note:** There is no guarantee as to precisely when, in what order, or even *if*, Spectrum Writer will call the I/O exit for a given input. For example, if Spectrum Writer determines that a record will not be included in the report (and determines this without requiring data from a given auxiliary input file), it may not call the auxiliary input file's I/O exit program at all (for that primary input record).

Following is a more detailed explanation of when Spectrum Writer reads records from different kinds of inputs.

For the **primary input** (named in the INPUT statement), Spectrum Writer simply calls the I/O exit repeatedly until the I/O exit indicates that there are no more records in the file. The I/O exit indicates this by setting the $IXRETCD field to H'4' when it has no more records to return to Spectrum Writer. For primary input files, Spectrum Writer always calls the I/O exit with the SEQ function (in $IXFUNC).

Auxiliary input files (those named in READ statements) are handled differently depending on whether or not the MULTI parm was also specified in the READ statement.

For **non-MULTI auxiliary inputs**, Spectrum Writer calls the I/O exit the first time it needs a field from a new auxiliary input. When subsequent fields from the same input record are needed, Spectrum Writer will not call the I/O exit again, since the record is already available for it to use. For non-MULTI inputs, Spectrum Writer calls the I/O exit a maximum of one time per primary input file record. (Spectrum Writer may call the I/O exit zero times if it does not need any fields from that auxiliary input for a particular primary input file record.) For non-MULTI auxiliary inputs, Spectrum Writer always calls the I/O exit with the KEY function (in $IXFUNC).

Processing is different for **MULTI-type auxiliary inputs**. In this case, each time Spectrum Writer reads a primary input file record, it calls the I/O exit repeatedly (with the same read key) until the exit indicates that there are no more records for that read key. The first call (for a given primary input record) will have a function of FRST. Subsequent calls (for the same primary input record) will have a function of NEXT. The I/O exit should indicate that there are no matching records (for FRST), or no *more* matching records (for NEXT), by setting $IXRETCD to H'4'. Once Spectrum Writer sees the return code of 4, it moves on to the next primary input file record.

Note: For simplicity, we have described the case of a request with a primary input file and a single MULTI-type auxiliary file. In cases where multiple MULTI-type auxiliary files are used, the exit is actually called repeatedly for each *logical combination* of primary input record and lower ranked auxiliary record(s).

Error Return Codes from the I/O Exit

For any type of input, the I/O exit can indicate to Spectrum Writer that an error condition exists which prevents the exit from "reading" records from the input file. The exit indicates this by setting $IXRETCD to H'12'. When Spectrum Writer sees a return code of 12 from an exit, it prints a file error message in the control listing (along with any message the I/O exit may have placed in the $IXERR field). Once a return code of 12 has been received from an I/O exit for an input, Spectrum Writer stops processing that input and does not call that I/O exit any more.

What Does Spectrum Writer Pass to the I/O Exit?

When the I/O exit is called, register 1 will point to a fullword containing the address of the $IX DSECT parm list. (The $IX DSECT is shown near the end of the sample program that begins on page 681.) The contents of the $IX DSECT will have been set correctly by Spectrum Writer, as described below. Register 13 points to an 18-fullword save area within Spectrum Writer which the I/O exit should use to save Spectrum Writer's registers. Register 14 contains the return address within Spectrum Writer. Register 15 contains the entry point address of the I/O exit.

Note: Spectrum Writer always runs in 24-bit addressing mode. Therefore, your I/O exit program will be called in 24-bit address mode and must return to Spectrum Writer in the same mode. Generally, that means you will link-edit your exit program with the AMODE=24 and RMODE=24 parms.

Note the $IX DSECT located near the end of the sample program. That DSECT shows the complete parm list that Spectrum Writer passes to all I/O exit programs. Following is a description of each item in that $IX DSECT.

CONTENTS OF SPECTRUM WRITER'S $IX DSECT

ITEM	DESCRIPTION
$IXNAME	This 4-byte character field always contains the constant value "READ" to identify the type of exit program being called.
$IXLEVEL	This 4-byte character field contains the constant value "0001" to identify the version level of this exit interface.
$IXFUNC	This 4-byte character field tells the exit program what function Spectrum Writer is requesting of it. The values for this field are: **SEQ** Read the next (or first) sequential record from the file. This function is used for any exit-type file used in an INPUT statement. **KEY** Read the record, if any, that corresponds to the key value (identified by the $IXKEYAD and $IXKEYLN fields). This function is used for any exit-type file used in a non-MULTI READ statement.

CONTENTS OF SPECTRUM WRITER'S $IX DSECT (CONTINUED)

ITEM	DESCRIPTION
	FRST Read the first record, if any, that corresponds to the key value (identified by the $IXKEYAD and $IXKEYLN fields). This function is used (in conjunction with NEXT) for exit-type files that have the MULTI parm in their READ statement. **NEXT** Read the next record, if any, that corresponds to the key value (identified by the $IXKEYAD and $IXKEYLN fields). This function is used (in conjunction with FRST) for exit-type files that have the MULTI parm in their READ statement. **CLOS** Perform any close processing that may be required. Spectrum Writer itself does not require any particular action for this call. This wrap-up call is provided in case your access method does require some type of close processing. Note that no CLOS call is made to files when either of these conditions exists: • no read requests were made to the file • the exit returned an error return code (12) to Spectrum Writer.
$IXRECNM	This 70-byte character field contains the record name of the input being processed. The record name is taken from the RECNAME parm of the INPUT or READ statement. If no RECNAME parm is specified, the record name defaults to the filename.
$IXFILNM	This 70-byte character field contains the filename of the file being processed.
$IXKEYAD	For requests that involve a read key (functions KEY, FRST and NEXT), this fullword contains the address of the key value to be used. The length of the key value is contained in the halfword field $IXKEYLN.
$IXPRMAD	This fullword contains the address of the parm text, if any, specified in the IOEXIT parm in the FILE statement. If no parm text was specified, this field contains hex zeros. The length of the parm text is contained in the halfword field $IXPRMLN.
$IXRECAD	This fullword contains the address of the I/O area that Spectrum Writer has reserved for the exit program to place the records that it reads for this file. The exit program should place its records here. The length of the area reserved for these records is contained in the halfword value $IXRECLN (and is determined by the LRECL parm in the FILE statement.) You can use the CLEARIO parm in the INPUT or READ statement to specify that this I/O area always be cleared (to hex zeros or to spaces) before each call to the I/O exit, or that it not be cleared at all.
$IXKRBAD	For primary input file requests (SEQ function) where a KEYRANGE parm was specified, this fullword contains the address of the beginning keyrange value to be used. The length of this value is contained in the halfword field $IXKRBLN.

CONTENTS OF SPECTRUM WRITER'S $IX DSECT (CONTINUED)	
ITEM	**DESCRIPTION**
$IXKREAD	For primary input file requests (SEQ function) where a KEYRANGE parm was specified, this fullword contains the address of the ending keyrange value to be used. The length of this value is contained in the halfword field $IXKRELN. If the user specified only a single value in the KEYRANGE parm, that value is used as both the beginning and the ending keyrange value. That is, $IXKRBAD and $IXKREAD will both contain the same address, and $IXKRBLN and $IXKRELN will both contain the same length.
$IXKEYLN	For requests that involve a read key (functions KEY, FRST and NEXT), this halfword contains the length of the read key value that is present at the address contained in $IXKEYAD. Note that Spectrum Writer does not perform any validity-checking on the readkey's length (since Spectrum Writer knows nothing about your file's structure). This length is simply the length of whatever read key field the user specified in the READ statement. Your exit program should determine whether the key length is a full key, a partial (generic) key, or an invalid key (too long) and should execute accordingly. If the key length is something that your exit program cannot handle, you should place an error message to that effect in $IXERR, set the return code ($IXRETCD) to 12 and return to Spectrum Writer. Spectrum Writer will print your error message for the user and stop processing the file.
$IXPRMLN	This halfword contains the length of the parm text, if any, (from the IOEXIT parm in the FILE statement) that appears at the address contained in $IXPRMAD.
$IXRECLN	This halfword contains the length of the I/O area reserved for the exit program at the address contained in $IXRECAD. This length is determined by the LRECL parm in the FILE statement.
$IXKRBLN	For primary input file requests (SEQ function) where a KEYRANGE parm was specified, this halfword contains the length of the beginning keyrange value that is present at the address contained in $IXKRBAD. Note that Spectrum Writer does not perform any sort of validity-checking on the length of the beginning keyrange value (since Spectrum Writer knows nothing about your file's structure).
$IXKRELN	For primary input file requests (SEQ function) where a KEYRANGE parm was specified, this halfword contains the length of the ending keyrange value that is present at the address contained in $IXKREAD. Note that Spectrum Writer does not perform any sort of validity-checking on the length of the ending keyrange value (since Spectrum Writer knows nothing about your file's structure).

CONTENTS OF SPECTRUM WRITER'S $IX DSECT (CONTINUED)	
ITEM	**DESCRIPTION**
$IXRETCD	This halfword must be set by the I/O exit program before it returns to Spectrum Writer after each call. The following list shows the valid values for $IXRETCD. If $IXRETCD contains any other value upon return to Spectrum Writer, an error message will print and no further access to the file will be attempted. **0** **Record read.** A record has been placed in the I/O area. (Or, for CLOS requests, the close processing, if any, has been performed.) **4** **No record is being returned.** Use return code 4 to indicate end-of-file (for SEQ requests) or record-not-found (for KEY, FRST and NEXT requests). **12** **Error.** Use this return code if you cannot process the file for any reason. Examples of this are: file is not available, key is wrong length, an I/O error occurred trying to process the file, parm information is invalid, etc. You should also place an error message indicating the exact error in $IXERR. That message will be printed in the control listing for the user to see. Once Spectrum Writer sees a return code of 12 for an input file, it does not attempt any further processing of that input.
$IXDDN	For OS/390, this 8-byte character fields contains the value of the DDNAME parm, if any, specified in the FILE, INPUT or READ statement for this file. For VSE, this field contains the DLBL/TLBL value (from the ATTR parm), if any, specified in the FILE, INPUT or READ statement for this file.
$IXMULTI	This 1-byte character field contains a Y if the user specified the MULTI ("multiple records per key") parm in the READ statement for this input.
$IXGEN	This 1-byte character field contains a Y if the user specified the GENERIC parm in the READ statement for this input.
$IXKGE	This 1-byte character field contains a Y if the user specified the KGE ("key greater or equal") parm in the READ statement for this input.
$IXUSER	This 50-byte, doubleword aligned area is available for the exit program to use any way it wishes. The area is initialized by Spectrum Writer to hex zeros before the first call. Thereafter, Spectrum Writer does not alter the contents of this field.
$IXERR	The I/O exit program should use this 60-byte character field for any messages it wishes to print in the control listing. Use this field to print error messages, warning messages, debug messages, etc. for the user. Spectrum Writer initializes this field to all blanks. Upon return to Spectrum Writer, if the first byte of this field is non-blank, Spectrum Writer prints the contents of this field as a Warning-level message in the control listing and blanks the field out again.

CONTENTS OF SPECTRUM WRITER'S $IX DSECT (CONTINUED)	
ITEM	**DESCRIPTION**
$IXUNUSD	This 50-byte area is reserved for future use and must not be used by the I/O exit program.

Most of the $IX fields are guaranteed to contain the same information on each call to the exit program. (A list of exceptions is shown below.) Knowing this can simplify the code you write. For example, the $IXRECAD value (that is, the address where your exit should put its record) will be the same for all calls to a particular input's I/O exit program. Thus, in the sample exit program, we used the $IXRECAD value on the first call to modify our RPL (to tell the RPL where to put the VSAM record during later GETs). We did not need to check on subsequent calls to see if the $IXRECAD value had changed.

For a given input's I/O exit, the only items in the $IX DSECT that might change from call to call are:

- the function code in $IXFUNC
- the return code, which is initialized to -1 (X'FFFF') by Spectrum Writer before each call.
- the error message area ($IXERR) is reset to blanks each time it is used.

What Does the I/O Exit Pass Back to Spectrum Writer?

Before returning to Spectrum Writer, the I/O exit program should do the following:

- set a valid return code in $IXRETCD. (The valid return codes are listed under the description of $IXRETCD in the section above.)
- when a return code of 0 is set (for any request other than CLOS), the exit must also place a record in the I/O area (pointed to by $IXRECAD, and for a length of $IXRECLN). Be careful not to move more than $IXRECLN number of bytes to this location. Doing so may cause unpredictable results or an ABEND. If you need a larger I/O area, re-run the job using a larger LRECL parm (OS/390) or ATTR parm record size (VSE).
- optionally, any message can be placed in $IXERR. This message will be printed in the control listing with a severity level of WARNING. The message must begin with a non-blank in the first byte.
- optionally, any information can be placed in $IXUSER and will be preserved between calls.

The I/O exit must not alter any other part of the $IX DSECT or memory areas pointed to by items in the $IX DSECT. The I/O exit must especially be careful not to write beyond the I/O area reserved for it (at $IXRECAD).

Warning: If your exit program never indicates EOF (via return code 4), Spectrum Writer will continue calling your exit program endlessly until the CPU time is

exceeded or the run ABENDs. To avoid this while developing new I/O exit programs, you may want to use the following option as a safeguard:

```
OPTION: MAXINPUT(1000)
```

The above statement tells Spectrum Writer to stop the run after 1000 primary records have been read (even if EOF has not yet been reached).

Sample I/O Exit Program

A sample I/O exit program written in Assembly language appears on the following pages. This sample program simply reads records from a normal KSDS VSAM file (our sample EMPL-FILE, as a matter of fact). Its purpose is to help illustrate how the exit program linkage and logical flow work. You can use this sample program as a model for writing your own I/O exit programs.

Note: If you would like a copy of this sample exit program, just call or e-mail us with your request. We will be happy to e-mail you a copy.

Here are some ideas that may help you when developing your own I/O exit.

- to prevent run-away jobs (caused by forgetting to return an EOF return code), start off using a MAXINPUT option, like this:

  ```
  OPTION: MAXINPUT(1000)
  ```

- specify TRACE in the IOEXIT parm, like this:

  ```
  FILE: MY-FILE IOEXIT('myprogram',TRACE) LRECL(500)
  ```

  The TRACE information in the control listing will help you see what is being passed to and from the IOEXIT, as well as the return code for each call. Once you have the basic flow working correctly, you can remove the TRACE parm since it produces a lot of output.

- you can have your exit put debug messages in the $IXERR field and they will appear in the control listing. Doing this instead of using TRACE may reduce the amount of output you have to wade through.

- by moving important "working storage" variables to the $IXUSER area at critical times, you can see (in the TRACE output) what values they had. If you need more room than this for debug information, request a larger I/O area (via the LRECL parm in the FILE statement). Then use the excess portion of the I/O area (beyond the actual record) to hold your debug values. The entire I/O area is printed in the TRACE output.

```
IOEXITA  TITLE '- SAMPLE SPECTRUM WRITER I/O EXIT'
*********************************************************************
*                                                                   *
*               SAMPLE I/O EXIT PROGRAM -- ASSEMBLY LANGUAGE        *
*                                                                   *
* THIS SAMPLE ASSEMBLER I/O EXIT READS RECORDS FROM A VSAM EMPLOYEE  *
* FILE AND PASSES THE RECORDS BACK TO SPECTRUM WRITER. IT CAN READ   *
* THE FILE EITHER SEQUENTIALLY OR RANDOMLY (USING KEYS).            *
*                                                                   *
* THIS SAMPLE EXIT PROGRAM IGNORES THE "GENERIC" AND "KGE" PARMS    *
* IN THE CALLING PARM INFO.                                         *
*                                                                   *
*  ON ENTRY TO THIS EXIT:                                           *
*     R1  -- POINTS TO A FULLWORD WHICH CONTAINS THE ADDRESS        *
*               OF THE $IX DSECT                                    *
*     R13 -- POINTS TO A 18-FULLWORD SAVEAREA IN CALLERS PROGRAM    *
*     R14 -- RETURN ADDRESS WITHIN CALLER'S PROGRAM                 *
*     R15 -- CONTAINS THE STARTING ADDRESS OF THIS EXIT PROGRAM     *
*                                                                   *
*  ON EXIT, THIS ROUTINE WILL HAVE SET:                             *
*    -THE RECORD TO BE PROCESSED (IF ANY) AT THE LOCATION SPECIFIED *
*     BY $IXRECAD (FOR A MAXIMUM LENGTH OF $IXRECLN).               *
*    -A RETURN CODE (IN $IXRETCD) AS FOLLOWS:                       *
*       0 -- NORMAL (WE RETURNED A RECORD TO BE PROCESSED)          *
*       4 -- EOF OR "KEY NOT FOUND"                                 *
*      12 -- ERROR CONDITION (FILE I/O ERROR, LOGICAL ERROR,        *
*            INVALID PARM, ETC.)                                    *
*    -OPTIONALLY (ON ERRORS) A MESSAGE IN $IXERR TO BE PRINTED IN   *
*     THE SPECTRUM WRITER CONTROL LISTING.                          *
*                                                                   *
*********************************************************************
IOEXITA  START 0
*
R0       EQU   0
R1       EQU   1
R2       EQU   2
R3       EQU   3
R4       EQU   4
R5       EQU   5
R6       EQU   6
R7       EQU   7
R8       EQU   8
R9       EQU   9
R10      EQU   10
R11      EQU   11
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15
*
         STM   R14,R12,12(R13)     SAVE CALLERS REGS
         LR    R10,R15             USE R10 AS BASE REGISTER FOR EXIT
         USING IOEXITA,R10         SET ADDRESSIBILITY FOR THIS EXIT
*
         ST    R13,OURSAVE+4       POINT OUR SAVE AREA TO CALLER'S SA
         LA    R15,OURSAVE         POINT TO OUR SAVEAREA
         ST    R15,8(R13)          POINT CALLER'S SAVEAREA TO OURS
         LR    R13,R15             LEAVE R13 POINTING TO OUR SAVEAREA
*
         L     R7,0(R1)            LOAD R7 WITH ADDR OF PARM DSECT
         USING $IX,R7              ADDRESS CALLER'S PARM DSECT
*
         CLC   $IXFUNC,=CL4'SEQ '  DOES CALLER WANT A SEQUENTIAL READ?
         BE    DOSEQ               YES - DO SEQUENTIAL IO LOGIC
```

Sample I/O exit program written in Assembly language

```
*
         CLC   $IXFUNC,=CL4'KEY '  DOES CALLER WANT A KEYED READ?
         BE    DOKEY               YES - DO KEYED IO LOGIC
*
         CLC   $IXFUNC,=CL4'FRST'  DOES CALLER WANT 1ST MATCHING KEY?
         BE    DOFIRST             YES - DO "FIRST" IO LOGIC
*
         CLC   $IXFUNC,=CL4'NEXT'  DOES CALLER WANT NEXT MACTHING KEY?
         BE    DONEXT              YES - DO "NEXT" IO LOGIC
*
         CLC   $IXFUNC,=CL4'CLOS'  DOES CALLER WANT TO CLOSE A FILE?
         BE    DOCLOSE             YES - DO CLOSE LOGIC
*
         MVC   $IXERR(22),=CL22'UNSUPPORTED FUNCTION: '
         MVC   $IXERR+22(4),$IXFUNC  SHOW THE FUNCTION
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*
*********************************************************************
*  DO SEQUENTIAL READ OF EMPLOYEE FILE                              *
*********************************************************************
DOSEQ    EQU   *
         CLI   SEQOPEN,C'Y'        HAVE WE OPENED THE SEQ ACB YET?
         BE    SEQISOPN            B IF YES - DON'T OPEN IT AGAIN
*
*********************************************************************
*  DO ONE-TIME STUFF ON FIRST CALL.  OPEN ACB AND MODIFY THE RPL.    *
*********************************************************************
         MVI   SEQOPEN,C'Y'        REMEMBER FILE HAS BEEN OPENED
         MVC   SEQNAME,$IXRECNM    SAVE NAME OF SEQ INPUT (FOR CLOSE)
*
         OPEN  SEQACB              OPEN THE ACB FOR SEQ I/O
         CH    R15,=H'4'           WAS OPEN SUCCESSFUL?
         BNH   SEQDORPL            YES - NOW PREPARE THE RPL
*
         MVC   $IXERR(25),=CL25'VSAM ERROR OPENING ACBSEQ'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
SEQDORPL EQU   *                   SEQSCB IS OPENED. MODIFY RPL ONCE
         L     R2,$IXRECAD         RECORD SHOULD GO HERE
         LH    R3,$IXRECLN         THIS MUCH ROOM AVAILABLE FOR RECORD
         MODCB RPL=SEQRPL,AREA=(R2),AREALEN=(R3)
*
         LTR   R15,R15             MODCB OK?
         BZ    SEQISOPN            YES - ONE-TIME STUFF DONE
*
         MVC   $IXERR(32),=CL32'VSAM ERROR DOING MODCB OF SEQRPL'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*********************************************************************
*  ONE-TIME STUFF HAS BEEN DONE.  GET NEXT SEQUENTIAL RECORD.       *
*********************************************************************
SEQISOPN EQU   *                   ONETIME STUFF DONE - DO GET
         GET   RPL=SEQRPL          READ RECORD INTO REC AREA
*
         LTR   R15,R15
         BZ    RETGOOD             IF WE GOT A RECORD, RETURN NOW
*                                   GET FEEDBACK TO SEE WHAT'S WRONG
         SHOWCB RPL=SEQRPL,AREA=(S,FEEDBACK),FIELDS=FDBK,LENGTH=4
```

Sample I/O exit program written in Assembly language (Continued)

```
*
         CLC   FEEDBACK,=F'4'      END-OF-FILE CODE ?
         BE    RETEOF              YES - RETURN INDICATING EOF
*
         MVC   $IXERR(26),=CL26'VSAM ERROR GETTING SEQRPL '
         MVC   $IXERR+26(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*
*********************************************************************
*  DO KEYED READ OF EMPLOYEE FILE                                   *
*********************************************************************
DOKEY    EQU   *
         CLI   KEYOPEN,C'Y'        HAVE WE OPENED THE KEYED ACB YET?
         BE    KEYISOPN            B IF YES - DON'T OPEN IT AGAIN
*
*********************************************************************
*  DO ONE-TIME STUFF ON FIRST CALL.  OPEN ACB AND MODIFY THE RPL.    *
*********************************************************************
         MVI   KEYOPEN,C'Y'        REMEMBER FILE HAS BEEN OPENED
         MVC   KEYNAME,$IXRECNM    SAVE NAME OF KEY INPUT (FOR CLOSE)
*
         OPEN  KEYACB              OPEN THE ACB FOR KEYED (DIRECT) I/O
         CH    R15,=H'4'           WAS OPEN SUCCESSFUL?
         BNH   KEYDORPL            YES - NOW PREPARE THE RPL
*
         MVC   $IXERR(25),=CL25'VSAM ERROR OPENING KEYACB'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
KEYDORPL EQU   *                   KEYACB IS OPENED -- PREPARE THE RPL
         L     R2,$IXRECAD         RECORD SHOULD GO HERE
         LH    R3,$IXRECLN         THIS MUCH ROOM AVAILABLE FOR RECORD
         L     R4,$IXKEYAD         THE KEY TO BE READ IS HERE
         LH    R5,$IXKEYLN         THIS IS THE LENGTH OF THE KEY
*
         MODCB RPL=KEYRPL,AREA=(R2),AREALEN=(R3),                      X
               ARG=(R4),KEYLEN=(R5)
*
*********************************************************************
*  ONE-TIME STUFF HAS BEEN DONE.  GET A KEYED RECORD.               *
*********************************************************************
KEYISOPN EQU   *                   ONE-TIME STUFF DONE - DO GET
         GET   RPL=KEYRPL          READ RECORD FOR KEY INTO REC AREA
*
         LTR   R15,R15
         BZ    RETGOOD             IF WE GOT A RECORD, RETURN NOW
*
*                                  GET FEEDBACK TO SEE WHAT'S WRONG
         SHOWCB RPL=KEYRPL,AREA=(S,FEEDBACK),FIELDS=FDBK,LENGTH=4
*
         CLC   FEEDBACK,=F'16'     RECORD NOT FOUND?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
*
         MVC   $IXERR(26),=CL26'VSAM ERROR GETTING KEYRPL '
         MVC   $IXERR+26(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*********************************************************************
*  DO READ-FIRST OF EMPLOYEE FILE                                   *
*********************************************************************
DOFIRST  EQU   *
         CLI   MULOPEN,C'Y'        HAVE WE OPENED THE MULTIACB YET?
         BE    MULISOPN            B IF YES - DON'T OPEN IT AGAIN
```

Sample I/O exit program written in Assembly language (Continued)

```
*
*********************************************************************
*  DO ONE-TIME STUFF ON FIRST CALL.  OPEN ACB AND MODIFY THE RPL.    *
*********************************************************************
         MVI   MULOPEN,C'Y'        REMEMBER FILE HAS BEEN OPENED
         MVC   MULNAME,$IXRECNM    SAVE NAME OF KEY INPUT (FOR CLOSE)
*
         OPEN  MULTIACB            OPEN THE ACB FOR MULTI READ I/O
         CH    R15,=H'4'           WAS OPEN SUCCESSFUL?
         BNH   MULDORPL            YES - NOW PREPARE THE RPL
*
         MVC   $IXERR(27),=CL27'VSAM ERROR OPENING MULTIACB'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
MULDORPL EQU   *                   MULTIACB IS OPEN -- PREPARE THE RPL
         L     R2,$IXRECAD         RECORD SHOULD GO HERE
         LH    R3,$IXRECLN         THIS MUCH ROOM AVAILABLE FOR RECORD
         L     R4,$IXKEYAD         THE KEY TO BE READ IS HERE
         LH    R5,$IXKEYLN         THIS IS THE LENGTH OF THE KEY
*
         MODCB RPL=MULTIRPL,AREA=(R2),AREALEN=(R3),                    X
               ARG=(R4),KEYLEN=(R5)
*
         LTR   R15,R15             MODCB OK?
         BZ    MULISOPN            YES - ONE-TIME STUFF DONE
*
         MVC   $IXERR(34),=CL34'VSAM ERROR DOING MODCB OF MULTIRPL'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*********************************************************************
*  ONE-TIME STUFF HAS BEEN DONE.  POINT AND GET 1ST RECORD.          *
*********************************************************************
MULISOPN EQU   *                   ONE-TIME STUFF DONE - DO POINT/GET
*
         POINT RPL=MULTIRPL        SET POINTER FOR DESIRED KEY
         LTR   R15,R15             OKAY?
         BZ    MULPNTOK            B IF POINT WAS OK
*
*********************************************************************
* I/O ERROR DOING POINT.  CHECK IT OUT.  (MAY JUST BE NOT FOUND)     *
*********************************************************************
         SHOWCB RPL=MULTIRPL,AREA=(S,FEEDBACK),FIELDS=FDBK,LENGTH=4
*
         CLC   FEEDBACK,=F'16'     RECORD NOT FOUND?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
*
         CLC   FEEDBACK,=F'4'      EOF?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
*
         MVC   $IXERR(29),=CL29'VSAM ERROR POINTING MULTIRPL '
         MVC   $IXERR+29(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*
         LTR   R15,R15             MODCB OK?
         BZ    KEYISOPN            YES - ONE-TIME STUFF DONE
*
         MVC   $IXERR(32),=CL32'VSAM ERROR DOING MODCB OF KEYRPL'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
```

Sample I/O exit program written in Assembly language (Continued)

```
*
MULPNTOK EQU   *                   POINT WAS OK - NOW GET FIRST REC
         GET   RPL=MULTIRPL        GET FIRST REC FOR CURRENT KEY
*
         LTR   R15,R15             GET THE RECORD?
         BZ    RETGOOD             IF WE GOT A RECORD, RETURN NOW
*
*                                  GET FEEDBACK TO SEE WHAT'S WRONG
         SHOWCB RPL=MULTIRPL,AREA=(S,FEEDBACK),FIELDS=FDBK,LENGTH=4
*
         CLC   FEEDBACK,=F'16'     RECORD NOT FOUND?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
         CLC   FEEDBACK,=F'4'      EOF?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
*
         MVC   $IXERR(28),=CL28'VSAM ERROR GETTING MULTIRPL '
         MVC   $IXERR+28(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*
*********************************************************************
*  WE GOT ANOTHER RECORD.  WE COMPARE IT'S KEY TO SEE IF IT IS A    *
*  MATCH FOR THE DESIRED READKEY.                                   *
*********************************************************************
DONEXT   EQU   *
*
*********************************************************************
*  ONE-TIME STUFF WAS DONE IN A PRIOR "READ FIRST" CALL.            *
*********************************************************************
         GET   RPL=MULTIRPL        GET NEXT SEQUENTIAL RECORD
*
         LTR   R15,R15             GET THE RECORD?
         BZ    NEXTOK              IF WE GOT A RECORD, CHECK IT'S KEY
*
*                                  GET FEEDBACK TO SEE WHAT'S WRONG
         SHOWCB RPL=MULTIRPL,AREA=(S,FEEDBACK),FIELDS=FDBK,LENGTH=4
*
         CLC   FEEDBACK,=F'4'      EOF?
         BE    RETNTFND            YES - RETURN INDICATING NOT FOUND
*
         MVC   $IXERR(35),=CL35'VSAM ERROR GETTING (NEXT) MULTIRPL '
         MVC   $IXERR+35(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
NEXTOK   EQU   *
         L     R2,$IXRECAD         RECORD SHOULD GO HERE
         LH    R3,$IXRECLN         THIS MUCH ROOM AVAILABLE FOR RECORD
         L     R4,$IXKEYAD         THE KEY TO BE READ IS HERE
         LH    R5,$IXKEYLN         THIS IS THE LENGTH OF THE KEY
*
         BCTR  R5,0                LENGTH MINUS 1 OF READKEY
         EX    R5,COMPKEY          SEE IF READKEY MATCHES RECORD KEY
         BE    RETGOOD             IF RECORD KEY MATCHES - RETURN REC
         B     RETNTFND            DOESN'T MATCH - RETURN "NOT FOUND"
*
COMPKEY  CLC   0(0,R2),0(R4)       COMPARE READKEY WITH RECORD KEY
*
*
```

Sample I/O exit program written in Assembly language (Continued)

```
*************************************************************************
*   CLOSE ONE OF SPECTRUM WRITER'S INPUTS                               *
*************************************************************************
DOCLOSE  EQU   *
         CLC   $IXRECNM,SEQNAME    IS THIS FOR THE SEQ ACB?
         BE    CLOSESEQ            B IF YES
*
         CLC   $IXRECNM,KEYNAME    IS THIS FOR THE KEYED ACB?
         BE    CLOSEKEY            B IF YES
*
         CLC   $IXRECNM,MULNAME    IS THIS FOR THE MULTI ACB?
         BE    CLOSEMUL            B IF YES
*
         MVC   $IXERR(31),=CL31'CLOSE REQUEST FOR UNKNOWN INPUT'
         B     RETERROR            RETURN WITH ERROR RETCODE
*
CLOSESEQ EQU   *
         CLOSE SEQACB
         B     RETGOOD
*
CLOSEKEY EQU   *
         CLOSE KEYACB
         B     RETGOOD
*
CLOSEMUL EQU   *
         CLOSE MULTIACB
         B     RETGOOD
*
*
*
*
*************************************************************************
*   RETURN TO SPECTRUM WRITER, AFTER SETTING CORRECT RETURN CODE.     *
*************************************************************************
RETGOOD  EQU   *
         MVC   $IXRETCD,=H'0'      INDICATE THE RECORD IS READY
         B     RETURN
*
RETNTFND EQU   *
RETEOF   EQU   *
         MVC   $IXRETCD,=H'4'      INDICATE EOF / KEY-NOT-FOUND
         B     RETURN
*
RETERROR EQU   *
         MVC   $IXRETCD,=H'12'     INDICATE LOGICAL/PHYSICAL ERROR
         B     RETURN
*
RETURN   EQU   *
         L     R13,OURSAVE+4       RESTORE CALLER'S R13 (SAVE AREA PTR)
         LM    R14,R12,12(R13)     RESTORE CALLER'S REGS FROM HIS SA
         BR    R14                 RETURN TO SPECTRUM WRITER
*
*
OURSAVE  DC    18F'0'              OUR SAVE AREA
*
FEEDBACK DS    F                   HOLDS FEEDBACK INFO FROM RPL
*
```

Sample I/O exit program written in Assembly language (Continued)

```
*
*********************************************************************
*   DATA READ FOR SEQUENTIAL FILE I/O                               *
*********************************************************************
SEQOPEN  DC    C'N'               FLAG - WHETHER SEQ ACB IS OPEN YET
SEQNAME  DC    CL70' '            SPECTRUM WRITER NAME OF SEQ INPUT
SEQACB   ACB   DDNAME=EMPLDD,     ACB FOR SEQUENTIAL IO TO EMPL FILE  X
               MACRF=(SEQ,KEY)
*
SEQRPL   RPL   ACB=SEQACB,        RPL FOR SEQUENTIAL IO TO EMPL FILE  X
               OPTCD=(KEY,SEQ)
*
*
*********************************************************************
*   DATA AREA FOR KEYED FILE I/O                                    *
*********************************************************************
KEYOPEN  DC    C'N'               FLAG - WHETHER KEY ACB IS OPEN YET
KEYNAME  DC    CL70' '            SPECTRUM WRITER NAME OF KEYED INPUT
KEYACB   ACB   DDNAME=EMPLDD,     ACB FOR KEYED IO TO EMPL FILE       X
               MACRF=(KEY,DIR)
*
KEYRPL   RPL   ACB=KEYACB,        RPL FOR KEYED IO TO EMPL FILE       X
               OPTCD=(KEY,DIR)
*
*
*********************************************************************
*   DATA AREA FOR MULTIPLE KEYS FILE I/O                            *
*********************************************************************
MULOPEN  DC    C'N'               FLAG - WHETHER MULTIACB IS OPEN YET
MULNAME  DC    CL70' '            SPECTRUM WRITER NAME OF MULTI INPUT
MULTIACB ACB   DDNAME=EMPLDD,     ACB FOR MULTI IO TO EMPL FILE       X
               MACRF=(KEY,SEQ)
*
MULTIRPL RPL   ACB=MULTIACB,      RPL FOR MULTI IO TO EMPL FILE       X
               OPTCD=(KEY,SEQ,GEN)
*
         EJECT
*********************************************************************
*                                                                   *
*        $IX --  PARM DSECT FOR CALLING USER I/O EXIT (FOR INPUT)   *
*                                                                   *
*********************************************************************
$IXDSECT DSECT ,                  IO EXIT (INPUT) PARM DSECT
$IX      DS    0D
*
$IXNAME  DC    CL4'READ'          NAME OF EXIT
$IXLEVEL DC    CL4'0001'          LEVEL NUMBER
$IXFUNC  DC    CL4'    '          FUNCTION (SEQ,KEY,FRST,NEXT,CLOS)
*********************************************************************
*  $IXFUNC CAN HAVE THESE VALUES ON ENTRY TO THE USER EXIT:         *
*    "SEQ "  -- RETURN THE NEXT (POSSIBLY FIRST) RECORD SEQUENTIALLY *
*               USED WITH EXIT-TYPE FILES NAMED IN THE INPUT: STMT.  *
*    "KEY "  -- RETURN THE RECORD (IF ANY) CORRESPONDING TO THE KEY  *
*               VALUE DESCRIBED BY $IXKEYAD AND $IXKEYLN            *
*               USED WITH EXIT-TYPE FILES NAMED IN A READ: STMT     *
*               WHICH DOES NOT CONTAIN THE "MULTI" PARM.            *
*    "FRST"  -- RETURN THE FIRST RECORD (IF ANY) CORRESPONDING TO    *
*               THE KEY VALUE DESCRIBED BY $IXKEYAD AND $IXKEYLN    *
*               USED WITH EXIT-TYPE FILES NAMED IN A READ: STMT     *
*               WHICH DOES CONTAIN THE "MULTI" PARM.                *
```

Sample I/O exit program written in Assembly language (Continued)

```
*    "NEXT"  -- RETURN THE NEXT RECORD (IF ANY) CORRESPONDING TO     *
*               THE KEY VALUE DESCRIBED BY $IXKEYAD AND $IXKEYLN     *
*               USED WITH EXIT-TYPE FILES NAMED IN A READ: STMT      *
*               WHICH DOES CONTAIN THE "MULTI" PARM.                 *
*    "CLOS"  -- SPECTRUM WRITER HAS FINISHED USING THIS FILE. EXIT   *
*               CAN PERFORM ANY CLOSE-UP LOGIC IT DESIRES, BUT NONE  *
*               IS REQUIRED BY SPECTURM WRITER.                      *
*               USED WITH ALL EXIT-TYPE FILES USED IN A RUN.         *
************************************************************************
$IXRECNM DS    CL70                RECNAME OF INPUT BEING PROCESSED
$IXFILNM DS    CL70                FILENAME OF FIELD BEING PROC'ED
         DS    0F                  ALIGN FOLLOWING TO FULLWORD
$IXKEYAD DS    A                   ADDR OF KEY VALUE (OR ZERO FOR SEQ)
$IXPRMAD DS    A                   ADDR OF PARM TEXT
$IXRECAD DS    A                   ADDR WHERE EXIT SHOULD PUT RECORD
$IXKRBAD DS    A                   ADDR OF KEYRANGE BEGIN KEY TEXT
$IXKREAD DS    A                   ADDR OF KEYRANGE END KEY TEXT
$IXKEYLN DS    AL2                 LENGTH OF KEY VALUE
$IXPRMLN DS    AL2                 LENGTH OF PARM TEXT
$IXRECLN DS    AL2                 LENGTH OF AREA RESERVED FOR RECORD
$IXKRBLN DS    AL2                 LGTH OF KEYRANGE BEGIN KEY TEXT
$IXKRELN DS    AL2                 LGTH OF KEYRANGE END KEY TEXT
$IXRETCD DS    AL2                 RETURN CODE FROM EXIT (TO S/W)
************************************************************************
*  $IXRETCD SHOULD BE SET TO ONE OF THE FOLLOWING VALUES BY THE EXIT: *
*    0 -- RECORD READ SUCCESSFULLY (FOR SEQ,KEY,FRST AND NEXT).       *
*         OR ,CLOSE LOGIC PERFORMED (FOR "CLOS" CALLS).               *
*    4 -- MEANS "FILE IS OK, BUT NO RECORD IS BEING RETURNED".        *
*         FOR SEQ CALLS, IT MEANS END-OF-FILE.                        *
*         FOR KEY AND FRST CALLS, MEANS NO RECORD EXISTS FOR THE KEY. *
*         FOR NEXT CALLS, MEANS NO MORE RECORDS EXIST FOR THE KEY.    *
*   12 -- MEANS THE FILE HAS A PHYSICAL OR LOGICAL ERROR AND IS NOT   *
*         USABLE. SPECTRUM WRITER SHOULD NOT ATTEMPT TO PROCESS THE   *
*         FILE FURTHER.                                               *
************************************************************************
```

```
$IXDDN   DS    CL8                 DDNAME/DLBL NAME
$IXMULTI DS    CL1                 Y/N "MULTI" PARM
$IXGEN   DS    CL1                 Y/N "GENERIC" PARM
$IXKGE   DS    CL1                 Y/N "KGE" PARM
         DS    0D                  ALIGN FOLLOWING TO DOUBLEWORD
$IXUSER  DS    CL50                USER AREA - INIT'ED TO X'00' ONCE
$IXERR   DS    CL60                ERROR MSG (SET BY USER EXIT)
$IXUNUSD DS    XL50                RESERVED
*
*
         END   IOEXITA
```

Sample I/O exit program written in Assembly language (Continued)

To Keep Your Manual Current,

Please File All Updates

Behind This Page.

Index

#

#ABS built-in function 475, 635
#AND built-in function 630
#ASCII built-in function 558, 631
#BEGMONTH built-in function 638
#BEGWEEK built-in function 638
#BEGYEAR built-in function 639
#COMDATE built-in field 626
#COMPRESS built-in function 631
#COMPUTES pseudo-record name 159, 501
#COUNTER built-in field 626
  use in BREAK statement 198, 200, 489
#DAY built-in function 631
#DAYNAME built-in field 625
  use in FOOTNOTE statement 175
  use in TITLE statement 55, 163
#DAYNUM built-in function 635
#DMY built-in function 640
#EBCDIC built-in function 562, 631
#EQUALS parm in SORT statement 597
#ERROR built-in function 642
#FORMAT built-in function 340, 463, 589, 632
#GRAND
  use in BREAK statement 207
#HHMMSS built-in field 627
#INCDATE built-in function 639
  example 266
#INCDATETIME built-in function 639
#INCDURATION built-in function 640
#INCTIME built-in function 641
#INDEX built-in function 635
#INT built-in function 635
#ISNUM built-in function 642
  example 257
#ITEM built-in field
  in COLUMNS statement 211
#ITEM1 through #ITEM9 built-in fields 626
  in COLUMNS statement 212
#ITEM-ENDING built-in field 625
  use in BREAK statement 198, 489
#ITEMS built-in field 626
  use in BREAK statement 182, 198, 200, 489
#JOBNAME built-in field 625
#LCASE built-in function 632
#LEAPYEAR built-in function 642
#LEFT built-in function 48, 632
#MAKEDATE built-in function 463, 474, 475, 640
#MAKENUM built-in function 274, 340, 463, 474, 475, 636
  use of 340
#MAKETIME built-in function 274, 463, 637, 641
#MAX built-in function 514, 637
#MDY built-in function 640
#MIN built-in function 637
#MINUTENUM built-in function 637
#MISSING built-in function 642
#MOD built-in function 637
#MONTH built-in function 633
#MONTHNUM built-in function 222, 637
#NUMWORDS built-in function 638
#OFF built-in function 450, 642
#ON built-in function 450, 643
#OR built-in function 633
#PAGENUM built-in field 626
  changing number of digits in page number 174
  use in FOOTNOTE statement 175
  use in TITLE statement 53, 55–56, 163, 604–??
#PARSE built-in function 633
#REALDATE built-in function 643
#RIGHT built-in function 634
#ROUND built-in function 638
#SECONDNUM built-in function 638
#SUBSTR built-in function 340, 634
  example 340
#TIME built-in field 55–56, 163, 175, 625
#TIME24 built-in field 163, 175, 625
#TODAY built-in field 53, 55–58, 163, 175, 604–??, 626
#TRANSLATE built-in function 633, 634
#UCASE built-in function 634
#XOR built-in function 635
#YEAR built-in function 635
#YEARNUM built-in function 222, 638
#YMD built-in function 640

*

A
  meaning of 228, 644
E
  meaning of 644
F
  meaning of 355, 644
  suppressing 566, 646
I
  meaning of 644

suppressing 577, 646
S
in total line at control break 182
meaning of 135, 136, 453, 645
using automatic scaling to suppress 454
U
meaning of 645
V
meaning of 645
suppressing 577, 646
Z
meaning of 645
suppressing 577, 646
S
in total line at control break 454
*PAGE
meaning of 446
? symbol
meaning in PICTURES 454, 456, 458, 573
@ symbol
meaning in PICTURES 454, 456, 458, 573

A

ABEND
for I/O errors 230, 569, 586
for normalization errors 550, 569, 587
in multiple report runs 291
ABS built-in function (see #ABS built-in function) 635
ABSDATE
data type 613
from CICS 613, 616
Absolute value
#ABS built-in function 635
ABSTIME
data type 616
Access method
used for input files (OS/390) 329
used for input files (VSE) 331
used for output file (OS/390) 417
used for output file (VSE) 431
Access, Microsoft
example 94, 99
ACCUM parm
in COLUMNS statement 128, 150, 273, 502
in COMPUTE statement 150, 273, 509
in FIELD statement 150, 273, 338, 346, 523
Accumulating
data for statistics, which columns 148, 502, 509, 523
Across
printing data across the page 219
Addition
adding days, weeks, months or years to a date 639
adding seconds, minutes or hours to a date-time 639
adding seconds, minutes or hours to a time 640, 641
how to perform 47, 473, 506
Address
formatting addresses 631
Addressing mode
of exit programs 666, 675
Alias
member name in copy library 363, 367, 648
use in COPY statement 517, 518
Alignment
of column headings in Web reports 306, 316
of columns in multi-line reports 151
of data in report columns 146, 505
of text and graphics in Web reports 305, 312, 323
of titles (left, center and right) 55, 153, 168, 603, 606
of titles in Web reports 298–300
of titles, default 53
of totals in Web reports 303, 306, 316
title doesn't look centered 168, 173, 606
title doesn't look right aligned 173, 606
(see also Justification)
Alphabetical order
of columns for all fields from a file 159
Alphabetizing
columns by field name 159, 503
the report 62, 595
Alternate index 232
AM
showing AM and PM 347, 622
Ambiguous field name
among DB2 column names 403, 405
error indicator (***A***) 644
using record name to resolve 77, 226, 228, 550, 589
Ampersand (&)
meaning in conditional expressions 466
AND built-in function (see #AND built-in function) 630
AND keyword
use in conditional expressions 42, 465, 471
Animation, in Web reports 294
Apostrophes (') 448

(see also Quotation marks)
Approximate
values, using to save space 454
Arithmetic operations
between different types of numeric fields 337
how to perform 47, 472, 506, 513
Arrays
how to define 243, 355
how to process 237–257
in COBOL record layouts 238, 327, 355, 377, 495
nested 244
normalizing 237–248, 548, 585
only in some record types 247, 549
parallel arrays 245
ASC parm
in DB2 ORDERBY parm 550, 587
in SORT statement 597
Ascending
order, data fetched from DB2 399
order, in SORT statement 62, 597
ASCII
converting to EBCDIC 631
files, fixed format 279
files, how to create 143, 276, 283
instead of EBCDIC 631–632
parm, in #FORMAT function 632
parm, in BREAK statement 191, 490
parm, in COLUMNS statement 128, 143, 283, 502
parm, in TITLE statement 167, 605
spaces between fields 490
(see also #ASCII built-in function)
ASCIITABLE parm
in OPTIONS statement 558
ASM statement 479
how to use 369
scope of 384
syntax 480
ASMLIB DD 382, 517
ASMLIB parm
in OPTIONS statement 383, 518, 559
Assembly language
character versus numeric fields 387
converting to FIELD statements 378
copying from Panvalet or Librarian 383
copying record layouts 382, 517
date and time fields 375, 379
decimal digits 388
default copy library (OS/390) 382
default copy library (VSE) 383, 559
DSECT statement 384, 390, 497
EQU statement 390
expressions supported 388, 389
fields, changing the column heading 381
record layouts, names assigned 377, 390
record layouts, starting column 384
record layouts, using 369, 421, 438, 479
repetition factors 377, 495
sublibrary and member type copied (VSE) 518
SYSLIB library 363, 516
writing a data exit program in 666
ASSIGN parm
in COMPUTE statement 50, 509
Asterisks (*)
S appears in total line 182
S, meaning of 135, 136, 453
in column one for comment lines 445
in total line at control break 71, 181, 206
meaning in COLUMN and DISP parm 352, 524
meaning of /* and */ 445
meaning of all error indicators (e.g. ***S***) 160, 644
multiplication symbol 47, 473
printing bar graphs 154
suppressing error indicators 646
(see also under ***A***, ***I***, etc.)
ATTR parm
in FILE statement 331–332, 432, 532
in INPUT statement 332, 433, 532, 544
in READ statement 332, 433, 532, 580
AU files 308
Audio clips, putting in report 294, 308
AUTOSORT parm
in OPTIONS statement 62, 559
Auxiliary input files
(see Files and READ statement)
AVERAGE (AVG) parm
in BREAK statement 67, 186, 196, 207, 483
in BREAK statement print expressions 191, 194, 200
in BREAK statement, two different uses 194
in SORT statement 186, 597
Averages
average line (see also Statistical lines) 67, 186, 483, 597
display format used for 140
excluding zero values 186, 485, 491, 598
how many decimal digits 140, 151
how to compute 512
how to print 67, 186, 191, 483, 597
printing at Grand Totals time 207, 483
printing in total line 191, 491
which columns receive 148, 339, 502, 509, 523

(see also AVERAGE parm and NZAVERAGE parm)
AVI files 308

B

Backing up
current location, when defining fields 352, 524
Bar
character (|) (see Vertical bar)
Bar graphs
BARGRAPH display format 619
how to print bar graphs 154
Base
base-2 and base-10 scaling 454, 458
BASIC language
"IF" statement 42, 540
PRINT USING equivalent 451
Batch type files
normalizing 247
processing 234
BCD data
date fields 612
numeric fields 610
time fields 614, 615
Before
report, putting lines 309, 318, 572
Beginning
of control group, printing lines at 200, 484, 486
of report, printing headings once 207
of report, putting lines before 309, 318, 572
BEGMONTH (see #BEGMONTH built-in function) 638
BEGWEEK (see #BEGWEEK built-in function) 638
BEGYEAR (see #BEGYEAR built-in function) 639
Big
biggest of several numbers, dates or times 637
literals, how to write 444
making a column bigger 60, 135, 505
numbers, scaling to fit in small column 454
records in input file 330, 535, 547, 584
report lines, how to produce 432
report width 432
(see also Width)
Billions
rounding to 453, 638
Binary data
comparing to packed data 337
writing to output file 282, 619
BINARY data type
needed in read key 589
numeric field 610
times stored as 614, 615
BINARY display format 619
BINARYUN
data type, for time fields 614, 615
display format 620
numeric data type 610
writing binary unsigned data to output file 620
Bind
DB2 plan name 560
BIT data type 616
Bit fields
bit field conditions 465, 471
bit literals 450
creating your own 450, 506, 514, 642, 643
data type 616
effect on default location in record 352
how bits are numbered 349, 523
how formatted in reports 349, 529, 618
how sorted 601
how to define 347, 521
logical operations 630, 633, 635
testing multiple bits 630
testing value of 465, 471, 514, 630
BIT parm
in FIELD statement 349, 523
BITEXIT
data type 616
BITS display format 618
BIZ parm
for fields printed at control breaks 192, 490
in COLUMNS statement 129, 140, 257, 503
in TITLE statement 167, 605
Blank
ASCII spaces between fields 490
in first column of control statement 444
inserting blank columns in PC files 110
padding 462, 470, 511
removing leading blanks 631
removing trailing blanks 631
spaces between report columns 128, 502
spaces between report columns in Web reports 301
spaces, required around minus sign 473
spaces, where allowed in control statement 443
suppressing blanks between fields in output files 560
Blank lines
between report lines 153, 573
in PC files 110
in report titles 153
printing after the total line 67, 69, 178, 486, 598

printing at control breaks 196, 486, 598
printing before the total line 184
printing in report body 499
suppressing 253, 257, 573
suppressing, before Grand Totals 279, 283, 567
Blanking out
all column headings 133, 134, 175, 567
individual column headings 132, 504
leading zeros 455
numbers, dates and times that are zero 129, 167, 192, 257, 490, 503, 605, 632
repeating values 129, 144, 505
the final "S" to form the singular 198, 489, 625
Blinking font, specifying 294, 322
BLKSIZE 331, 332, 533, 567, 570
Bold
font, specifying 294, 296, 298, 301, 303, 322
Boolean
expressions (see conditional expressions)
fields (see Bit fields)
Both
of two conditions are true 42, 465, 471
Bottom of report
margin 154, 571
printing footnote lines 175, 538
putting lines after 309, 318, 572
BREAK statement 481
break occurs at wrong place 600
built-in fields available 198
control break spacing 67, 69, 178, 486
control break spacing, summary reports 576
customizing the total line at control breaks 182, 198, 487
formatting dates, times and numbers 192, 491
how to use 65, 177
how to use with PC files 108
in Web reports 303, 308–309
justification parm in print expressions 192, 491
order of BREAK statements 204
parms (see under name of parm)
print expressions 189, 488
printing a certain number of detail lines per break 561
printing a field's total or average value 193, 491
printing a total line at control breaks 487
printing averages at control breaks 67
printing custom lines at control breaks 188, 483
printing lines at beginning of a control group 200, 484
printing statistical lines at control breaks 67, 186, 483
printing the number of items in control group 182, 198, 489
printing the number of items included in the report so far 198, 489
requesting multiple control breaks 69, 204, 210
resetting page number 180
skipping to new page 67, 178, 486
spacing factor in print expressions 191, 490
suppressing the total line at control breaks 185, 487
syntax 482
using a PICTURE to format numeric data 194, 451
using to customize the Grand Totals 207, 483
where to put 204, 445
width of items in lines printed at control breaks 182, 193, 491
(see also Control breaks)
Breaking down
totals 217
Buffer
for input files, specifying in JCL 423, 433
for reading input files 330, 331, 355, 533, 535, 545, 547, 567, 581, 584
speed-up tips 658
with VSAM I/O 544, 581, 658
BUFND parm
in INPUT statement 544, 659
in READ statement 581, 659
speed-up tips 659
BUFNI parm
in INPUT statement 544, 659
in READ statement 581, 659
speed-up tips 659
Built-in fields 624
available in BREAK statement 198, 489
available in TITLE and FOOTNOTE statements 53, 163, 604
(see also under name of built-in field)
Built-in functions 628
use in COMPUTE statement 48, 475, 514
(see also under name of built-in function)
BYCOL parm
in COLUMNS statement 159, 503
BYDEF parm
in COLUMNS statement 159, 503
BYNAME parm
in COLUMNS statement 159, 503
Byte
ASCII versus EBCDIC 558, 562, 631, 632
bits in 349, 523

C

Calculations
  how to perform 47, 472, 506
  using different types of numeric fields 337
Capital letters (see Case)
Carriage control character
  allowing for in LRECL parm 431
  suppressing 279, 415, 567
Case
  lower case 632
  sorting mixed case fields 634
  upper case 634
Category
  totalling a field by 217
CENTER parm
  in #FORMAT built-in function 632
  in BREAK statement 192
  in COLUMNS statement 129, 146
  in TITLE statement 168, 173, 606
Centering
  CENTER parm needed in centered titles 606
  column headings 132
  data in report columns 146, 505
  data in titles, looks wrong 168, 173, 606
  items in control break lines 192, 491
  titles 53, 55, 168, 603
  titles, in Web reports 298, 322
  (see also Alignment and Justification)
Cents
  rounding to whole dollars 339, 512, 638
Century
  day in century 640
  which century for YY dates 269, 559
CENTURY parm
  in OPTIONS statement 269, 559
Chaining input files 226
Changing
  translating characters 634
CHARACTER
  data type 609
  display format 618
  display format, use in FORMAT option 563
Character fields
  ASCII versus EBCDIC 632
  changing case 632, 634
  comparing 40, 449, 462, 470, 513
  comparing to numeric fields 463
  converting to date 640
  converting to numeric 340, 463, 636, 642
  converting to time 274, 641
  counting words in 638
  creating your own 48, 449, 506, 514
  how sorted 600
  how to define 333, 521
  list of data types 609
  maximum size 511
  numeric data in 339, 450
  parsing words from 633
  scanning for a text 462, 471, 635
  substrings 634
  translating characters 634
  which contain numeric data 642
  writing character literals 42, 448, 470, 472
Character operations
  how to perform 48, 473, 506
Characters
  ASCII versus EBCDIC 143, 558, 562, 631
  which ones allowed in file and field names 446
CHAREXIT
  data type 609
Charts
  in Web reports 294, 305
CICS
  ABSDATE value 613, 616
  downloading from (VSE) 432
CLEAR parm
  in INPUT statement 545, 662
  in READ statement 581, 662
Clicking mouse, in Web reports 308
COBLIB DD 382, 517
COBLIB parm
  in OPTIONS statement 383, 518, 559
COBOL
  "IF" statement 42, 459, 540
  arrays, how to process 238
  ASSIGN clause, FD, and record structure 327
  converting to FIELD statements 378
  copybook library 363, 516
  copying from Panvalet or Librarian 383
  copying record layouts 382, 517
  date and time fields 375, 379
  default copy library (OS/390) 382
  default copy library (VSE) 383, 559
  EXAMINE (see #TRANSLATE)
  fields, changing the column heading 381
  FILLER 386
  Julian dates 375
  level 01 REDEFINES 384, 386, 497
  level indicators 385, 386
  OCCURS clause 249, 255, 327, 377, 495
  record layouts, names assigned 377, 386
  record layouts, starting column 384

record layouts, using 369, 421, 438, 493
REDEFINES clause 352, 368, 385, 524
sequence numbers 372, 495
SIGN IS SEPARATE clause 387
slack bytes 243
sublibrary and member type copied 518
SYNCHRONIZED parm 243
UNSTRING (see #PARSE)
writing a data exit program in 671
COBOL statement 493
how to use 369
scope of 384
syntax 494
Codes
completion 230, 425, 439, 549, 569, 586
COLHDGONCE parm
in OPTIONS statement 133, 175, 278, 560
Collating order 600
Colon (:)
after statement name 34, 443
changing delimiter for formatting times 576
use as a relation operator 461
Colors
different colors in one column 315
list of HTML colors 322
specifying 294, 298, 319, 322, 323
COLSEP parm
in OPTIONS statement 156, 278, 560
COLSPACE parm
in OPTIONS statement 128, 282, 315, 560
Column
field's starting column in record 350, 352, 353, 524
in control statement, when first one blank 444
in control statement, when first one contains asterisk 445
in control statement, which ones to use 443
printing titles in a specific column 173
starting, in COBOL and Assembly record layouts 497
COLUMN (COL) parm
in ASM & COBOL statements 381, 494
in FIELD statement 350, 352, 353, 524
Column (in report)
order in which they appear 159
scaling big numbers to fit 454
Column headings
aligning in Web reports 306, 316
blanking out individual ones 132, 504
effect of dash and underscore in name 504
for computed fields 47, 511
for literal columns 133, 501
how to change 60, 129, 130
how to justify (left, center and right) 132
in FIELD statement 326
in multi-line reports 130, 133, 153, 254, 566
in Web reports 301, 306, 316, 564, 565
making shorter 130, 135
one-line headings 133, 279, 564
options, summary 133
parm in COLUMNS statement 504
printing just once 133, 134, 175, 278, 560
running down the page 219
specifying when defining a field 326, 350, 527
specifying with TITLE statements 169
splitting onto multiple lines 130, 133, 350, 504, 527
suppressing all 93, 133, 134, 175, 279, 567
suppressing the underscore line 132–134, 504, 568
truncation of 135
use of vertical bar (|) 130, 132, 133, 563
using field name as 130
when suppressed 566
(see also Titles)
COLUMNS statement 498
#COMPUTES keyword 159, 500, 501
#ITEM built-in field 211
advanced features 125
all blank 153, 499
column headings 60, 129, 130, 133, 153, 504
column headings, suppressing all 567
columns look skewed 151
excluding fields from output 158, 159, 504, 505
formatting dates, times and numbers 58, 129, 137, 503
in Web reports 306
including all fields from a file 158, 501
justification within columns 129, 146, 505
literal columns 126, 133, 153, 448, 501
multiple statements 151, 254, 498, 566, 575
order of all fields from a file 159, 503
parentheses 128
parms allowed in 128, 499
printing certain characters between report columns 560
printing full-page forms 153
printing line numbers 211
printing variable number of lines per input record 249
quotation marks, apostrophes 126, 501
repeating values, suppressing 129, 144
shifting report right 154, 181
spacing between columns 128, 151, 502

spacing between report lines 573
syntax 499
truncating a column 135
using a PICTURE to format numeric data 137–139, 451
using record name to resolve ambiguous field name 77, 226, 228, 550, 589
where to put 445
which columns are totalled 128, 148, 339, 502
width of columns 60, 129, 135, 182, 505
writing all fields to PC file 88
Combining character fields 48
COMDATE built-in field (see #COMDATE built-in field) 626
Comma (,)
as delimiter in output files 278, 560
in control statements 443
in number, using a different character 452
in numbers, whether to print 139, 278, 339, 452, 619, 623
not allowed in numeric literals 42, 449
unwanted commas in numbers 339
used to separate parms 128, 167, 191
using dot instead of comma for numbers 140, 619
Comma delimited file
converting entire mainframe file into 88
Comments
how to write 445
HTML comments 324
in SWALIAS member 368
within scope of ASM and COBOL statements 385
COMP (see BINARY)
COMP-1 (see BINARY)
COMP-3 (see PACKED)
Comparing
a field to hexadecimal value 464, 472
character fields 40, 340, 449, 450, 462, 470
date fields 42, 341, 449, 462, 470
how to write conditional expressions 459
numeric fields 42, 337, 340, 449, 450, 461, 470
operands of different length 462
operands of different types 340, 450, 463
time fields 44, 344, 450, 462, 470
when to use quotation marks 42, 340, 450, 470
Completion codes 230, 425, 439, 549, 569, 586
for empty/non-empty runs 562, 568
for exception reports 568
Complex
conditional expressions 465
COMPRESS built-in function (see #COMPRESS built-in function) 631
Computational expressions 472
bit, how to write 476
character, how to write 48, 449, 472
date, how to write 449, 475
examples 474, 513
list of built-in functions 628
numeric, how to write 46, 449, 472
order of evaluation 474
speed-up tips 657
syntax 472
time, how to write 273
use of parentheses in 474
(see also COMPUTE statement)
COMPUTE statement 506
assigning different values based on conditions 50, 508
column headings for computed fields 47, 60, 130, 511
computing an average 512
computing true ratios, percentages 202, 474, 510, 515
concatenation operation 48, 473, 514
conditional 50, 508
conditional, example 214, 217, 234
converting character to numeric data 340
converting data to different type 632, 636, 640–641
converting numeric to character data 340
creating a read key for READ statement 79, 589
creating bit fields 450, 476, 514
creating character fields 48, 449, 474, 513
creating date fields 269, 449, 475, 513
creating numeric fields 46, 449, 474, 513
creating time fields 272, 450
data type of result 508
default value assigned 50, 508
division by zero 577
examples 449–450, 474, 513, 630
hexadecimal values 514
how dates, times and numbers are formatted 510
how many decimal digits 512
how to use 46, 98
in file definitions 159
in Web reports 306
justifying the result (left, right, center) 632
keeping in copy library 159, 363
list of built-in functions 628
math operations 46, 473, 513
order of evaluation 508
overflow error 577
parms (see under name of parm)
propagating errors 647
RETAIN parm 234, 258, 512, 656

size of result field 511
speed-up tips 655, 656
syntax 507
use of built-in functions 48, 100, 475, 514
using to detect invalid data 257
when an operand is in error 647
where to put 46, 363, 445
which computed fields are totalled 150, 509
writing conditional expressions (WHEN parm) 459
(see also Computational expressions)
Concatenation
example 474, 514
how to perform 48
operator 473
removing excess blank spaces 631
Conditional COMPUTE statement 50, 508, 513
Conditional expressions 459
bit field conditions 465, 471
comparing character operands 40, 340, 449, 462, 470
comparing date operands 42, 341, 449, 462, 470
comparing hexadecimal values 449, 464, 472
comparing numeric operands 42, 337, 340, 449, 461, 470
comparing operands of different lengths 462
comparing operands of different types 340, 451, 463
comparing time operands 44, 273, 344, 450, 462, 470
comparing to multiple values 468, 471
how to simplify long expressions 468
how to write 40, 459
in COMPUTE statement 50, 513
in INCLUDEIF statement 40, 541
mixing relation and bit field conditions 466
order of evaluation 467
relation operators allowed 461
searching for a text in a character field 462, 471
selecting fields with invalid data 464
speed-up tips 652, 655, 657
use of ampersand (&) in 466
use of not sign (¬) in 470
use of quotation marks, apostrophes 42, 451, 470
use of the keyword "AND" 42, 465, 471
use of the keyword "NOT" 469, 471
use of the keyword "OR" 42, 466, 471
use of vertical bar (|) in 467
using both "AND" and "OR" in 467, 471
using parentheses in 44, 467, 469, 471
with multiple conditions 42, 50, 465, 471
(see also INCLUDEIF statement)
Conditions
assigning value to field based on 50, 508
which records to include in report 40, 540
(see also Conditional expressions)
Contains operator (:) 461, 462, 471
Continuing
control statements on multiple lines 444
literals across lines 444
Control breaks 65, 108, 177
S appears in lines printed at 182
blanking out repeating values at start of 505
break field must be a sort field 65, 177
breaks at wrong place 600
computing true ratios, percentages 202, 474
counting occurrences of a value 214
definition of 65
determining level of 181, 204
how to format dates, times and numbers 491
in PC files 108
in Web reports 303, 308
multiple 210
number of items in control group 181, 489
printing a certain number of detail lines per break 561
printing a line at control breaks in Web reports 303, 323
printing averages at 186, 191, 483, 491, 597
printing blank lines at 67, 69, 178, 184, 196, 486, 598
printing data from files at 189, 489
printing footing lines at 483
printing lines at beginning of control group 484, 486
printing multiple lines at 196
printing statistics at 186, 191, 484–485, 491, 597–598
printing the current date at 185
printing the number of items included in report so far 489
resetting page number 180
skipping to new page 598
spacing at 178, 486, 598
spacing at, for summary reports 576
statistical lines, customizing 186, 207, 483–485
statistical lines, order in which printed 188, 196
the Grand Total control break 483
total line (see also Totals and Total line)
total line split onto two lines 181
total line, customizing 196, 207, 487
total line, default 180
total line, multiple 184
total line, suppressing 487, 599

using to produce summary reports 73, 209, 216
where total line prints 184
(see also BREAK statement and SORT statement)
Control listing
DD used in JCL 414
logical unit written to (VSE) 427, 433
printing records copied from copy library 364, 519, 547, 584
skipping to new page 446
Control statements
how to write 443
introduction 84
keeping in a copy library 360
list of 86, 477
maximum number allowed 568
order 444
putting comments in 445
syntax (see under name of statement)
that apply to all reports in shop 424
that define files and fields 326
that require more than one line 444
what DD used to read 414, 424
what logical unit read from (VSE) 427
which columns to use 443
Convention
used in control statement syntax 478
Conversion
different types of date fields 269, 341
different types of numeric fields 337
different types of time fields 344
GMT time to local 575
of character to numeric data 340
of character to time data 274
of COBOL and Assembler layouts to FIELD statements 378
of numeric to character data 341
of numeric to time data 274
of one data type to another 463, 632, 636, 640–641
of time to numeric data 274
Copy library
accessed for READ and INPUT statements 363
assigning (VSE) 439, 576
copying records from non-PDS files 364, 519
DB2 file definitions 408
how to use 360, 421, 437, 516
making reports without using 360
preventing automatic copying 364, 545, 582
printing copied records in control listing 364, 519, 547, 584
saving COBOL and ASM record layouts in 377
saving shop-wide options 424
setup (OS/390) 420
setup (VSE) 437
used within ASM statement 383, 559
used within COBOL statement 383, 559
using an alias 363, 367, 648
which DD in JCL used for 414, 423
which member copied 367, 517
COPY parm
in INPUT statement 364, 545
in READ statement 364, 582
COPY statement 516
copying COBOL and Assembler record layouts 382
default copy library 559
how to use 364, 423
listing copied statements 446
parms (see under name of parm)
syntax 517
within scope of ASM and COBOL statements 385
Count
discrepancy in VSAM record count 424, 433
how many times a value occurs 214
Counter
line numbers in report 211
COUNTER built-in field (see #COUNTER built-in field) 626
CPU
utilization, speed-up tips 652
Creating your own fields 46, 98, 472, 506
(see also COMPUTE statement)
Crosstab reports 217
Cumulative
number of items printed in report 198, 489, 626
Currency
showing currency in PICTURE 140, 456
Current
date, built-in field 626
location, in COBOL and Assembler layouts 384
location, when defining fields 352, 353, 524
time, built-in field 625, 627
Cursor
in DB2 397
Customizing
the total line at control breaks 182
Cutoff
century cutoff year 269

D

DASD files

used as input (VSE) 533
Dash (-)
blanks required around 473
formatting negative numbers, where to put 452
in numeric literals 449
meaning in COLUMN or DISP parm 351, 524
name broken at, for column headings 130, 504
subtraction symbol 47, 473
use in field names 446, 473
Data
character versus numeric data 339, 450
how to format in report 58, 137, 503
including only certain values in report 40, 540
invalid, testing for 464, 472, 647
representation, date fields 341, 611
representation, numeric fields 335, 610
representation, time fields 613
specifying the input file 34, 542
the five types 333, 448
Data exit programs
DD used in JCL 415
decimal digits returned by 525
how to use 357
passing parms to 357, 360, 525, 534, 546, 582, 666
sample program 666
size of character result 526
Data set name (see DSNAME)
Data types
in FIELD statement 529
list of 609
listing of, for each input field 551
of COMPUTE fields 508
Databases 391
Date fields
creating your own 449, 506
default lengths 611
defining so that month name is always spelled out 343, 510, 526, 621
SMF dates 613, 622
stored in hexadecimal format 341, 464, 472
testing for valid data 464, 472, 643, 647
tips for working with 269
(see also Dates)
DATEDELIM parm
in OPTIONS statement 139, 140, 560
DATEEXIT
data type 613
Dates
adding to, subtracting from 221, 475, 640
adding/subtracting days, weeks, months or years 639
calculating first & last days of a week, month or year 638, 639
comm area date (VSE) 626
comparing 42, 341, 449, 462, 470, 513
converting numeric day, month and year into a date 640
converting to character value 632
converting to numeric value 475, 636
current date, built-in field 626
day of week for a given date 631
DD/MM/YY date literals 561
default display format 139, 618
default display format, changing 562
defining date fields 340, 521
delimiter used 139, 140, 270, 560, 611, 612, 620–622
did time interval cross into next day 639
extracting the day, month and year portions 221, 368, 635, 637–638
formatting in report 58, 137, 167, 192, 343, 503, 620
handling invalid dates 254, 257, 643, 644
how date fields stored in input file 341, 529, 611
how sorted 600
in COBOL and Assembler record layouts 375, 379
in PC files 271
including date in footnotes 175
including date in titles 163, 604–??
including in total line 185
including only certain dates in report 42, 470
Julian 271, 343
month name for a given date 633
non-standard 357
number of days between two dates 636
numeric day of week for a given date 635
printing blanks instead of zero dates 129, 167, 192, 249, 490, 503, 605, 632
range allowed in date literals 449
selecting the earliest/latest of several dates 637
spelling month name out 55, 58, 139, 167, 192, 343, 503, 621, 632
taking into account when computing time intervals 274
testing for leap year 642
which century for 2-digit years 559
writing date literals 449, 560–561, 576
writing julian date to output file 282
zero assigned for missing fields 229, 594
(see also Date fields)
Date-time pairs
adding to, subtracting from 639

DAY built-in function (see #DAY built-in function) 631
Day of week
built-in field 625
calculating the date corresponding to any day of a week 638
computing for a given date 631
including in footnotes 175
including in titles 163
number representing, for a given date 635
Daylight Savings Time 640, 641
DAYNAME built-in field (see #DAYNAME built-in field) 625
DAYNUM built-in function (see #DAYNUM built-in function) 635
Days
adding to a date field 475, 640
adding to or subtracting from a date field 639
converting numeric day, month and year into a date 640
day in century 640
day of month, for a given date 635
did time interval cross into next day 639
number of days between two dates 636
DB2 392
ambiguous field names 403, 405
ASC and DESC parms 399, 587
column headings 407
cursor 397
defining input table name 534, 545, 582
digits allowed in numeric fields 410
display formats 407
getting list of columns' data type 397
host variables 405, 406
how to create a PC file 395
how to create a report 393
JCL required 393, 415
list of DB2 columns in table 397, 550, 589
literals, format of 405, 407
missing rows 229, 230, 594
mixing DB2 and non-DB2 data 392
ORDERBY parm 399, 550, 587
plan name 409, 560
qualifiers 406
reading multiple rows 79, 401, 584
saving definitions in copy library 408
SELECT clause 590
setup 409
subsystem 393, 409, 561
table names 393
tables as auxiliary input file 400, 590
using multiple DB2 tables 400, 403
using record names with 403, 405
views 393
WHERE parm 397, 405, 552, 590
which rows to read 552, 590
DB2 tables
testing for missing row 642
DB2NAME parm
in FILE statement 534
in INPUT statement 395, 545
in READ statement 582
DB2PLAN parm
in OPTIONS statement 560
DB2SUBSYS parm
in OPTIONS statement 393, 561
DC and DS statements in Assembler 387
DCB parm in JCL
for output files 415
DD statement in JCL
for report output 414, 419, 571
sort work files 414, 574
which one used to read input files 329, 423, 534, 546, 582
which ones needed 414
writing FIELD statements to 379, 496
DDMMYY
date fields 611, 612, 620–621
DD/MM/YY date literals 141, 561
DDMMYYLIT parm
in OPTIONS statement 141, 561
DDMMYYYY
date fields 611, 612, 620–621
DDNAME parm
in COPY statement 519
in FILE statement 329, 415, 423, 534
in INPUT statement 331, 423, 546
in READ statement 331, 423, 582
Decimal digits
extracting integer value 635
how many in averages 151
how many stored in record 338, 525
how many to print 139, 452, 503
in Assembler layouts 388
in computed fields 512
in time fields 273, 346, 512, 613
returned by data exit programs 358, 525
rounding 512, 638
DECIMALS parm
in FIELD statement 338, 346, 525
Decrypting data 357
Default
alignment of titles 603
column headings for computed fields 511

display format, how to change 562
display formats 618
field location in record 351, 353, 524
justification of data 146
location, effect of defining bit fields 352
record name 589
sort order 559
spacing at control breaks 486
total line at control break 487
value assigned for missing records 229, 594
value assigned in COMPUTE statement 50, 508
Defining
bit fields 347
character fields 333, 339
date fields 340
fields created in exit programs 357, 666
fields in an earlier file 356
files and fields, how to 326, 420, 437, 521, 531
files automatically 363
files without using a copy library 360
how fields will be formatted in reports 335, 338, 343, 347, 349, 526
how to create your own fields 46, 98, 472, 506
numeric fields 335, 339
same part of record multiple times 327, 352, 368, 524
the column heading to use for a field 350, 511
time fields 344
where fields are located in record 350, 352, 353, 524
which fields should be totalled 148, 338, 339, 523
Definition statements 326
Delimited
files, how to create 276
Delimiters
in date fields 270, 611, 612, 620–622
in output files for PC programs 560
in time fields 613, 622–623
used in PC files 278
used to format dates 139, 140, 560
used to format times 139, 140, 576, 622–623
used to parse character strings 633
DEPENDING ON clause in COBOL 255, 266, 378
DESC parm
in DB2 ORDERBY parm 399, 550, 587
in SORT statement 62, 213, 597
DETAIL parm
in OPTIONS statement 211, 212, 561
Detail records
in batch type files 234
Detail report lines
suppressing 73, 209, 216
Different
assigning different values to created field 508
lengths, comparing operands of 462, 470, 472
Digits
calculating how many digits in packed fields 337
decimal, dropping 635
decimal, how many print in averages 151
decimal, how many stored in record 338, 525
decimal, rounding 512, 638
extracting certain digits from a number 340
how many stored in record 337
how many to print 139, 452, 503
in page number, how many 174, 626
maximum number allowed in literals 449
not enough room to display 645
number allowed in DB2 data 410
rounding to thousands, millions 638
(see also Decimal digits)
Dimension
arrays in records 327
Direct reads
auxiliary files read randomly 76, 591
DISP parm
in ASM & COBOL statements 494
in FIELD statement 524
Displacement
fields' starting displacement in records 350, 352, 353, 524
DISPLACEMENT (DISP) parm
in FIELD statement 263, 350, 353
DISPLAY
data type 610
display format 619
Display formats
changing the default 278, 562
default 139, 618
for PC files 503
formatting numbers with dots instead of commas 140, 619
how to write PICTUREs 451
how to write TPICTUREs 458
in COLUMNS statement 58, 129, 137, 503
in COMPUTE statement 510
in FIELD statement 326, 526
list of 617
of fields in the title 167, 605
of fields printed at control breaks 192, 491
removing excess blank spaces 631
specifying delimiter for dates 560
specifying delimiter for times 576
specifying for output files 278

used in total (and average) line 140, 151
using to create a character field 632
which quotation mark used 573
Division
division by zero indicator (***Z***) 645
division by zero, suppressing 577, 646
how to perform 47, 473, 506
performing division at control breaks 202
remainder (#MOD built-in function) 637
results in overflow (**V**) 645
DIVTOTS parm
in COMPUTE statement 202, 510, 515
DLBL statement in JCL
sort work files 434, 435, 574
used for writing output 427, 432, 435, 570
which one used for input files 331, 432, 533
writing FIELD statements to 379
DMY see #DMY built-in function) 640
Dollar sign ($)
how to print 139, 339, 452, 503, 619
meaning in PICTUREs 455
Dollars
DOLLAR display format 619
printing whole dollars 339, 512, 638
Dot (.)
using instead of commas in numbers 619
DOTSEP
display format 140, 619
DOUBLE parm
in OPTIONS statement 153, 249, 573
Downloading
files from POWER queue 429
from CICS (VSE) 432
only selected records 93
small summary files 113
Drawings, in Web reports 305
DSECT statement in Assembler 384, 387, 390
DSNAME
relation to file name 329, 331, 534, 546, 582
Dump
record dump for invalid data
Invalid data
dump of 566
record dump for normalization errors 566
Duplicate
records in file for a key 232
Duration
totalling 150
DXPARM parm
in FIELD statement 525, 666
DXPROG parm
in FIELD statement 358, 525, 666
DXRETDEC parm
in FIELD statement 358, 525, 667
DXRETLEN parm
in FIELD statement 358, 526, 667
Dynamic
HTML 297, 306, 315

E

EBCDIC
built-in function (see #EBCDIC built-in function) 631
converting to ASCII 143, 631–632
EBCDICTABLE parm
in OPTIONS statement 562
Either
of two conditions 42, 466, 471
ELSE parm
in COMPUTE statement 50, 511
Empty
report, completion code for 562
EMPTYCC parm
in OPTIONS statement 562
EMPTYMSG parm
in OPTIONS statement 562
Encrypted data
how to process 357
End of file
forcing EOF early 551, 576
End of report
printing lines after 309, 318, 572
printing lines at 207
Ending
of words (singular or plural) 198, 489, 625
EQU statement in Assembler 387, 390
EQUAL parm (see #EQUAL parm)
Equal to
comparing contents of fields 40, 461, 470
Error indicators
ambiguous reference (***A***) 644
divide by zero (***Z***) 645
error (***E***) 644
invalid data (***I***) 644
list of 160, 644
offset error (***F***) 355, 644
overflow (***V***) 645
propagation of 647
size (***S***) 135, 136, 182, 453, 645
suppressing 646
testing a field for errors 642

undefined field (***U***) 645
(see also under ***A***, ***I***, etc.)
Error messages
changing I/O error severity 569, 586
changing severity of 549, 569, 586
DD used in JCL 414
logical unit written to (VSE) 427, 433
normalization errors 248, 566
ERROR see #ERROR built-in function) 642
ESDS VSAM files
reading 329, 331
writing to 417, 432
writing to (VSE) 432
EXAMINE (see under #TRANSLATE)
Examples
files used in 648
Excel
example 89, 109, 111, 118, 119, 121
producing output file for 572
Exception reports
completion code 568
Excluding
fields from output 158, 159, 504, 505
Exit programs
(see Data exit programs and I/O Exit)
EXITPARM parm
in FILE statement 360, 534
in INPUT statement 546
in READ statement 582
Expressions
computational (see Computational expressions)
conditional (see Conditional expressions)
EXTENT statement in JCL 433, 434

F

False
bit value 642
Features, list of 25
FIELD statement 521
creating from Assembler record layout 378, 479
creating from Cobol record layouts 378, 493
data types, list of 609
defining a field's column (or displacement) 353, 524
defining a field’s column (or displacement) 350–355
defining arrays 243, 355
defining column headings 350
how many decimal digits 338, 525
how to define bit fields 347
how to define character fields 333, 339
how to define date fields 340
how to define fields created in exits 357, 666
how to define numeric fields 335, 339
how to define time fields 344
how to use 333
keeping in a copy library 363
location determined by another field 266, 353, 528
making starting column relocatable 524
multiple fields for same column in record 237
parms (see under name of parm)
purpose 326
redefining part of a record 327, 352, 368, 524
rules for field names 446
syntax 522
where to put 360, 421, 437
with DB2 data 407, 408
(see also Fields)
Fields
built-in (see Built-in fields)
comparing contents of 40, 460, 470
converting character to numeric, and v.v. 340
creating your own 46, 98, 472, 506
defining as character versus numeric 339
defining for an earlier file 356
defining one-time fields 368
defining, how to 326, 333
how date fields stored in files 341, 611
how many decimal digits in 338, 512, 525
how many digits in 337
how numeric fields stored in files 335, 610
how time fields stored in files 613
listing of fields in input file 37, 372, 550, 589
name cannot be split across lines 444
name used as column heading 130
qualifying field name with record name 77, 226, 228, 550, 589
resolving ambiguous field names 77, 226, 228, 447, 550, 589
specifying which to print 498
testing for errors in 642
testing for missing fields 230, 642
used in examples 648
where located in records 350, 352, 353
(see also FIELD statement)
File names
for naming Web reports 296
used as record names 228, 550, 589
FILE parm
in ASM & COBOL statements 372, 374, 495

in FIELD statement 356, 369, 526
FILE statement 531
how to use 328
keeping in a copy library 363
maximum record length 533, 535
overriding parms temporarily 331
parms (see under name of parm)
purpose 326
rules for file names 446
syntax 532
use with ASM and COBOL statements 370, 372, 379
variable length files 535
VSE file attributes 532
where to put 360, 421, 437
which DD used for file 423, 534
which DLBL/TLBL used for file 432
with DB2 data 407, 408
(see also Files)
Files
assigning file names 329, 532
auxiliary input files 76, 578, 587, 591
auxiliary input files are keyed 590
chaining (nesting) input files 226
copying statements from 363, 516, 545, 582
DDNAME and DSNAME used 329, 423, 517, 519
defining automatically 363
defining without using a copy library 360
how primary and auxiliary input files are processed 76, 552, 591
how to define (OS/390) 326, 328
how to define (VSE) 326, 331
I/O errors 230, 569, 586
input file attributes (VSE) 532, 544, 580
maximum record length 330, 331, 332, 547, 584
multiple input files 76, 224, 578
overriding file definition 330, 332, 542, 578
primary input file 76, 542, 552
reading a certain number of records 565
reading if key greater than or equal 230
reading multiple records for the same key 232
reading multiple records from same file 224, 589
reading with generic key 230, 547
resolving ambiguous file names 226
sample files used in examples 648
sorting mainframe files 283
specifying the input file 34, 542, 578
stop reading before EOF 551, 576, 660, 661
subsetting mainframe files 283
types of files supported 329, 331, 536
types of files supported (VSE) 533
using PDS files as input 329
using tape files as input 329, 331, 332
using VSAM files as input 329, 331, 551, 590
variable length 352
which DD statement used to read 329, 423, 534, 546, 582
which DLBL statement used to read 331, 432
(see also FILE statement)
FILLER 386
FILSZ parm, of system Sort program 574
First
day of a week, month or year, calculating 638, 639
line of report, putting lines before 309, 318, 572
FIXED BINARY (see BINARY)
FIXED DECIMAL (see PACKED)
Fixed format ASCII 279
Flags (see Bit fields)
Fonts
blinking 294, 322
bold font 294, 296, 298, 301, 303, 322
colored font 298, 315, 319
fontname 322, 323
italics 294, 303, 323
mainframe printers 418, 572
non-proportional 301
proportional 301
size 322–323
specifying 294, 301, 319, 322, 323
subscripts 324
superscripts 324
underlining 294, 303, 324
FOOTING parm
in BREAK statement 110, 184, 188, 207, 286, 483
in BREAK statement, how to use 188
in BREAK statement, multiple 196
in BREAK statement, printing blank lines 184
in BREAK statement, using instead of total line 196
where footing line prints 196
Footings
at bottom of each page (see FOOTNOTE statement)
at end of control breaks (see FOOTING parm)
lines printed at end of report 207
FOOTNOTE statement 538
alignment (left, center and right) 539
how to use 175
including date, time and page number 175
suppressing footnote lines 568
syntax 539
to force full page length 300

where to put 445
Forcing lower level control breaks 206
Format (see Display formats)
FORMAT built-in function (see #FORMAT built-in function) 632
FORMAT parm
in FIELD statement 326, 335, 338, 343, 347, 526
in OPTIONS statement 140, 271, 278, 562
Forms, how to print 153
Free format
control statements 443
fields, scanning for text in 462, 471
FULLWORD
data type 610
writing fullwords to output file 619
Functions (see Built-in functions)

G

GB (gigabytes)
formatting 454
GENERIC parm
in READ statement 230, 232, 583–588
GETVIS 433
GIF files 305
GMT times 575
GRAND parm (see #GRAND)
Grand totals
aligning in Web reports 303, 306, 316
customizing 207, 483
display format used in 140
how many decimals in 140
in Web reports 303, 309
printing averages at 207, 483
printing statistical lines at 207, 209, 483
prints by default 35
size error in (***S***) 135, 136, 454
spacing at 209, 279, 567
suppressing 209, 279, 567
totalling time fields 150
when put on new page 209
which columns receive 148, 339, 502, 509, 523
(see also Totals and Total line)
Graphics
aligning in Web reports 305, 323
aligning text and graphics 312
at control breaks 308
characters, in literals 448
in report titles 305, 312
in Web reports 294, 305, 306, 323
width of 323
Graphs
bar, how to print 154
Greater than
comparing contents of fields 42, 461, 470
largest of several fields 637
read if key greater than or equal 230
Gregorian dates 269, 272, 343
Grouping
computations 46, 474
report lines 65
rows in PC file 108

H

HALFWORD
data type 610
writing halfwords to output file 619
Harvard Graphics
column headings for 279
producing output file for 564
HDGSEP parm
in OPTIONS statement 132, 133, 350, 563
Header records
in batch type files 234, 289
normalizing batch files 247
HEADING parm
in BREAK statement 200, 207, 484
in FIELD statement 326, 350, 527
Headings
at beginning of a control group 200, 484, 486
column headings (see Column headings)
printing at top of each page 207
printing control break headings on each page 202, 486
printing once at beginning of report 207
row 153, 501
(see also Titles)
HEX display format 618
use in COLUMNS statement 139
use in FORMAT option 563
Hexadecimal representation
dates stored in 341, 612, 621
how to print 139, 335, 618
in computational expressions 449, 514
in conditional expressions 449, 464, 472, 664
times stored in 614, 615, 623
writing literals in 448, 472, 664
HGCOLHDG parm

in OPTIONS statement 133, 279, 564
HHMM
time fields 613, 614, 622–623
HHMMSS
built-in field (see #HHMMSS built-in field) 627
time fields 613, 614, 622–623
Higher
level of control break 204
High-values 472
in date field 271
Histograms (see Bar graphs)
Holes, leaving room to punch 154
Host variable
in DB2 expressions 405, 406
Hot links, in Web reports 294, 308, 322
Hours
12-hour format 347, 622
added to STCK fields 575
adding to or subtracting from a date-time 639
adding to or subtracting from a time 640, 641
displaying times as 622, 623
extracting from a given time 635
HOURS data type 615
since midnight 615
HTML
<BODY> tag 319
<HEAD> tag 319
<IMG> tag 323
<PRE> tag 319
adding hot links 308, 322
audio and video clips 308
centering text 298, 322
closing tags 318
creating HTML reports 294
dynamic HTML 297, 306, 315
file name extensions 296
for body of report 301
for column headings 301, 316, 564, 565
for control breaks 303, 308, 309
for total lines 301, 303, 309, 564
HTML comments 316, 324
HTML headers 323
HTML labels 309, 322
HTML tables 312, 324
in report titles 298, 305, 308, 312
list of HTML colors 322
list of HTML tags 321
opening tags 318
parm, in OPTIONS statement 294, 301, 316, 319, 320, 564
printing a horizontal line 303, 323
putting graphics in report 305, 306, 323
syntax-checking HTML 297
where to find online specifications 321
writing your own 296
HTMLAID parm
in OPTIONS statement 301, 316, 319, 320, 564
Hundreds
hundredths of seconds in time field 346
rounding to 638
Hypertext links (see Hot links)

I

I/O
errors, changing severity 569, 586
If logic 459
in COMPUTE statement 50, 513
selecting records to include in report 40, 540
(see also INCLUDEIF statement and WHEN parm)
IF statement
in COBOL, PL/I or BASIC 42
in COLUMNS statement 159, 501
INCDATE (see #INCDATE built-in function) 639
INCDATETIME (see #INCDATETIME built-in function) 639
INCDURATION (see #INCDURATION built-in function) 640
INCLUDEIF statement 540
comparing date fields 341
comparing time fields 344
equivalent to DB2 WHERE clause 397
examples 449–451, 470
how to use 40, 93
including a certain number of records in report 565
including only certain dates in report 42, 449, 470
including only certain times in report 44, 450
multiple statements 40
no records included 562
omitting 540
reading a certain number of records 565
selecting certain whole records to output 283
selecting records with invalid data 464, 472
specifying multiple conditions 40, 465, 471
speed-up tips 652
syntax 541
use of the keyword "AND" 42, 465, 471
use of the keyword "NOT" 471
use of the keyword "OR" 42, 466, 471
where to put 40
which fields allowed in 40

writing conditional expressions 40, 459
(see also Conditional expressions)
Including
a certain number of records in report 565
selected records in report 540
(see also INCLUDEIF statement)
INCTIME (see #INCTIME built-in function) 641
INDEX built-in function (see #INDEX built-in function) 635
Index variables 237
In-line
putting definition statements in-line 360
INNER parm
in COLUMNS statement 158, 505
INPUT statement 542
copies records from copy library 363, 545
how to use 34
I/O errors 569
listing records copied from copy library 364, 547
naming the record 228, 550
overriding file definition parms 330, 332, 542
parms (see under name of parm)
read partial file 551
reading a certain number of records 565
reading DB2 tables 393, 545
specifying more than one input file 76, 578
syntax 543
variable length files 547
where to put 445
which DD used for file 423, 546
which DLBL/TLBL used for file 432
INT built-in function (see #INT built-in function) 635
International formatting options 140
Internet Explorer 294
Internet protocol address (see IP address)
Interval
computing time interval 274
totalling 150
Invalid data
how sorted 601
identifying records that contain 464, 472, 647
indicator (***I***) 644
suppressing error 577, 646
suppressing from report 254, 257
testing for 642, 643
IOEXIT
file type 533, 551
parm, in FILE statement 534, 673
parm, in INPUT statement 546
parm, in READ statement 583
IP address, parsing 633
ISNUM see #ISNUM built-in function) 642
Italic
font, specifying 294, 303, 323
ITEM1-ITEM9 built-in fields (see #ITEM1-ITEM9 built-in fields) 626
ITEM-ENDING built-in field (see #ITEM-ENDING built-in field) 625
Items
number of, in control group 181, 198, 489, 626
number of, in report so far 198, 489, 626
number of, in whole report 35
ITEMS built-in field (see #ITEMS built-in field) 626

J

JCL 411
completion codes 230, 425, 439, 549, 569, 586
copy library used 423, 517
copying statements not in the copy library 519
DATE statement (VSE) 626
DD used for report 414, 417, 419, 571
DD used for sort work files 414, 574
default blocksize 567
DLBL used for report 435
DLBL used for sort work files 434, 435, 574
EXEC statement SIZE parm (VSE) 433, 575
for DB2 393, 415
for multiple reports (OS/390) 419
for multiple reports (VSE) 435
for OS/390 systems 412
for PC files (OS/390) 415
for PC files (VSE) 429
for reports (OS/390) 412, 415
for reports (VSE) 427
for VSE systems 425
list of DDs used 414
logical unit assignments 427
logical unit used for report (VSE) 431
LRECL of output file 571
no copy library used 360
sample PROC (OS/390) 417
specifying buffer size in JCL 423, 433
what DD statement used for input files 329, 423, 534, 546, 582
which DLBL/TLBL used for input files 331, 432, 533
writing FIELD statements to output file 379
Job
completion codes 230, 549, 569, 586
including jobname in report 625
JOBNAME built-in field (see #JOBNAME built-in

field) 625
JPG files 305
Julian dates 343, 612, 613
date literals 44
formatting output as 621–622
in Cobol records 375
in input file 341, 612, 613
with all zeros or all nines 271
working with 269, 271
writing to output file 282
Justification
doesn't look correct 168, 173, 606
maximum size allowed 146
of column headings 132
of data in report columns 129, 146, 505
of data printed at control breaks 192, 491
of data within titles 165, 168, 606
of data, default 146
of titles (left, center and right) 55, 153, 168, 603
(see also Alignment)

K

KB (kilobytes)
formatting 454
KEEPRDW parm
in FILE statement 263, 353, 535
in INPUT statement 353, 547
in OPTIONS statement 565
KEY parm (see READKEY parm)
KEYRANGE parm
in INPUT statement 547, 591, 660
Keys
auxiliary input files 590
auxiliary input files are keyed 76, 587, 591
building a packed or binary read key 589
creating read key with COMPUTE statement 79, 589
generic 230, 232, 547
generic (see also GENERIC)
greater than or equal to 230, 583
key to one file is contained in another file 226
reading a keyrange 547, 660
selecting key for READ statement 77, 226, 587
KGE parm
in READ statement 230, 583–588
KSDS VSAM files 329, 331, 547

L

Label
HTML labels 309, 322
tape labels (VSE) 533
Large (see Big)
Laser printers
fonts 572
setup string 572
skipping to new sheet 572
Last
line of report, putting lines after 309, 318, 572
sign in last digit of number 610
LCASE built-in function (see #LCASE built-in function) 632
Leading
blanks, removing 631
slash, in TITLE statement 169
zero, in time literals 450
zero, not required in date literals 449
zero, suppression 139, 455, 619
zeros, printing 455, 619
Leap year
testing for 642
LEAPYEAR see #LEAPYEAR built-in function) 642
Left alignment
of titles 55, 153, 168, 603
(see also Alignment and Left justification)
LEFT built-in function (see #LEFT built-in function) 632
Left justification
of column headings 132
of data in report columns 146, 505
of items in control break lines 192, 491
(see also Justification)
Left margin
aligning titles with 168, 603
how to specify 154, 565
moving first report column over 181, 210
(see also Margins)
LEFT parm
in #FORMAT built-in function 632
in BREAK statement 192
in COLUMNS statement 129, 146
in TITLE statement 168, 606
LEFTMARGIN parm
in OPTIONS statement 154, 565
Legend, for PC files 133
Length
allowed for file and field names 444, 446

comparing operands of different length 462, 470, 472
of date fields 611
of PC file records (OS/390) 415, 571
of PC file records (VSE) 431
of time fields 613
of variable length records 353
LENGTH (LEN) parm
in FIELD statement 335, 337, 341, 346, 528
Less than
#MIN built-in function 637
comparing contents of fields 44, 461, 470
Letters
ASCII versus EBCDIC 558, 562, 631, 632
lower case 632
upper case 634
Level, of control breaks 181, 204
Librarian (OS/390)
copying records from 383, 519
Librarian (VSE)
assembler copy library 559
COBOL copy library 559
copying records from 516
using as copy library 363, 437, 576
where loaded 434
which member type read 518, 566
Limiting
records read from input file 551, 576
Lines
how many per page 571
line number in report 211
printing a vertical line between columns 156
putting a line at control breaks in Web reports 303, 323
Links (see Hot links)
LIST parm
in COPY statement 446, 519
in INPUT statement 364, 547
in READ statement 364, 584
LIST/NOLIST parm
in COLUMNS statement 159, 505
Listing
of fields in a DB2 table 397
of fields in a file 37, 372, 374, 550, 589
records copied from copy library 364, 519, 547, 584
Literals 448
headings for columns of literal text 133, 501
how to write 42, 448
HTML tags 297
in body of report 126, 153, 501
in DB2 expressions 405, 407
in hexadecimal format 448, 464, 472, 514
in PC files 501
in titles 53, 161, 444, 448, 605
putting in PC file 95
that don't fit on single line 444
using DD/MM/YY date literals 561
when to use quotation marks 340, 450
writing time literals 450
Location
of field in record, after defining a bit field 352
of field in record, default 351, 524
Logical operations
#OFF built-in function 642
#ON built-in function 643
AND operation 630
OR operation 633
XOR operation 635
Logical units
assignments 427
for tapes (VSE) 332
Logo
in Web reports 294, 305, 312, 319, 323
LONG1-LONG3
date display format 621
Loop
looping through arrays 238
Lotus 1-2-3
example 101, 103
producing output file for 572
Lower
#MIN built-in function 637
level of control break 204
Lower case 632
Low-values 472
in date field 271
LRECL parm
in FILE statement 330, 535
in INPUT statement 547
in JCL, for output files 415
in READ statement 584
LRECL of output file 571

M

MAINFRAME parm
in OPTIONS statement 565
Mainframes
producing output files for 280, 565
sorting mainframe files 283
subsetting mainframe files 283

MAINSIZE parm
for system sort program 568, 575
MAKEDATE built-in function (see #MAKEDATE built-in function) 640
MAKENUM built-in function (see #MAKENUM built-in function) 636
MAKETIME built-in function (see #MAKETIME built-in function) 641
Margins
aligning titles with 168, 603
how to specify 154, 565, 571
moving first report column over 181, 210
Mathematical operations
between different types of numeric fields 337
how to perform 47, 472, 506, 513
(see also COMPUTE statement and Statistical lines)
MAX built-in function (see #MAX built-in function) 637
Maximum
#MAX built-in function 637
length of file and field names 444, 446
line (see also Statistical lines) 67, 186, 484, 597
line, printing at Grand Total time 207, 483
number of control statements 568
number of digits allowed in literals 449
number of lines to print 566
number of pages to print 566
number of records to include in report 565
number of records to read 565
selecting the largest of several values 637
size of character fields 511
value in control group, printing 67, 186, 192, 200, 491
value, which columns receive 148, 339, 502, 509, 523
year allowed in date literals 449
(see also MAXIMUM parm)
MAXIMUM (MAX) parm
in BREAK statement 67, 186, 484
in BREAK statement print expressions 192, 196, 200
in BREAK statement, two different uses 194
in SORT statement 186, 597
MAXINCLUDE parm
in OPTIONS statement 565, 664
MAXINPUT parm
in OPTIONS statement 565, 663
MAXINVSHOW parm
in OPTIONS statement 566
MAXNORMDUMP parm
in OPTIONS statement 248, 550, 566, 570, 587
MAXOCCURS parm
in ASM & COBOL statements 378, 495
MAXPAGES parm
in OPTIONS statement 566, 567, 664
suppressing message 279, 567
MAXPRINT parm
in OPTIONS statement 566, 567, 664
suppressing message 279, 567
MB (megabytes)
formatting 454
MDY see #MDY built-in function) 640
Member
of copy library, which one copied 367, 517
type, of VSE library 566
Memory (see Storage)
MEMTYPE parm
in OPTIONS statement 566
Messages
changing severity of 230, 549, 569, 586
suppressing maximum printed message 279, 567
Microsoft Access
example 94, 99
suppressing columns headings
Access, Microsoft
suppressing columns headings 93
Microsoft Excel
example 89, 109, 111, 118, 119, 121
Microsoft Works
example 106
MID files 308
Midnight
did time interval cross into next day 639
Millions
rounding to 453, 638
MIN built-in function (see #MIN built-in function) 637
Minimum
#MIN built-in function 637
excluding zero values 186, 485, 491, 598
line (see also Statistical lines) 67, 186, 484, 598
line, printing at Grand Total time 207, 483
value in control group, printing 67, 186, 192, 200, 491
value, which columns receive 148, 339, 502, 509, 523
(see also MINIMUM parm and NZMINIMUM parm)
MINIMUM (MIN) parm
in BREAK statement 67, 186, 484
in BREAK statement print expressions 192, 200
in BREAK statement, two different uses 194
in SORT statement 186, 598

Minus sign (-)
blanks required around 473
formatting negative numbers, where to put 452
in numeric literals 449
meaning in COLUMN or DISP parm 351, 524
name broken at, for column headings 130, 504
subtraction symbol 47, 473
use in field names 446, 473
MINUTENUM built-in function (see #MINUTENUM built-in function) 637
Minutes
adding to or subtracting from a date-time 639
adding to or subtracting from a time 640, 641
displaying times as 622, 623
extracting from a given time 637
MINS data type 614, 615
rounding to minutes 58, 347
since midnight 614, 615
Missing
records 231, 424, 433
records, default value used 229, 594
records, how to detect 230, 642
MISSING see #MISSING built-in function) 642
MISSOFFSET parm
in OPTIONS statement 566, 646
MMDDYY
date fields 611, 612, 620–621
MMDDYYYY
date fields 611, 612, 620–621
MOD built-in function (see #MOD built-in function) 637
Month
adding/subtracting months to/from a date 639
calculating first & last days of a month 638
calculating previous 221
converting numeric day, month and year into a date 640
extracting for a given date 221, 368, 637
name, for a given date 633
spelling out 139, 192, 343, 503, 621, 632
spelling out name, in column headings 224
spelling out name, in titles 55, 58, 167
MONTH built-in function (see #MONTH built-in function) 633
MONTHNUM built-in function (see #MONTHNUM built-in function) 637
MULTI parm
in READ statement 79, 232, 401, 584–588, 590, 658
with DB2 tables 590
MULTICOLHDG parm
in OPTIONS statement 133, 153, 254, 566
Multi-line reports 151, 249, 498, 566
Multiple
BREAK statements 69
COLUMNS statements 151, 254, 498, 566
conditions 40, 50, 465, 471
control breaks 69, 204, 210
DB2 tables 400, 403
fields defined at same location in record 158, 327, 352, 368, 505
footing lines at control breaks 196
INCLUDEIF statements 40, 540
input files 76, 224, 578
levels of totals 71, 204, 210
lines required for control statement 444
lines, splitting column headings 130, 350, 504, 527
multiple step jobs 284
NEWOUT statements 554
PC file records per input record 498
READ statements 79, 224
records for a READ statement 584
records from same input file 224
records in file for a key 232
report lines per record 151, 249, 498
reports in one run 289
reports, JCL changes (OS/390) 419
reports, JCL changes (VSE) 435
reports, sort work files for 414, 420, 435, 574
reports, storage for sort 575
sort fields 595
TITLE statements 53
total lines at control breaks 184
values in a relation condition 468, 471
Multiplication
how to perform 47, 473, 506
results in overflow (**V**) 645

N

NAME parm (see RECNAME parm)
Names
assigning field names 335
assigning file names 329, 367
field names from COBOL and Assembler record layouts 377, 386, 390
getting list of DB2 column names 397
getting list of field names 37, 372, 378, 550, 589
month, spelling out 55, 58, 139, 167, 192, 343, 503, 621, 633
of day for a given date 631

of day for current day 625
removing blanks between last and first name 631
rules for file, field, and record names 444, 446
sorting mixed case names 634
spelling out state name 228, 513
(see also File names and Record names)
Narrower
making a column narrower 60, 135, 505
(see also Width)
Negate
how to negate a condition 469, 471
Negative
numbers, scaled down to far or to zero 455
sign (see Minus sign)
Nesting
control breaks 204, 210
input files 226
nested arrays 244
parentheses 467, 474
New fields
how to create 46, 98, 472, 506
New page
skipping to in control listing 446
skipping to, in report 67, 178, 253, 486, 598
NEWOUT statement 289, 554
NEWSHEET (NEWSHEET1) parm
in BREAK statement 180, 209, 487, 572
in SORT statement 180, 209, 572, 598
Next page
skipping to new page 67
Nibble
C versus F for packed data 630, 633
NOACCUM parm
how to use 150
in COLUMNS statement 128, 150, 502
in COMPUTE statement 150, 509
in FIELD statement 150, 326, 338, 346, 451, 523
NOBLOCKSIZE parm
in OPTIONS statement 567
NOCC parm
in OPTIONS statement 279, 320, 567
NOCHECK parm
in OPTIONS statement 567
NOCLEARIO parm
in OPTIONS statement 567
NOCOLHDGS parm
in OPTIONS statement 93, 133, 175, 279, 320, 567
NOCOMMAS display format 619
NOGRANDSPACES parm
in OPTIONS statement 279, 283, 567
NOGRANDTOTAL parm
in OPTIONS statement 209, 279, 567
NOLABEL 533
NOMAXMSG parm
in OPTIONS statement 567
NONORMALIZE parm
in INPUT statement 548, 549
in READ statement 585
Non-proportional font 301
Non-zero average
line (see also Statistical lines) 67, 186, 485, 598
value in control group, printing 67, 186, 193, 200, 491
which columns receive 148, 339, 502, 509, 523
Non-zero minimum
line (see also Statistical lines) 67, 186, 485, 598
value in control group, printing 67, 186, 193, 200, 491
value, which columns receive 148, 339, 502, 509, 523
NOOVERPRINT parm
in OPTIONS statement 133, 568
NOREPEAT/NOREPEATPAGE parm
in COLUMNS statement 129, 144, 505
Normalization
errors 248, 549, 566, 569, 586
FIELD statements needed 243, 355
nested arrays 244
of arrays 237–248, 548, 585
of batched files 247
parallel arrays 245
preventing 548, 585
selected records only 549, 586
(see also NORMALIZE parm and NORMWHEN parm)
NORMALIZE parm
in FILE statement 242, 536
in INPUT statement 240, 548
in READ statement 585
NORMWHEN parm
in FILE statement 536
in INPUT statement 247, 549
in READ statement 586
NOSEQ parm
in ASM & COBOL statements 372, 495
NOSORTSIZE parm
in OPTIONS statement 568
NOSTOPWHEN parm
in INPUT statement 536, 549, 551
NOSYSINLIMIT parm
in OPTIONS statement 568
Not character (¬)

use in conditional expressions 470
NOT keyword
use in conditional expressions 469, 471
NOTALIAS parm
in COPY statement 519
NOTEMPTYCC parm
in OPTIONS statement 568
NOTITLES parm
in OPTIONS statement 134, 175, 279, 313, 568
NOTOTAL parm
in BREAK statement 185, 196, 209, 212, 487
in SORT statement 185, 599
NOUNDERSCORES parm
in OPTIONS statement 133, 134, 300, 320, 568
Number
in COLUMNS statement, meaning of 502, 505
including a certain number of records in report 565
of characters in report line 160, 417
of lines to print 566
of occurrences, counting 214
of pages to print 566
reading a certain number of records 565
Number of items
as column in report 211
in control group 181, 198, 489, 626
included in report 35
printed in report so far 198, 489, 626
Number sign (#), meaning of 53
NUMERIC
data type 610
display format 619
Numeric fields
comparing 42, 337, 449, 461, 470, 514
comparing to character fields 463
confusing with character fields containing numbers 339, 450
converting to a time value 274, 637, 641
converting to character 340, 632
converting to date value 475, 640
creating your own 46, 449, 472, 506
default display format 562, 618
formatting in report 58, 137, 167, 192, 338, 451, 503, 619, 632
formatting with dots instead of commas 140, 619
how sorted 600
how stored in input file 335, 529, 610
how to define 335, 521
integer portion 635
performing calculations 47, 449, 472, 506, 513
printing as a bar graph 154
printing blanks instead of zero 129, 167, 192, 490, 503, 605, 632
sign in last digit 610
specifying where to put plus, minus sign 452
testing for valid number 464, 472, 642, 647
writing numeric literals 42, 449, 470
zero assigned if record missing 229, 594
NUMERIC-SLD data type 610
NUMWORDS built-in function (see #NUMWORDS built-in function) 638
NZAVERAGE (NZAVG) parm
in BREAK statement 67, 186, 485
in BREAK statement print expressions 193, 200
in BREAK statement, two different uses 194
in SORT statement 186, 598
NZMINIMUM (NZMIN) parm
in BREAK statement 67, 186, 485
in BREAK statement print expressions 193, 200
in BREAK statement, two different uses 194
in SORT statement 186, 598

O

Occurrences
counting in a file 214
OCCURS clause in COBOL 249, 255, 266, 327, 355, 377, 495
how to process arrays 237
Odd page numbers
skipping to 180, 487
ODDPAGE (ODDPAGE1) parm
in BREAK statement 180, 209, 487
in SORT statement 180, 209, 598
OFF built-in function (see #OFF built-in function) 642
OFFSET parm
error in calculating 644
in FIELD statement 266, 353, 528
suppressing errors in 566, 646
OFFTEXT parm
in FIELD statement 349, 529, 601
ON built-in function (see #ON built-in function) 643
One-time
lines at beginning or end of report 309, 318, 572
One-to-many I/O 232, 584
ONIOERROR parm
in OPTIONS statement 569
in READ statement 586
ONNORMERROR parm
in INPUT statement 549
in OPTIONS statement 569
in READ statement 586
ONTEXT parm

in FIELD statement 349, 529, 601
Operating systems 411
Operations
character, how to perform 48, 473, 506
mathematical, how to perform 47, 473, 506
(see also COMPUTE statement and Computational expressions)
Operators
allowed in relation conditions 461
OPTIONS statement 555
column heading options 133, 175
parms (see under name of parm)
parms for custom PC files 279
sort-related options 601
specifying shop-wide options 414, 424
syntax 556–558
title options 133, 175
Web report options 320
where to put 445, 555
OR built-in function (see #OR built-in function) 633
OR keyword
use in conditional expressions 42, 466, 471
Order
in which BREAK statements appear 204
in which columns appear 159, 503
in which conditions are evaluated 467
of control statements 444
of input file processing 552
of operations in computational expressions 474
of report, how to specify 62, 595
of rows in PC file, how to specify 105
(see also SORT statement)
ORDERBY parm
in INPUT statement 399, 550
in READ statement 401, 587
similar to DB2 ORDER BY clause 399
ORG statement in Assembler 387
OS/390 operating system 412
OUTATTR parm
in ASM and COBOL statements 379, 496
in OPTIONS statement 429, 431, 435, 570
OUTDDN parm
in ASM and COBOL statements 379, 496
in OPTIONS statement 420, 571
OUTER parm
in COLUMNS statement 158, 505
OUTLRECL parm
in OPTIONS statement 418, 420, 571
Output files
access method used 417, 431
attributes (VSE) 570
blocksize 567
DD used for 414, 419, 571
DLBL used for 427, 435
empty 562
for mainframe programs 280, 565
for non-standard PC programs 275
HTML 294
JCL (OS/390) 415
JCL (VSE) 429
logical unit written to (VSE) 427, 435
making binary data fields 282
making packed data fields 282
not empty 568
printing one-line column headings 279
producing delimited ASCII file 276
producing fixed format ASCII file 279
record size 571
spacing between fields 282, 560
specifying display format 278, 282, 562
specifying field width 282
specifying record length (OS/390) 415
specifying record length (VSE) 432
specifying the delimiter 278, 560
suppressing blank lines 279, 283, 567
suppressing column headings 279, 283, 567
suppressing report titles 134, 175, 279, 283, 568
suppressing the carriage control character 279, 283, 567
suppressing the Grand Totals 279, 283, 567
suppressing underscores 568
using as an input file 234, 258
writing entire records from input file 283
writing FIELD statements to (OS/390) 496
writing FIELD statements to (VSE) 496
writing julian dates 282
writing selected records to output file 283
writing to VSAM (OS/390) 417, 571
writing to VSAM (VSE) 432
(see also PC files)
OUTPUT parm
in OPTIONS statement 279
OUTTYPE parm
in OPTIONS statement 417, 420, 571
Overflow
error indicator (***V***) 645
suppressing error 577, 646
testing for 642
Overlap
excluding overlapping fields from output 158, 505
title parts 173
total line text overlapping a column total 181
Overprinting

suppressing 133, 568
Overriding
column headings 60, 129, 130, 504
file definition parms 330, 332, 542, 578

P

Packed data
C or F in zone nibble 630, 633
calculating the number of digits 337
comparing with binary data 337
dates stored as 621–622
invalid 257, 577, 644
testing for invalid data 642
testing for valid data 464, 472, 647
unsigned, writing to output file 620
writing to output file 282, 620, 622, 623
PACKED data type
dates stored as 341, 612, 613
needed in read key 589
numeric field 333, 335, 610
times stored as 614, 615
PACKUN
data type, for time fields 614, 615
numeric data type 610
writing packed unsigned data to output file 620
Padding
blank, computed fields 511
blank, operands of different lengths 462, 470
hex literals 449
zero, operands of different lengths 472
Page
control group headings at top of 202, 207, 486
fixing page length 300
how many lines per page 571
layout, crosstab reports 219
maximum number to print 566
maximum width of 160, 417
printing footnotes at bottom 175
skipping to new page 67, 178, 253, 486
skipping to new sheet of paper 180, 487, 572, 598
skipping to new, in control listing 446
skipping to odd page 180, 487, 598
splitting related report lines across pages 253, 575
when Grand Totals put on new page 209
PAGE (PAGE1) parm
in BREAK statement 67, 178, 180, 209, 486
in SORT statement 178, 180, 185, 209, 598
Page breaks
suppressing 134, 175, 568, 577
Web reports 300
(see also Control breaks and BREAK statement)
Page number
built-in field 626
changing number of digits in 174
including in footnotes 175
including in titles 163, 604
resetting to page one 180, 486, 598
skipping to odd page 180, 487, 598
PAGELEN parm
in OPTIONS statement 154, 300, 424, 571
PAGENUM built-in field (see #PAGENUM built-in field) 626
Panvalet
copying records from 383, 519
Paper
skipping to new sheet 180, 487, 572
Paradox
example 114
Parentheses
nesting 467, 474
use in COLUMNS statement 128
use in computational expressions 46, 474
use in conditional expressions 44, 467, 469, 471
use in SORT statement 178
use in TITLE statement 167
PARM parm (see EXITPARM parm)
Parms
passed to system sort program 574
passing parms to exit programs 357, 360, 525, 534, 546, 582, 666
PARSE built-in function (see #PARSE built-in function) 633
Parsing
IP addresses 633
Partial
input file, reading 551, 576
keys (see GENERIC parm)
Partitioned data set (see PDS files)
Path to VSAM file 232
Payback chart 25
PC file
converting entire mainframe file 88
PC files
ASCII files 143, 219, 276, 279
blank lines in 110
control breaks in 108
creating from an existing report 258
dates 271
default total line format 108
delimiter used 278

display formats 278, 503
enclosing data in quotes 278
how to request 84, 572
inserting blank columns 110
JCL (OS/390) 415
JCL (VSE) 429
logical unit written to (VSE) 427, 435
printing one-line column headings 279, 564
producing non-standard PC files 275
related parms in OPTIONS statement 279
sorting 105
specifying record length (OS/390) 415
specifying record length (VSE) 432
summary files 110, 113
suppressing column headings 279
suppressing the carriage control character 279
suppressing the Grand Totals 279
using DB2 data 395
writing multiple records per input record 498
writing to VSAM (OS/390) 417
writing to VSAM (VSE) 432
YYYY dates 278
PDS files
copying statements from non-PDS files 519
rules for naming members 367
used as input to report 329, 536
using as copy library 363, 421, 423, 516, 517
PDSDDN parm
in COPY statement 517, 520
Percent
computing for control group 202, 510, 515
of totals 284
percentage change, how to compute 474
showing percent sign in PICTUREs 456
Period (.)
using instead of commas in numbers 140, 619
Photographs
in Web reports 294, 305, 306
PICTURE format
can prevent totalling 148, 502, 509, 523
compared to COBOL 327
currency indicator 140, 456
display format 619
eliminating unwanted leading digits 455
for international users 140
for time fields 622
how to write 451
in BREAK statement, examples 194
in COLUMNS statement, examples 138
meaning of @ and ? symbols 453, 454, 456, 458, 573
number scaled down too far or to zero 455
scaling to thousands, millions 453, 573
using to change column width 137
using to round out decimal digits 339
when allowed 340, 453
(see also TPICTURE format)
PL/1
"IF" statement 42, 540
INCLUDE library 363, 516
INDEX built-in function 462, 635
Plan
DB2 plan name 409, 560
Plural
ending of word 198, 489, 625
Plus sign (+)
addition symbol 47, 473
concatenation symbol 48, 473
formatting positive numbers, where to put 452
meaning in COLUMN or DISP parm 351, 524
meaning in PICTUREs 455
PM
showing AM and PM 347, 622
Pointers
to field within a record 266, 353, 528
POSTSCRIPT parm 318
in OPTIONS statement 297, 309, 315, 321, 572
Pound sign (#)
meaning of 53, 447
use in field names 446
POWER
downloading files from 429
writing output to (VSE) 431
Prefix
in variable length records 353, 524
using to resolve ambiguous field names 77, 226, 228, 550, 589
PRESCRIPT parm 318
in OPTIONS statement 297, 309, 313, 315, 572
in options statement 321
Previous
month, calculating 221
record, saving data from 234, 258
year, calculating 222
Primary input file
(see Files and INPUT statement)
Print expressions
in BREAK statement 189, 488
in COLUMNS statement 126, 499
in FOOTNOTE statement 175, 539
in TITLE statement 161, 169, 603
PRINT USING in BASIC 451
Printing
DD used for report 414, 417, 419

logical unit used for report (VSE) 427, 431, 435
on laser printer 572
printer can't overprint 568
records copied from copy library 364, 519, 547, 584
Priority
in evaluating conditions 467
of operations in computational expressions 46, 474
PROC
sample PROC (OS/390) 417
Propagation
of error conditions (e.g. ***I***) 647
Proportional font 301
PRTSETUP parm
in OPTIONS statement 418, 572
PRTSHEET parm
in OPTIONS statement 424, 572
Punching holes
leaving room for 154

Q

QCHAR parm
display format 618
in OPTIONS statement 573, 618
QSAM 417
Qualified field names 77, 226, 228, 447, 550, 589
Quattro Pro
spreadsheet, example 91
Question mark (?)
meaning in PICTURES 454, 456, 458, 573
Quit
reading input file, when to 551, 576
Quotation marks (" and ')
enclosing data in, for PC files 278, 618, 621, 623
imbedded within a literal 448
needed with character literals 42, 448, 470
use in BREAK statement 189
use in COLUMNS statement 126, 501
use in INCLUDEIF statement 42, 470
use in TITLE statement 53, 163, 444, 448, 605
when needed around numbers 340, 450
which character to use 42, 95, 448
which used for QCHAR display format 573

R

Random reads
auxiliary files read randomly 76, 591
Rank, printing in a report 211
Ratios, computing for control group 510
RDW 263, 353, 524, 535, 547, 565
READ statement 578
building a packed or binary read key 589
chaining 226
choosing the read key for 77, 226, 587
copies records from copy library 363, 582
default record name 228
generic keys 230, 232, 583
how to use 76, 116, 224
I/O errors 569, 586
key greater than or equal to 230, 583
listing records copied from copy library 364, 584
multiple READ statements for same file 224
multiple statements 79, 224, 578
naming the record 228, 589
overriding file definition parms 330, 332, 578
parms (see under name of parm)
reading DB2 tables 400, 582
reading multiple DB2 rows 79, 401
reading multiple records 584, 658
reading multiple records with the same key value 232
record not found 229, 594, 642
sorting on field from auxiliary input file 79
syntax 579
using COMPUTE field as read key 79, 589
VSAM versus sequential files 76, 590
where to put 445
which DD used for file 423, 582
which DLBL/TLBL used for file 432
Reading
a certain number of records 565
a range of records 660, 661
I/O errors 569, 586
not all VSAM records read 424
when to stop reading input file 551, 576
READKEY parm
building a packed or binary key 589
equivalent for DB2 tables 400, 590
in READ statement 230, 232, 587
in READ statement, key to file contained in another file 226
in READ statement, using a COMPUTE field 79
key greater than or equal to 230, 583
reading multiple records 584
reading multiple records with the same key value 232
using generic keys 230, 583
REALDATE see #REALDATE built-in function) 643

RECNAME parm
in INPUT statement 550
in READ statement 226, 228, 589
rules for record names 446
Record descriptor word (see RDW)
Record layouts
Assembler 369, 421, 438, 479
COBOL 369, 421, 438, 493
Record names
default value 228, 550, 589
how to assign 228, 550, 589
resolving ambiguous field names 77, 226, 228, 550, 589
rules for record names 446
use in COLUMNS statement 77, 88, 158, 226, 228, 500, 501, 550, 589
with DB2 data 403, 405
Record types
processing differently 234, 247
when to quit reading input file 551, 576
Records
defining the fields within 326, 521
discrepancy in record count 424, 433
length of variable length records 353
maximum size 330, 331, 332, 533, 535, 547, 584
not found for READ statement 229, 594
number of, included in report 35
reading more than one for the same key value 232
reading more than one from the same file 224
size of output records (OS/390) 571
size of output records (VSE) 570
specifying which to include in report 40, 449, 540
testing for missing records 230, 642
writing selected records to output file 283
RECSIZE 331, 332, 533, 570
REDEFINES clause in COBOL 327, 385
Redefining
part of a record 158, 352, 368, 505, 524
Relation
conditions, how to write 460
operators, list of 461
RELOC parm
in ASM and COBOL statements 381, 497
Relocatable
COLUMNS and DISP parms 524
Remainder, after a division 637
REPEAT parm
in BREAK statement 200, 202, 207, 486
Repeating
control group headings 202, 207, 486
values, blanking out 129, 144, 505
Reports
crosstab 219
for the Web 294
logical unit written to (VSE) 427, 431, 435
size of report line 571
using report output as input 234, 258
which DD used for 414, 417, 419, 571
wider than 132 characters 432
RETAIN parm
in COMPUTE statement 234, 258, 289, 512, 656
Retaining
data from previous records (see RETAIN parm)
Reverse
logic, in conditional expressions 469, 471
Right alignment
of titles 55, 168, 603
of titles, looks wrong 173, 606
RIGHT built-in function (see #RIGHT built-in function) 634
Right justification
needed in right aligned titles 173, 606
of column headings 132
of data in report columns 146, 505
of items in control break lines 192, 491
(see also Justification)
Right margin
aligning titles with 168
(see also Margins)
RIGHT parm
in #FORMAT built-in function 632
in BREAK statement 192
in COLUMNS statement 129, 146
in TITLE statement 168, 173, 606
ROUND built-in function (see #ROUND built-in function) 638
Rounding
decimal digits in computed fields 512
numbers to different scales 453
times 58, 347
to thousands, millions 638
using PICTURE to round 339
Row headings 153, 501
RRDS VSAM files 329, 331
Running count
of number of items printed in report 198, 489, 626

S

SAM files
  used as input (VSE) 533
Saving
  data from previous record 234, 258
SCALEPIC parm
  in OPTIONS statement 454, 573
Scaling
  eliminating unwanted leading digits 455
  number scaled down too far or to zero 455
  numbers automatically, how to 453
Scanning
  a field for a given text 462, 471, 635
Scope
  of ASM and COBOL statements 384
Searching
  a character field for a text 462, 471, 635
SECONDNUM (see #SECONDNUM built-in function) 638
Seconds
  adding to or subtracting from a date-time 639
  adding to or subtracting from a time 640, 641
  converting hours, minutes and seconds into 636
  decimal digits 346
  displaying times as 622, 623
  extracting from a given time 638
  how many seconds between two times 636
  hundredths of seconds 346
  omitting from time literals 44
  rounding to minutes 58, 347
  rounding to whole seconds 512
  SECS data type 614
  since midnight 614
Security, DB2 409
SELECT clause (DB2) 590
Selecting
  a certain number of records 565
  which records to include in report 40, 449, 540
Sequence numbers
  in COBOL record layouts 372, 495
Sequential
  files, defining 329, 331, 536
  files, not allowed in READ statement 76, 590
  files, used as input (OS/390) 536, 551
  files, used as input (VSE) 533
  primary input file read sequentially 76, 552
Setup
  copy library (OS/390) 420
  copy library (VSE) 437
  DB2 409
  defining files and fields 326, 521, 531
  JCL (OS/390) 412, 415
  JCL (VSE) 427, 429
  printer setup text 418, 572
  PROC (OS/390) 417
Sheet
  skipping to new sheet of paper 180, 487, 572, 598
Shifting report 154, 181
SHORT1-SHORT3
  date display format 621
Shorten (see Width)
SHOWFLDS parm
  in ASM and COBOL statements 372, 378, 497
  in INPUT statement 37, 397, 550
  in READ statement 397, 589
Sign
  nibble in packed data, changing 630, 633
  plus or minus, computing absolute value 635
  plus or minus, in numeric literals 449
  plus or minus, where to print 452
  sign in last digit of number 610
  unsigned numeric data 610
SIGN clause in COBOL 387
SINGLESPACE (SINGLE) parm
  in OPTIONS statement 573
Singular
  ending of word 198, 489, 625
Size
  block size of input file (VSE) 533
  block size of output (VSE) 570
  error indicator (**S**) 135, 453, 645
  error indicator (**S**), in total line 182, 454
  error indicator (**S**), using automatic scaling to suppress 454
  of column, changing 60, 135, 505
  of computed fields 511
  of fields in input records 528
  of fields in output files 282
  of items in control break lines 193, 491
  of items in the title 165, 168, 606
  of output records (OS/390) 571
  of output records (VSE) 570
  of report, maximum 160, 432
  record size (VSE) 533
  (see also Width)
SIZE parm in JCL 433, 575
Skewed
  report columns 151
  titles 168, 173, 606
  (see also Alignment)
SKIPBLANKDET parm

in OPTIONS statement 253, 257, 573
Skipping
to new page in control listing 446
to new page in report 67, 178, 486, 598
SKIPZERODET parm
in OPTIONS statement 249, 254, 257, 573
Slack bytes, in Cobol layouts 243
Slash (/)
changing delimiter for formatting dates 560
division symbol 46, 473
leading, in TITLE statement 169
meaning of /* and */ 445
trailing, in TITLE statement 169
used to align titles 55, 168
Smaller
making a column smaller 60, 135, 505
smallest of several values 637
(see also Width)
SMF files
date format 613, 622
tips for using 263
Social security numbers, how to format 340
Sort order
bit fields 601
character fields 600
dates 600
how to specify 62
mixed case fields 634
numeric fields 600
tie-breakers 559
times 600
(see also SORT statement)
SORT statement 595
ascending/descending order 62, 597
automatic sorting 62, 559
collating sequence used 600
computed field as sort field 506
control break spacing 67, 178, 598
control break spacing, summary reports 576
how invalid data is sorted 601
how to use 62, 105
JCL required 414, 434
multiple sort fields 62, 595
multiple sort tasks 414, 420, 435, 574, 575
multiple statements 595
name of sort program to use 574
options affecting 601
parms 62
preserving input file order 597
pre-sorting the input file 286, 659
printing averages at control breaks 186, 597
printing statistical lines at control breaks 186, 597–598
quitting the sort early 567
requesting control breaks 177, 596
requesting multiple control breaks 204, 210
requesting subtotals 177, 599
size parm passed to sort program 420, 436, 568, 575
skipping to new page 178, 598
sort field from auxiliary input file 79
sort program parms 574, 662
sort work files 414, 420, 434, 435, 574, 575
sorting mainframe files 283
speed-up tips 662
syntax 596
tie-breakers 597
using ORDERBY to sort DB2 data 399
where sort program loaded (VSE) 434
where to put 62, 445
which fields allowed 596
SORTDD parm
in OPTIONS statement 420, 436, 574
SORTNAME parm
in OPTIONS statement 574
SORTOPT parm
in OPTIONS statement 574, 663
SORTSIZE parm
in OPTIONS statement 420, 436, 575
SORTWK01 DD 414, 574
SORTWK1 DLBL 434, 574
SORTWORKNUM parm
in OPTIONS statement 434, 575
SPACE parm
for PC files 110
in BREAK statement 178, 209, 486
Spaces
ASCII spaces between fields 490
leading and trailing, removing 631
where allowed in control statement 443
Spacing
at control breaks 67, 69, 178, 486, 598
between report columns 128, 151, 502
between report columns in Web reports 301
between report lines 153, 573
of Grand Totals 209
report margins 154
(see also Spacing factor)
Spacing factor
default 128, 560
in COLUMNS statement 128, 151, 502
in lines printed at control breaks 191, 490
in titles 165, 167, 605

of zero 128, 198
used to shift report columns over 181, 210
Special characters
in literals 448, 472
Special forms, how to print 153
Speed-up tips 652
SPLITDETAIL parm
in OPTIONS statement 253, 575
Splitting
column headings into multiple lines 130, 350, 504, 527
control statement into multiple lines 444
report into multiple lines 151
titles into parts 55, 168, 603
why total line split into two lines 181
SRT2WK01, SRT3WK01, etc. DD 414, 420
SRT2WK1, SRT3WK1, etc. DLBL statements 436
Stacking
column headings 132, 350, 504, 527
Standards
shop standards 424
Stars (*) (see Asterisks)
STARTCOL parm
in ASM & COBOL statements 384, 497
in ASM and COBOL statements 384
STARTDISP parm
in ASM & COBOL statements 384, 497
State
spelling out name 228, 513
Statistical lines
at control breaks, customizing 186, 483–485
how to print 67, 186, 483–485, 597–598
order in which they print 188
printing at Grand Totals time 207, 483
printing the number of items in a control group 198, 489
which columns included in 148, 339, 502, 509, 523
which ones print at end of report 209
Statistics
breaking totals down 217
counting occurrences 214
for individual fields, how to print 191, 491
VSAM record count wrong 424
(see also Statistical lines)
STCKADJ parm
in OPTIONS statement 274, 575, 613, 616
STCKDATE
data type 575, 613
STCKTIME
data type 274, 575, 616
STDLABEL 533
STEPLIB DD 415
Stop
reading the input file, early 551, 576
the run if I/O error 569, 586
STOPWHEN parm
general 591
in FILE statement 536
in INPUT statement 551, 661
in OPTIONS statement 551, 576
overriding 549
Storage
used for sort 420, 436, 568, 575
Stringing fields together 48, 474
Subheadings (see Headings)
SUBLIB parm
in COPY statement 518, 520
in OPTIONS statement 383, 439, 518, 576
Sublibrary (VSE) (see Librarian (VSE) and Copy library)
Subroutines
(see Data exit programs and I/O Exit)
Subscripts, in Web reports 324
Subset
subsetting a mainframe file 283
SUBSTR built-in function (see #SUBSTR built-in function) 634
Subsystem
which DB2 subsystem 393, 409, 561
Subtraction
blanks required around minus sign 473
from a date field 221
how to perform 47, 473, 506
subtracting days, weeks, months or years from a date 639
subtracting seconds, minutes or hours from a date-time 639
subtracting seconds, minutes or hours from a time 640, 641
SUMMARY parm
in OPTIONS statement 73, 113, 214, 576
Summary reports
counting the number of occurrences 214
how to produce 73, 209, 576
summary PC files 110, 113
Superscripts, in Web reports 324
Suppressing
all column headings 133, 134, 175, 279, 283, 567
automatic copying from copy library 364, 545, 582
blank lines 253, 257, 573
blank lines at Grand Totals 279, 283, 567

blanks between fields 560, 631
carriage control character 279, 283, 415, 567
decimal digits in numbers 339, 452
detail report lines 73, 209, 216, 561, 576
error indicators 646
individual column headings 132, 504
leading zeros 139, 455, 619
lines with only zero values 249, 573
message when maximum lines/pages printed 567
overprinting 568
page breaks 134, 175, 568, 577
repeating values 129, 144
the Grand Totals 209, 279, 283, 567
the letter "S" when only one item 198, 489, 625
the total line at control breaks 185, 196, 487, 599
titles 134, 175, 279, 283, 568
totals for certain columns 148, 326, 338, 339, 502, 509, 523
underscore lines 133, 134, 504, 568
zeros 129, 167, 192, 490, 503, 605
SWALIAS
member in copy library 367, 422, 518, 519, 648
SWCOPY DD 382, 414, 422, 423, 517
SWLIST DD 414, 446
SWOPTION DD 414, 424
SWOUT002, SWOUT003, etc. DD 414, 419, 571
SWOUTPUT DD 414, 415, 417
blocksize 567
SYNCHRONIZED parm
in Cobol 243
Syntax
convention used 478
general rules 443
of computational expressions 472
of conditional expressions 459
of control statements 478
syntax-checking HTML 297
SYS010 427, 433
SYS011 427, 429, 431
SYS012, SYS013 etc. 427, 435
SYSIN DD 414
SYSIPT 427
SYSLST 431
SYSnnn associated with input files 533
SYSOUT DD 414

T

Tab character
as delimiter in output files 278, 560
Tables
HTML tables 312, 324
in COMPUTE statement 513, 656
Tabular
report format 219
Tapes
standard/nolabel (VSE) 533
tape files, used as input 329, 331, 332, 533, 536, 551
which tape drive 332
writing output to tapes (VSE) 432, 570
Telephone numbers
how to define 328
how to format 326, 340, 632
Testing
a bit field 465, 471, 514
for missing records 230
for valid data 257
one or more conditions 459, 470
records for inclusion in report 40, 449, 540
Thousands
rounding to 453, 638
Tie-breakers
used in sort 559, 597
TIME built-in field (see #TIME built-in field) 625
Time fields
comparing 462
conversion from GMT to local time 575
creating your own 450, 506
current system time 625, 627
decimal digits 346, 512, 525, 613
default lengths 613
how to define 344
testing for valid data 464, 472, 647
totalling 150
with hundredths of seconds 346
(see also Times)
Time of day
built-in field 625
converting to proper time of day 641
TIME24 built-in field (see #TIME24 built-in field) 625
TIMEDELIM parm
in OPTIONS statement 139, 140, 576
TIMEEXIT data type 616
Times
12-hour format 347, 622
adding/subtracting seconds, minutes or hours 640, 641
comparing 44, 273, 344, 450, 470
converting character data to time value 274, 641
converting numeric values to times 274, 637, 641

converting to character value 632
converting to numeric value 274, 463, 636
decimal digits 273
default display format 139, 562, 618
defining time fields 521
delimiter used 139, 576, 613, 622–623
extracting the hours, minutes and seconds portions 635, 637, 638
formatting in report 58, 137, 167, 192, 273, 458, 503, 622
handling invalid times 644
how many seconds in 636
how sorted 600
how time fields stored in input file 529, 613
in COBOL and Assembler record layouts 375, 379
including only certain times in a report 44, 470
including time of day in footnotes 175
including time of day in titles 163, 604
interval between two times 636
interval, did it cross into next day 639
on different days, computing interval between 274
performing calculations 472
rounding to minutes 58, 347
selecting the largest of several times 637
selecting the smallest of several times 637
showing AM and PM 347, 622
tips for using time fields 272
totalling 150, 273, 346, 502, 509, 523
writing time literals 44, 450
zero times, printing blanks 129, 167, 192, 490, 503, 605
zeros assigned for missing fields 229
TITLE statement 602
alignment (left, center and right) 55, 168, 603, 606
blank titles 153, 154
built-in fields available for 604
centered by default 53
centered data looks wrong 168, 173, 606
centering, in Web reports 298
how dates, times and numbers are formatted 167, 605
how to use 53, 161
in Web reports 298, 305, 308
including data from files 161, 604
including date, time, page number in title 53, 55, 163, 604
justifying contents of fields 165, 168, 606
leading, trailing slashes 153, 169
multiple 53
omitting 53
overlap 173
parms allowed in 167
printing in a certain column 173
right aligned part looks wrong 173, 606
spacing between items 165, 167, 605
specifying column headings with 153, 169
specifying width of fields 165, 168, 606
spelling out month name 55, 58, 167
suppressing titles 568
syntax 603
that won't fit on a single line 444
underlining 153
use of quotation marks, apostrophes 53, 163, 444, 448, 605
use of slash for alignment 55, 168, 603
where to put 53, 445
(see also Titles)
TITLEONCE parm
in OPTIONS statement 134, 175, 300, 321, 577
Titles
how to specify 53, 161
in Web reports 298, 305, 308, 312, 564
not at top of PC screen 298
options, summary of 175
printing at bottom of page 175, 538
printing just once 134, 175, 577
printing lower on page 154
putting graphics in title 305, 312
saving data from titles in input files 260
suppressing 134, 175, 279
(see also TITLE statement, FOOTNOTE statement and Column headings)
TLBL statement 332, 432
(see also DLBL statement)
TODAY built-in field (see #TODAY built-in field) 626
Top margin, how to specify 154
Top of page
printing heading lines 202, 207, 486
Top of report
putting lines before 309, 318
Top ten type reports 212, 561
TOTAL (TOT) parm
in BREAK statement 182, 198, 207, 487
in BREAK statement print expression 193, 196, 200
in BREAK statement, two different uses 194
in SORT statement 177, 204, 210, 599
Total line
S appears in 182, 454
aligning in Web reports 303, 306, 316
customizing 182, 196, 198, 487

display format used for 140, 491
how default total line looks 180, 487
how default total line looks in PC file 108
how to print 65, 177, 487, 599
how to suppress 185, 487, 599
in Web reports 301, 303, 309, 564
justification used in 491
level indicated by asterisks 206
multiple levels 69, 204, 210
multiple total lines at control breaks 184
percentages for control group 202, 510, 515
PICTURE can prevent totalling 148
printing blank lines after 67, 178, 486, 598
printing blank lines before 184
printing only the total lines in a report 73, 211, 216
printing the current date in 185
printing the number of items in a control group 182, 198, 489
suppressing, for a particular column 148, 326, 502, 509, 523
totalling time fields 150, 273, 346, 502, 509, 523
using footing instead of total line 196
where it prints at control break 184
which columns are totalled 128, 148, 326, 338, 339, 502, 509, 523
why split into two lines 181, 303
(see also Totals and Grand totals)
Totals
breaking down 217
customizing the total line at control breaks 487
how to print an individual field's total 193, 491
how to request 487, 599
percent of totals 284
(see also Total line and Grand totals)
TPICTURE
display format 622
how to write 273, 458
Trailer records, in batch type files 237
Trailing
blanks, removing 631
plus or minus sign 452
slash, in TITLE statement 153, 169
TRANSLATE built-in function (see #TRANSLATE built-in function) 634
Translation
between ASCII and EBCDIC 631
TRIPLESPACE parm in OPTIONS statement 573
True, bit value 643
Truncation
how to perform 512
of column headings 135
of columns 135, 160
of decimal digits (#INT built-in function) 635
what to do 160, 432
Type
member type of VSE library 566
TYPE parm
choosing character versus numeric 339
comparing fields of different types 340, 450, 463
converting field to different type 463, 632, 637, 641
in FIELD statement 335, 341, 344, 358, 529
in FILE statement 329, 536
in INPUT statement 551
in READ statement 590
list of data types 609
types of data 333, 448

U

UCASE built-in function (see #UCASE built-in function) 634
Undefined field indicator (***U***) 645
Underlined font
in Web reports 294, 303, 324
Underscore (_)
name broken at, for column headings 130, 504
printing in titles 153
suppressing in column headings 132, 300, 504, 568
suppressing overprinting 133, 134, 568
use in field names 446
Unique
field names, how to make 77, 226, 228, 550, 589
file key not unique 232
Unsigned numeric data 610
UNSTRING (see under #PARSE)
Upper case 634
User-defined fields 46, 98, 506

V

Valid
data, testing for 464, 472, 647
validating HTML 297
Values
comparing contents of fields 40, 460, 470
including only certain values in report 40, 460, 540

Variable
location in record 266, 353, 528
number of report lines per input record 249
Variable length files
clearing the I/O area 545, 567, 581
defining 533
record descriptor word (RDW) 352, 353, 524, 535, 547, 565
Vertical bar
printing a vertical line between columns 156
use in column headings 130, 350, 504, 527
use in conditional expressions 467
using a different character 132, 133, 563
Video clips, putting in report 294, 308
VSAM files
alternate indexes, paths 232
defining 329, 331, 536
key greater than or equal to 583
keyed reads to 76, 224
missing records 229, 230, 424, 433, 594
must be KSDS for READ statement 590
reading a limited keyrange 547, 660
reading multiple records 224, 584
specifying BUFND 423, 433, 544, 581
specifying BUFNI 423, 433, 544, 581
speed-up tip 423, 433, 547, 658
testing for missing records 642
used as input (OS/390) 329, 551
used as input (VSE) 331, 533
using generic keys 547, 583
VSAM-managed SAM files 331, 533
writing output to (OS/390) 417, 571
writing output to (VSE) 432, 570
VSE operating system 425

W

WAV files 308
Web reports 294
adding hot links 308, 322
aligning column headings 316
aligning Grand totals 303, 306, 316
aligning text and graphics 305, 312, 323
aligning titles 300
audio and video clips 308
background 319
bold font 296, 298, 301, 322
centering text 298, 322
colored font 298, 319, 322, 323
column headings 301, 306, 316, 564, 565
different colors in one column 315
dynamic HTML 315
file name extension 296
font, specifying 301, 319, 322, 323
HTML tables 312
including graphics 305, 306, 323
including graphics at control breaks 308
including graphics in titles 305, 312
summary of options for 320
titles 296, 308, 564
titles not at top of PC screen 298, 300
total lines 301
Week
adding/subtracting weeks to/from a date 639
calculating any day of week in a given week 638
day of (see Day of week)
WHEN parm
in COMPUTE statement 50, 459, 513
order of evaluation 508
WHERE parm
in INPUT statement 397, 405, 552
in READ statement 400, 584, 587, 590
reading multiple rows 584
similar to DB2 WHERE clause 397
syntax 405
testing if no records found for it 642
Whole
numbers, how to round out decimal digits 339, 452, 512, 638
Width
of column, changing 60, 129, 135, 505
of computed fields 511
of fields in output files 282
of graphics in Web reports 323
of items in lines printed at control breaks 182, 193, 491
of items in the title 165, 168, 606
of numeric data in report, specifying with a PICTURE 452
of report, bigger than 132 characters 432
of report, maximum 160
Window, century windowing 269
Words
counting words in a string 638
parsing a character string 633
searching for, within a string 461, 635
Work files
sort 414, 434, 435, 574
sort (VSE) 575
Worldwide Web (see Web reports)

X

X
  meaning of X'1234' type literals 448, 464
XOR built-in function (see #XOR built-in function) 635

Y

Year
  2-digit or 4-digit 278, 282, 449, 559, 613, 620
  adding/subtracting years to/from a date 639
  calculating first & last days of a year 639
  calculating previous 222
  converting numeric day, month and year into a date 640
  extracting for a given date 635, 638
  maximum year allowed in literals 449
  testing for leap year 642
  which century 559
YEAR built-in function (see #YEAR built-in function) 635
YEARNUM built-in function (see #YEARNUM built-in function) 638
Yesterday
  computing yesterday's date 639
Yesterday, computing yesterday's date 266
YMD see #YMD built-in function) 640
YY
  year in dates, which century 269, 559
YYDDD
  date fields 612, 613, 621–622
  Julian dates 271
YYMMDD
  date fields 611, 612, 621
YYYYDDD
  date fields 612, 613, 621–622
YYYYDDMM
  date fields 612
YYYYMMDD
  date fields 611, 612, 621–622

Z

Zero
  assigned to missing date, time and numeric fields 229, 594
  division by zero 645
  division by zero, suppressing 577, 646
  excluding zero values from averages and minimums 186, 485, 491, 598
  leading zero in date literals 449
  leading zero suppression 139, 455
  leading zeros, printing 455
  number scaled down to zero 455
  padding 472
  printing blanks instead of zeros 257, 632
  records included in report 562
  spaces between items in title 605
  spaces between items in total line 198, 490
  spaces between report columns 128, 502
  suppressing lines with only zeros 249, 573
  treating invalid data as zeros 577
  value, in date 271
ZERODIVBYZERO parm
  in OPTIONS statement 577, 646
ZEROINVDATA parm
  in OPTIONS statement 254, 577, 646
ZEROOVERFLOW parm
  in OPTIONS statement 577, 646
Zoned data 387
  C versus F for packed data 630, 633