CIMS Lab, Inc.

CIMS Chargeback

Report Writer User Guide

Version 2.7.1

Supports CIMS/MVS 11.3
and CIMS/VSE 10.2

Title and Publication Number

CIMS Lab Publication Number: RW-UG-271-01

Printed: November, 1999

Information in this guide is subject to change without notice and does not constitute a commitment on the part of CIMS Lab, Inc. It is supplied on an "as is" basis without any warranty of any kind, either explicit or implied. Information may be changed or updated in this guide at any time.

Mailing Address

CIMS Lab, Inc.
3013 Douglas Blvd., Suite 120
Roseville, CA 95661-3842

Preface

As companies continue to integrate computer technology into their business operations, it becomes increasingly important to properly administer the IT function, particularly with respect to performance and cost. And the best way to control costs is to plan for them.

CIMS Chargeback is a comprehensive, flexible software solution that consolidates a wide variety of accounting data for multiple operating systems into a single file that may be accessed from either the mainframe or a workstation. Simply put, CIMS Chargeback is an essential component of an effective financial management system.

Philosophy

Originally developed in 1974, CIMS has focused on meeting the financial and resource reporting requirements of Information Services Departments. CIMS has evolved with corporate IT management requirements. Focused commitment to client service and support sets CIMS apart from competing products. Our goal is to provide the best chargeback and resource reporting software in the world at the lowest possible cost to our customers.

The CIMS Lab strongly believes in and executes the concept of continuous product improvement. Customers have access to CIMS product development personnel to ensure that customer feedback and other critical issues are incorporated into the next release of the product.

Contacting the CIMS Lab

You can contact us with any questions or problems you have. Please use one of the methods below to contact us.

For product assistance or information, contact:

USA & Canada, toll free	(800) 283-4267
International	(916) 783-8525
FAX	(916) 783-2090
World Wide Web	**www.cimslab.com**

Our Mailing Address is:

CIMS Lab, Inc.
3013 Douglas Blvd., Suite 120
Roseville, CA 95661-3842

Related Publications

As you use this guide, you might find it helpful to have these additional books available for reference:

- *CIMS Chargeback MVS User Guide*
- *CIMS Chargeback CICS User Guide*
- *CIMS Chargeback VM/CMS User Guide*
- *CIMS Chargeback VSE User Guide*
- *CIMS Chargeback Report Writer Sample Reports for MVS*

Table of Contents

List of Figures

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 Report Writer. To learn how to use Report Writer, we suggest the following steps:

Step 1. Read **Chapter 1, "Introduction"** to learn just what Report Writer is and what it can do for you.

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 Report Writer.

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 Report Writer.

Step 4. Start producing your own reports and PC 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 Report Writer and defining your input files, also read **Chapter 5, "How to Define Your Input Files"** and **Chapter 7, "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 Report Writer. The User's Guide contains over 100 examples of actual Report Writer runs. It also explains how to define files and setup the JCL needed to execute Report 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 Report 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 Report 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.

How This Manual is Organized

Chapter 1, "Introduction"
This chapter explains just what Report 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 Report Writer how to produce a report. Everyone who will be producing reports with Report 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 Report Writer how to produce a PC file. Everyone who will be producing PC files with Report 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 Report 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 Report Writer is capable of doing. You may find that you can use Report Writer to produce reports that you thought were too complicated for a Report Writer.

Chapter 5, "How to Define Your Input Files"
This chapter shows how to define your company's files to Report 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 Report Writer file definitions should read this chapter.

Chapter 6, "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 Report Writer with a special database.

Chapter 7, "Operating System Considerations"
This chapter explains what "job control language" (JCL) is necessary to run a Report Writer job under different operating systems. The analyst or programmer responsible for setting up the JCL to run Report Writer should read this chapter.

Chapter 8, "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 9, "Control Statement Syntax"
This chapter shows the complete syntax for each of Report 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 Report Writer supports in input files.

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

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

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

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

Appendix F, "Sample File Definitions"
This appendix shows the Report Writer file definitions (and the raw contents) of the sample files used for the examples in this manual.

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

Appendix H, "How to Import PC Files"
This appendix shows the exact steps used to import PC files into a number of popular PC programs.

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

Appendix J, “Year 2000 Information“
This appendix discusses issues related to the Year 2000.

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

(This page left blank intentionally.)

Part 1.
User's Guide

Chapter 1. Introduction

Chapter Table of Contents

Chapter 1. Introduction

What Is Report Writer?

Report Writer is three powerful programs in one.

1) It's an easy–to–use, full function **4GL report writer.**
2) It's a powerful **PC–format utility.** Use its 4GL language to easily turn any mainframe data into PC files that can be used in all popular PC programs!
3) 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

Report Writer makes it *easy* to produce custom reports from your company's existing files. Programmer productivity increases dramatically with Report Writer.

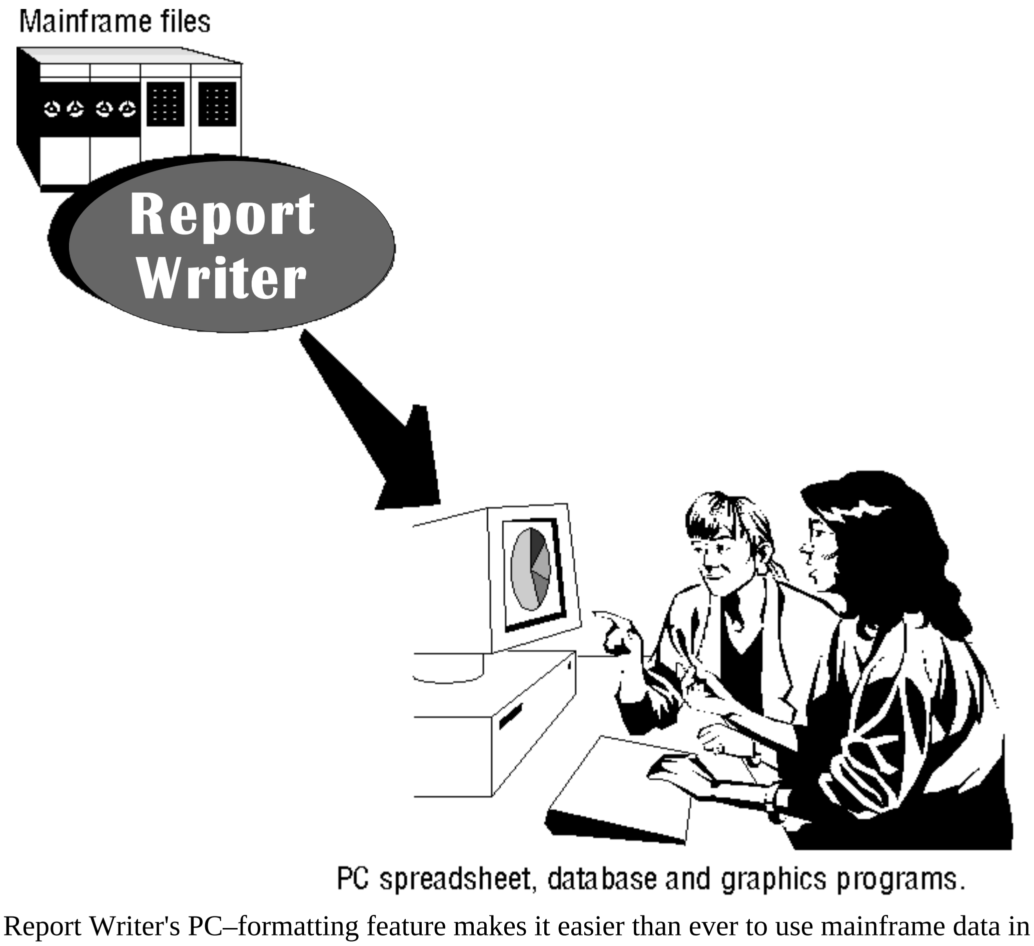
To produce a new report without Report Writer, a programmer has to write a new program in a language such as COBOL. The programmer must code all of the I/O routines, the selection logic, the computations, summarization, sorting, formatting, page breaks, titles, column headings, etc. The process of coding, 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 Report Writer. With Report Writer, you no longer need to write detailed programming instructions. You simply describe the desired report to Report 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 Report Writer using only *two* statements. Try that with COBOL! Add a few more statements and you can produce more complex reports.

With Report 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 Report Writer.

Report 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 Report Writer.

Use Mainframe Data in Any PC Program



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

Use Mainframe Data in Any PC Program

Report 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. Report 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 Report Writer comes in.

Report 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 Report Writer reformats these fields into standard ASCII data that your PC program can use.

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

Create Custom Mainframe Files in Minutes

Report Writer creates mainframe output files just as easily as PC–formatted 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 Report Writer write out the desired data in any format you choose. Use Report 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 Report Writer Benefits You!

Here are a few examples of the ways that Report 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 Report 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 Report Writer to quickly produce your regular *production reports*. Its usefulness is not limited to just ad hoc reports.

Fast One–Time Queries

Report 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 Report Writer. You'll wonder how you ever got along without it.

Provide Reports for CICS Systems

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

Save Money on Special Analyses

Without Report 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 Report 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 encourage programmers to develop new applications using them. Because Report Writer is written entirely in efficient assembly language, your reports run amazingly fast.

In many cases, there is no significant difference between Report 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.), Report 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, Report 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.

Report 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 LAN use
- print high quality graphics on laser printers
- create color graphics, overhead transparencies and slides for fabulous presentations

Report 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 Report Writer for one–time file conversions needed when downsizing mainframe applications to run on PC systems. Report 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

Report Writer reduces download time and hard disk usage by letting you download only the data you actually need (not the entire mainframe file.) Why tie up a PC for hours 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 actually needs. Report Writer lets you do the extraction on the mainframe, before you download the data.

Save Wasted Employee Time Caused by Slow PC Processing

Report Writer eliminates hours of needless PC processing by moving much of that processing from the PC to the mainframe. Here's a few of the ways Report Writer lets your PC users zip along rather than idling over slow PCs.

No more waiting on 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 Report Writer summarize it on the mainframe. Then just download the small summary file to your PC.

Rather than wait on your PC to **compute** new columns in your spreadsheet, let Report Writer create the new columns on your mainframe. Then download them along with the other mainframe data.

Disk I/O is slow on PCs. So why merge data from multiple files on your PC? Use Report Writer to combine data from multiple mainframe files (or DB2 tables) into a single file before you download it to the PC.

Avoid the Million Dollar Mistake!

You've probably read one of those sad stories in the computer magazines. In order to save a little money, a company decides to continue using a manual process rather than automate. Someone manually keys data from a report into a PC spreadsheet. Sure it takes that person an hour or more a day to type it all in, but the company saves the money that the automated software would have cost. This slow manual process works well enough for a while. Then one day— disaster! A critical number is typed in wrong and goes unnoticed. The spreadsheet computes a bid or some other critical figure based on the incorrect data. A bid goes out the door that could cost the company hundreds of thousands, even millions, of dollars. All because they "couldn't afford" to automate by buying the right tool. Report Writer lets you avoid making the Million Dollar Mistake!

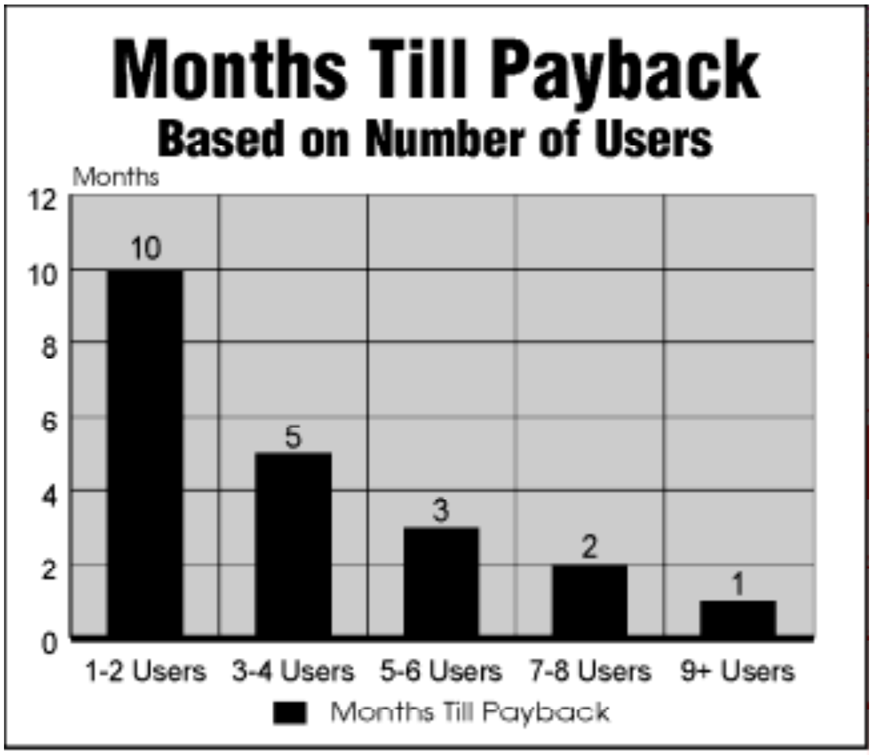
Report Writer Pays for Itself Fast!

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

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

Report 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, Report 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 slow sorting and number–crunching functions 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 Report Writer, there are no expensive PC components to purchase and maintain. All you need is Report 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 small investment in Report Writer.



Conservative Assumptions Used: Group 38 MVS CPU; salary and benefits total $50/hour; used 3 hours per week per employee; development time is half that of COBOL.

Report Writer Features

Here are some of Report Writer's major features:

- Year 2000 ready
- 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 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 Report Writer's simple data dictionary for added functionality.
- no data definition required for DB2 tables — Report Writer accesses the definition from your DB2 system
- produces efficient internal machine code that is easy on your CPU
- produces output files for mainframe use, as well as PC files
- report lines are not limited to only 132 characters. Report Writer can format a report as wide as your laser printer will support.
- automatically prints bar graphs
- ability to print full–page forms
- ability to skip to a new 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 results in at least a partial report almost every try
- has thorough, clear documentation, including a User's Guide in non–technical language for end–users
- validity–checks numeric data before processing it, so that no S0C7 abends occur
- ability to display file data in hexadecimal format, for analyzing invalid data
- translates fields from ASCII to EBCDIC 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 between *any* numeric fields, (even if one is packed and one is binary, for example.)
- comparisons are allowed between *any* date fields (even if one is Julian and one is Gregorian, for example.)
- supports dates with 2–digit or 4–digit years

- supports your 2–digit years even *after* the year 2000, with its century windowing feature
- 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 parse and compress features
- "compress" formatting features lets you, for example, compress separate city, state and ZIP fields into a normal address 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 mainframe 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 per report or PC file
- 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
- user exit interfaces for any special handling required at the field level or record level
- prints end of job statistics, such as how many records read from each input file, and how many records included in report

(This page left blank intentionally.)

Chapter 2. How to Request a Report

Chapter Table of Contents

Chapter 2. How to Request a Report

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

Report 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 Report Writer. (See page 19.)

The box on page 15 lists all of the Report Writer control statements, and tells you which aspect of the report each one deals with. The lessons in this chapter illustrate how these control statements work.

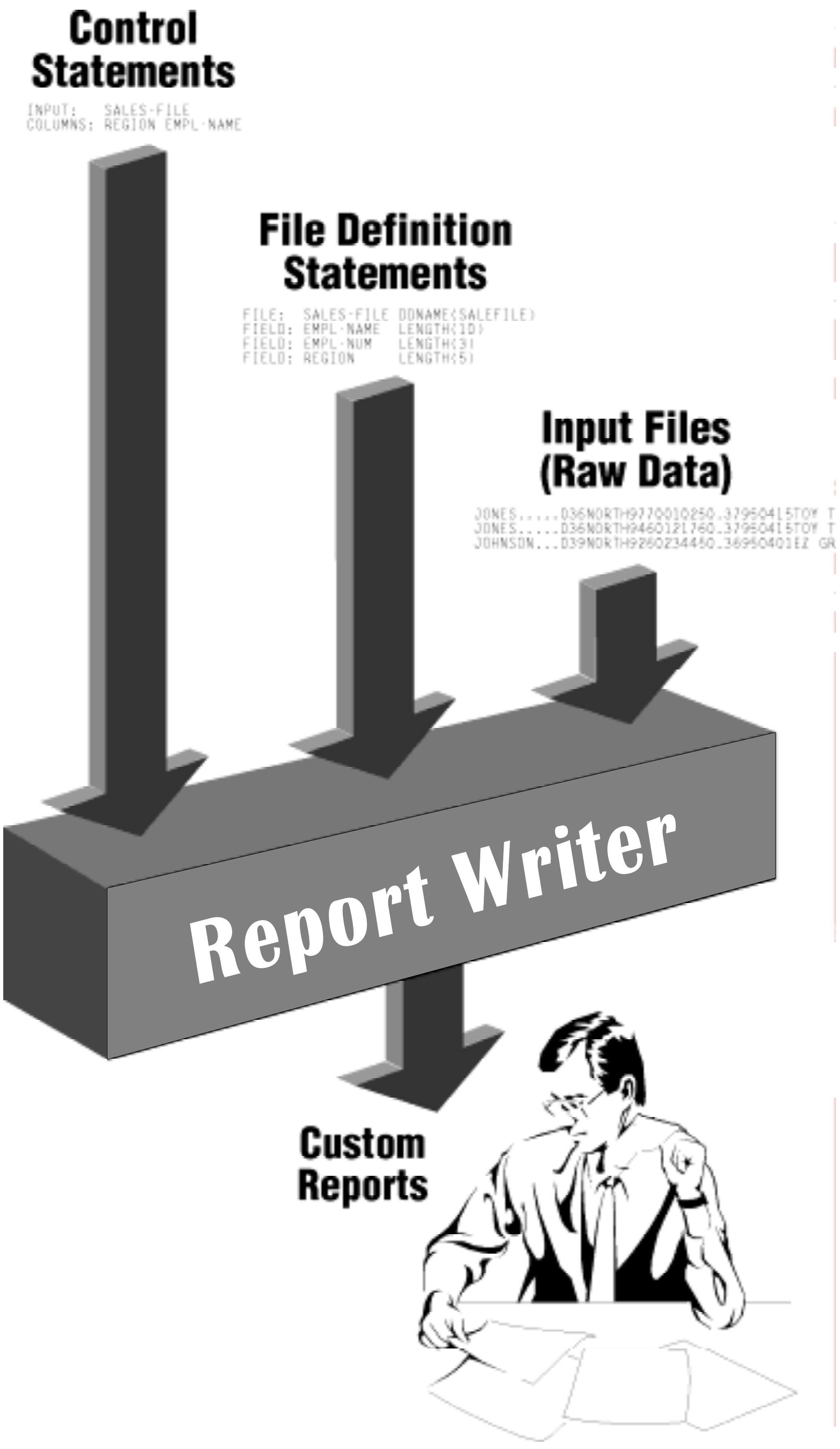
Once you've written the necessary control statements, submit a batch job to execute Report Writer. Report Writer examines the control statements describing the report you want. It also automatically reads the appropriate "file definition" statements stored in a copy library. (These statements define the input files needed for your report.) Report Writer then accesses the input file(s) and prepares the desired report. Reports can be sent directly to a printer. Or, use your company's sysout browsing facility (such as ISPF, IOF or POWER) to view your report online, as soon as it is finished.

The remainder of this chapter is divided into nine easy lessons that explain how to use Report Writer's control statements to create custom reports. After reading just the first lesson, you will be able to produce useful reports with Report 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.



```
INPUT:    SALES-FILE
COLUMNS: REGION EMPL-NAME
```

```
FILE:  SALES-FILE DDNAME(SALEFILE)
FIELD: EMPL-NAME  LENGTH(10)
FIELD: EMPL-NUM   LENGTH(3)
FIELD: REGION     LENGTH(5)
```

```
JONES.....036NORTH977001025O.37950415TOY T
JONES.....036NORTH946012176O.37950415TOY T
JOHNSON...039NORTH926023445O.36950401EZ GR
```

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 9, "Control Statement Syntax."

REPORT WRITER CONTROL STATEMENTS
(GROUPED BY FUNCTION)

Statements that Define Data

`FILE`	Defines a file
`FIELD`	Defines a field within a file
`ASM`	Lets you define a file using an Assembler record layout
`COBOL`	Lets you define a file using a Cobol record layout

Statements that Make Data Available to a Report

`INPUT`	Specifies the primary input file
`READ`	Specifies an auxiliary input file
`COMPUTE`	Creates a new field

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
`COPY`	Copies additional control statements for processing

(This page left blank intentionally.)

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 Report Writer. For example:

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

Figure 1 (page 19) 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 Report 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 Report Writer that you want to use a file named SALES–FILE as the input for your report.

All Report 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 SALES-FILE.

> **Note:** 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, including the name of the employee who made the sale, their employee number, their sales region, the date and time of the sale, the customer's name, the amount of the sale, and so on. Each of these items of data is called a *field*. A complete description of this sample SALES–FILE is shown in Appendix F, "Sample File Definitions" (page 587.)

How to Use the COLUMNS Statement

After identifying the input file to use, the next step is to tell Report 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 Report 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 Report Writer everything it needs to produce a report. The report produced is shown in **Figure 1**.

You now see how *easy* it is to produce reports with Report 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 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
```

Notes:

- 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

Figure 1 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, described in Appendix F, "Sample File Definitions" (page 588.) 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 2**.

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

Using Your Company's Files

You may be wondering how Report Writer knows the names of *your company's* files and fields. The answer is that your company's files are defined to Report Writer by other control statements that are kept in a Report Writer "copy library." For example, the statements used to define the SALES–FILE that we used earlier in this lesson are shown on page 587.

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 Report Writer Copy Library, in a format similar to that shown on page 587.

If you already know the name of the file to use, you can also get a list of all of its fields by adding the SHOWFLDS(YES) parm to your INPUT statement like this:

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

The above statement tells Report 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 has not yet been defined, see Chapter 5, "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)
```

Notes:

- 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 2 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 Report Writer which input file to use for a particular report
- a COLUMNS statement is needed to tell Report 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 8, "General Syntax Rules." In that chapter you will learn such things as:

- **how long** each line can be (page 384)
- how to **continue** control statements onto multiple lines (page 385)

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 this chapter in Lesson 5, "Changing the Format of Your Report". Other topics are discussed in Chapter 4, "Beyond the Basics." The additional features include:

- how to specify your own **column headings** for a report (pages 44 and 127)
- how to make a column in the report **wider or narrower** (pages 46 and 131)
- how to change the way that **numbers, dates and times are formatted** in your report (pages 44 and 132)
- how to make a report column that contains a **literal text** (page 124)
- how to specify the number of **spaces** to leave between columns in your report (page 124)
- how to specify which numeric columns to include in the **Grand Totals** (page 144)
- how to print **multiple report lines** for each input record (page 147)
- how to produce reports that are **wider than 132 characters** (see page 362 or 374)

The complete syntax for the INPUT and COLUMNS statements is given in Chapter 9, "Control Statement Syntax."

(This page left blank intentionally.)

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, Report Writer defaults to including every record from the input file. For example, the report on page 19 included all *sales* from the SALES–FILE. And the report on page 21 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 Report 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 19. 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 Report Writer to "*include*" records from the SALES-FILE "*if*" the EMPL–NAME field is equal to 'JONES'. Report 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 3** 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. You are *not limited* to just those fields that are listed in the COLUMNS statement.

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
```

Notes:

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

Figure 3 Using an INCLUDEIF statement to specify which records to include in a report

How to Write Conditional Expressions

The INCLUDEIF statement consists of a **conditional expression**. The complete rules for writing conditional expressions are explained beginning on page 399. 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 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 is true. **Figure 4** 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 beginning on page 389).

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
```

Notes:

- 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 4 Including records in a report if either of two conditions is true

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 Report 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
```

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.

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.)

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.

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 Chapter 9, "Control Statement Syntax," beginning on page 481. The additional features include:

- how to use **symbols** rather than the actual words AND and OR in your conditional expressions
- how to **scan** a character field, to see if a certain text exists *anywhere* within the field
- how to specify conditions based on **bit fields**
- how to specify a condition based on a field's raw **hexadecimal** value
- how to specify date literals in DD/MM/YY or DD/MM/YYYY format (page 137), like this:

  ```
  INCLUDEIF: SALES–DATE > 15/4/1995
  ```
- you may also be able to use the **KEYRANGE parm** of the INPUT statement to limit the records included in your run (page 485)

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 beginning on page 410. Generally, your expression will consist of one or more mathematical 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 5** shows a report that uses the above COMPUTE statement.

> **Note:** COMPUTE statements normally appear *after* the INPUT statement, but must appear *before* any other control statements that refer to the field being created. In the example on page 31, 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 may appear to be a part of a field name. Blanks are optional around the other operator symbols.

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
```

Notes:

- 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 5 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 5 (page 31) shows a report that uses the COMPUTE statement shown above.

In addition to the basic arithmetic operations, there are also a 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 566.)

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.

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)
```

Notes:

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

Figure 6 Using the COMPUTE statement to create character fields

There are a number of other built–in functions which can also be used. A complete list of built–in functions is found in Appendix D, "Built-In Functions" (page 566.)

Figure 6 (page 33) shows a report that uses each of the COMPUTE statements shown in the preceding examples.

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, Report 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.

You are allowed to 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.

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
```

Notes:

- 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 7 Assigning values to computed fields based on conditions

Figure 7 (page 35) 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* will be 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 Lesson 2 on page 26.) The complete rules for writing conditional expressions are given beginning on page 399. Additional examples of COMPUTE statements are shown beginning on page 451.

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 the 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 9, "Control Statement Syntax" (page 444). Other additional features are discussed in Chapter 4, "Beyond the Basics." Examples of the additional topics include:

- how to create **date** type fields (page 452)
- how to create **time** type fields (page 254)
- how to create **bit** type fields (page 452)
- how to specify how many **decimal places** a numeric or time field should contain (page 450)
- how to specify **column headings** for the fields you create (page 449)
- how to specify how your field should be **formatted** when it is printed in a report (page 447)
- how to specify whether a numeric or time field should be totalled in the **Grand Totals** line at the end of the report (page 144)
- how to **retain the value** of a COMPUTE field in certain cases (page 238)

(This page left blank intentionally.)

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, Report Writer provides a default title.

To specify your own report titles, simply use one or more TITLE statements. For each TITLE statement you supply, Report 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 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 page 385 for more information on continuing control statement.

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 Report Writer's built–in field names begin with this character. That is to 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. The words #TODAY and #PAGENUM are the names of *fields,* whose *contents* we want to print in the title. 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 8 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
```

Notes:

- the report now 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 8 Using the TITLE statement to specify your own titles

More Date and Time Features

When you use #TODAY in your title, Report 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 9** uses the LONG1 display format. The use of LONG1 and other display formats is discussed in more detail beginning on page 170. For a complete list of display formats to choose from when formatting dates in your titles, see Appendix B, "Display Formats" (page 550.)

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 9 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 – EMPLOYEE DIRECTORY'  /  '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 9 shows a sample report that illustrates the use of slashes to align 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
```

Notes:

- 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 9 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 Report 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." Examples of additional features include:

- how to include **data from the input file** in your title (page 55 and 165)
- how to change the way the **dates, times and numbers are formatted** in the title (page 170)
- how to use **any combination** of left aligned, centered, and right aligned title parts (page 174)
- how to print "footnotes" at the **bottom** of each page of the report (page 180)

The complete syntax for the TITLE statement is given in Chapter 9, "Control Statement Syntax" (page 531.)

(This page left blank intentionally.)

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

Report 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 550.) When no display format is specified (as in most of the examples in the previous lessons), Report 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 Report Writer's LONG1 format (that is, as MMMMMMMMM DD, YYYY.)
- the SALES–DATE field 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, 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 10 shows a report that illustrates these display formats.

Specifying Column Headings

Another way to customize your report is with override column headings. You remember that Report 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 10**, 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')
```

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
```

Notes:

- 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 5 bytes wide
- Changes made to the detail line formatting are also reflected in the Grand Total line

Figure 10 Using override display formats, column headings and column widths

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, Report Writer chooses a default column width. You may want a larger column width (to hold larger numeric values, for example.) Of, 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 Report Writer to make the TAX column just 5 bytes wide in the report. This is also illustrated in the report in **Figure 10** (page 45.)

Note: you can specify more than one override for a single field. The order is not important. Just separate the overrides with spaces and/or a comma. For example, the following statement specifies a override column heading and display format and width:

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

Summary

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

- use a **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." Examples of additional formatting features include:

- how to **align** data within its column (page 142)
- how to **blank out repeating values** (page 140)
- how to **blank out zero values** (page 126)
- how to change the **spacing between columns** in a report (page 124)
- how to use a **character other than the vertical bar** (|) to separate column headings into multiple lines (page 127)
- how to change the default display format for **all fields** in a report (page 500)
- how to format reports using **international (non–USA) conventions** (see page 137)

(This page left blank intentionally.)

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, Report 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 all appeared in sales date order. (For example, see the report on page 19.) 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 (page 21.)

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. Report 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 11** shows a report produced with the above statement.

By default, Report 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 Report 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
```

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

Figure 11 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." Examples of additional features include:

- creating a **control break** from the SORT statement (page 182)
- specifying **control break spacing** from the SORT statement (page 183)
- requesting **totals and statistics** in the SORT statement (page 194)

The complete syntax for the SORT statements is given in Chapter 9, "Control Statement Syntax" (page 524.)

(This page left blank intentionally.)

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 11** (page 49) 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:

```
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 12 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
```

Notes:

- 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 12 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 Report Writer what kind of spacing to perform at the control break. By default, Report 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 13** illustrates the use of the PAGE spacing parm to request a page break.

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)

Notes:

- 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

Figure 13 A BREAK statement that produces 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
```

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

Figure 14 shows a sample report that uses the preceding BREAK 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
```

Notes:

- the AVERAGE and MAXIMUM parms (in the BREAK statement) cause 2 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 14 A report that prints statistical information at control breaks

How to Request Multiple Control Breaks

You may designate *more than one* sort field as a control break field. Report 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 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 15** 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  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
```

Notes:

- 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 15 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 certain **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 ." Examples of additional topics include:

- additional control break spacing parms, including one that skips to a **new sheet of paper** (page 183)
- how to print one or more **customized lines at the beginning** of a control break (page 208)
- how to print one or more **customized lines at the end** of a control break (page 196)
- how to **customize the total line**, and the other statistical lines (page 190 and 194)
- how to **suppress the total line** at a control break (page 193)
- how to print *only* the total lines to produce a **summary report** (page 62 and 218)
- how to compute **percentages and ratios** that apply to an entire control group (page 187)

The complete syntax for the BREAK statement is given in Chapter 9, "Control Statement Syntax" (page 421).

(This page left blank intentionally.)

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 53. 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 16 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
```

Notes:

- this is the same report as on page 53, 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 16 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 190)
- **printing statistics** (such as averages, maximums and minimums) in your summary report (page 194)
- creating **multiple levels** of summarization (page 211)
- printing a **limited number of detail records** in each control group, creating reports such as "The Top 3 Sales in Each Region" (page 220)

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 Report Writer's most powerful features is its ability to use *any number* of input 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 Report Writer processes the primary and auxiliary input files. Report 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, Report Writer also reads one record from each of those files. (These files are always read *randomly*, using a key.) At this point, Report 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 other fields can be used

After processing this set of records, Report 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* whole 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.) Most VSAM files are 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 Report 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 17**, 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.

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
```

Notes:

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

Figure 17 A report that uses only the primary input file

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 21.) 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" to use in reading the EMPL–FILE. We can make the EMPL–FILE an auxiliary input file, then, by simply adding this statement:

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

This READ statement tells Report Writer to use the EMPL–NUM field from each record in the SALES–FILE as a key for reading 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 (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, "Sample File Definitions.") 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 page 232.)

Figure 18 shows a sample 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 use fields from the primary input 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

           SALES
           FILE    SOCIAL
   EMPL    EMPL      SEC      SALES                                     PRODUCT
   NAME     NUM      NUM       DATE      CUSTOMER             AMOUNT      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
```

Notes:

- 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 18 A report that uses a READ statement to specify an auxiliary input file

How to Use Multiple READ Statements

You are allowed to use an *unlimited* number of READ statements in requesting a report. The sample report in **Figure 19** 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 an another auxiliary input file. (The PRODUCT–FILE is described in Appendix F, "Sample File Definitions.")

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 has *three input files*. Data from *all* of these files can be used in any of the subsequent control statements. In the sample report in **Figure 19**, 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)
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

            SALES
            FILE   SOCIAL
    EMPL    EMPL     SEC      SALES                                      PRODUCT     PRODUCT
    NAME    NUM      NUM       DATE       CUSTOMER            AMOUNT      CODE        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
```

Notes:

- 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 19 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**
- 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." Examples of additional features include:

- how to assign a **record name** to the records read from auxiliary input files (page 232)
- how to read more than one record from the **same auxiliary input file** (page 228)
- how to use **data from one auxiliary input file as the read key** to another auxiliary input file (page 230)
- what happens when **no record is found** for a particular read key (page 233)
- how to determine whether the read for a particular key was **successful** or not (page 233)
- how to use the READ statement to obtain data from a **DB2 table or view** (page 338)

The complete syntax for the READ statement is given in Chapter 9, "Control Statement Syntax" (page 510).

Chapter 3. How to Request a PC File

Chapter Table of Contents

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. Report Writer makes using mainframe data in PC programs as easy as 1-2-3.

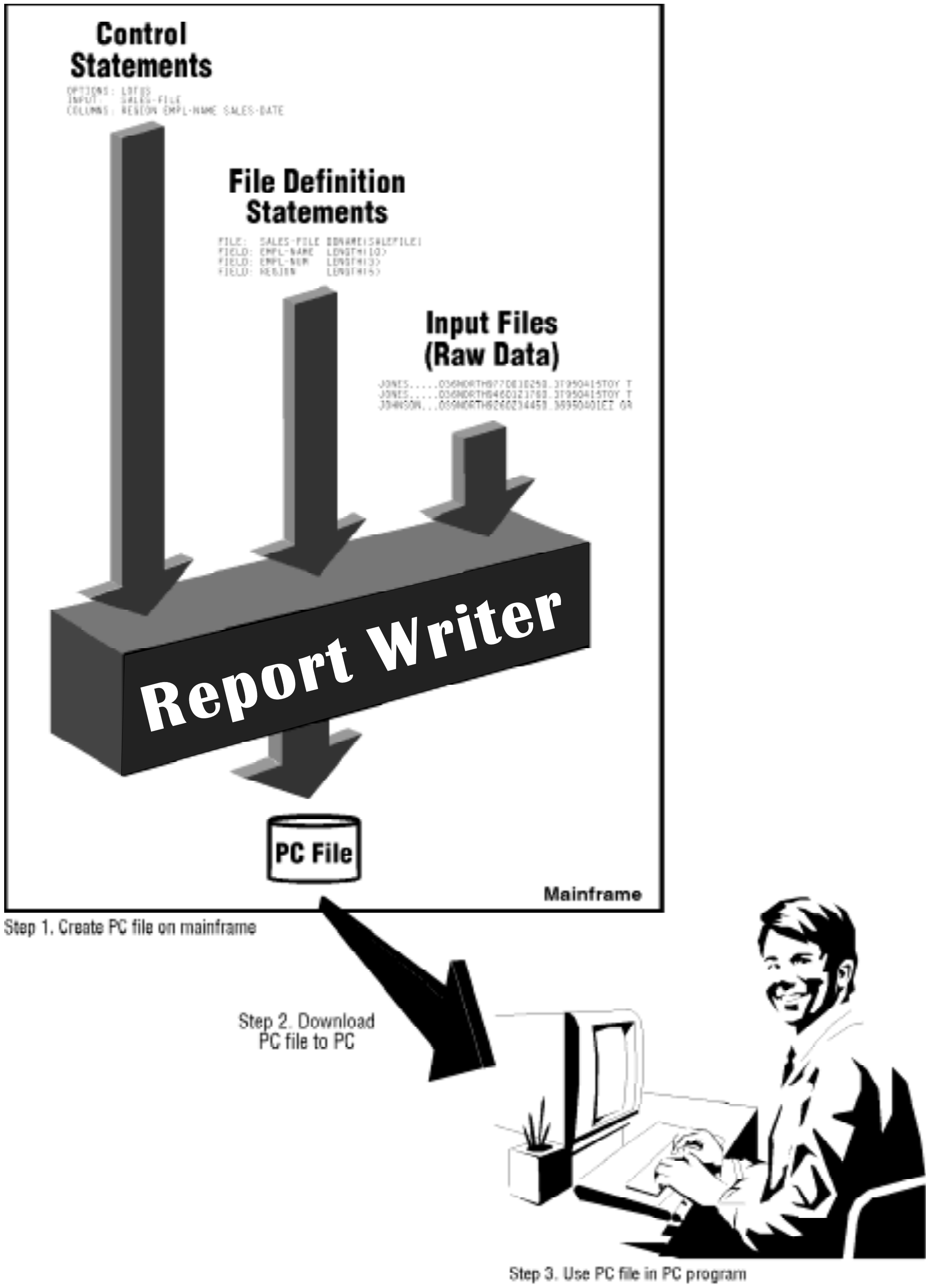
1. **Use Report Writer to create a custom PC file on your mainframe.**
   Report Writer's language is non–procedural, which means you just describe the *result* you want, not the programming steps needed to do it. Describe your PC file with a few simple "control statements". (These control statements 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 the control statements work.

   Once you've written the necessary control statements, submit a batch job to execute Report Writer. Report 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.) Report 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 into your PC program.**
   Start up your PC program and "open" or "import" the PC file with a few simple keystrokes. (Appendix H, "How to Import PC Files" describes the exact steps to use in many popular PC programs.) The PC program then reads the PC file and 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 PC is as easy as that with Report Writer!


Control Statements
File Definition Statements
Input Files (Raw Data)
Report Writer
PC File
Mainframe
Step 1. Create PC file on mainframe
Step 2. Download PC file to PC
Step 3. Use PC file in PC program

```
OPTIONS: LOTUS
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE
```

```
FILE:  SALES-FILE DDNAME(SALEFILE)
FIELD: EMPL-NAME  LENGTH(10)
FIELD: EMPL-NUM   LENGTH(3)
FIELD: REGION     LENGTH(5)
```

```
JONES.....036NORTH077DE10250.3T950415TOY T
JONES.....036NORTH046D121760.3T950415TOY T
JOHNSON...035NORTH026D234450.3S950401EI GR
```

Figure 4

The box on page 77 lists all of the Report Writer control statements used in 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 Report 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 almost every PC file you request.

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.

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 9, "Control Statement Syntax."

CONTROL STATEMENTS USED TO CREATE PC FILES (GROUPED BY FUNCTION)

Statements that Define Data

`FILE`	Defines a mainframe file
`FIELD`	Defines a field within a mainframe file
`ASM`	Lets you define a mainframe file using an Assembler record layout
`COBOL`	Lets you define a mainframe file using a Cobol record layout

Statements that Make Data Available to a PC File

`INPUT`	Specifies the primary input file
`READ`	Specifies an auxiliary input file
`COMPUTE`	Creates a new field

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 columns and column headings

Statements that Define the PC File Order, and Control Breaks

`SORT`	Specifies 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
`COPY`	Copies additional control statements for processing

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

This lesson teaches you how to produce a PC file using just three simple control statements. These statements are:

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

You only need three statements to create a PC file with Report Writer. For example:

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

The OPTION statement above tells Report Writer that you want to convert mainframe data into a PC file to use in Lotus 1–2–3.

The INPUT statement identifies the mainframe file containing the data you want to use in Lotus 1–2–3. 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 in an imaginary company.

The COLUMNS statement specifies which columns of data we want in our Lotus spreadsheet. Each field listed in this statement becomes one column in the spreadsheet. In this case we've requested columns for: 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.

With just these three statements, we've given Report Writer everything it needs to turn mainframe data into a PC file for Lotus 1–2–3!

Figure 20 illustrates this. The box on top shows the three control statements we just discussed. Based on these statements, Report Writer creates a PC file containing the requested data in Lotus import file format.

The PC screen shows the Lotus 1–2–3 spreadsheet obtained by importing the PC file. The spreadsheet contains the mainframe data we requested, properly laid out into rows and columns. There are even column headings for each column.

Once the mainframe data is in your PC spreadsheet, the possibilities of how to use it are limitless. As an example, **Figure 20** shows a simple Lotus graph created from the AMOUNT and TAX columns in the spreadsheet.

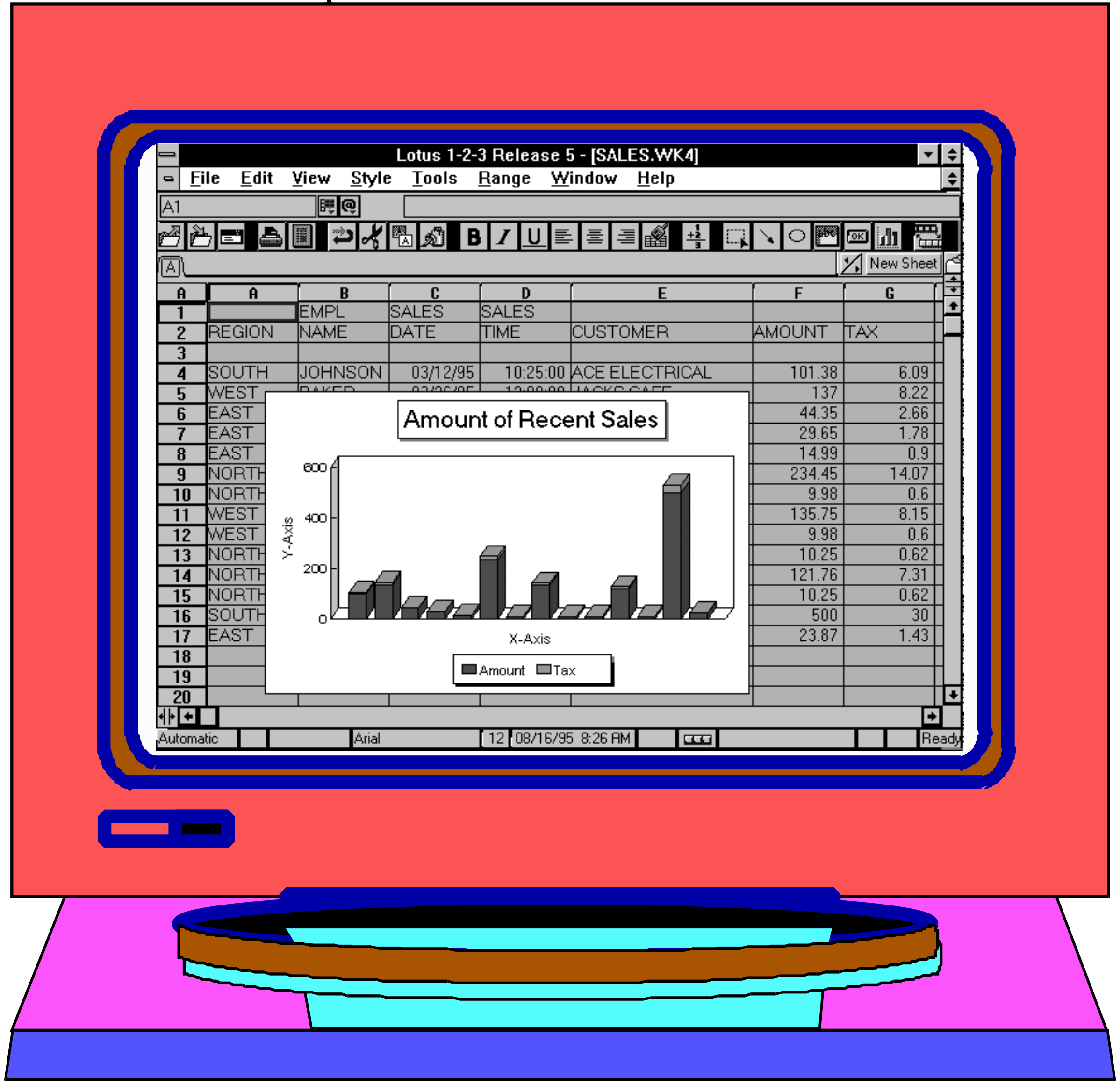
That's all there is to creating custom PC files with Report Writer. With just three simple statements we did what would otherwise have taken an entire COBOL program to do!

The following pages discuss these three control statements (and the importing process) in a little more detail.

These control statements:

```
OPTIONS: LOTUS
INPUT:   SALES–FILE
COLUMNS: REGION EMPL–NAME SALES–DATE SALES–TIME CUSTOMER AMOUNT TAX
```

Result in this Lotus 1–2–3 spreadsheet:



A	A	B	C	D	E	F	G
1		EMPL	SALES	SALES			
2	REGION	NAME	DATE	TIME	CUSTOMER	AMOUNT	TAX
3							
4	SOUTH	JOHNSON	03/12/95	10:25:00	ACE ELECTRICAL	101.38	6.09
5	WEST	BAKER	03/26/95	12:00:00	JACKS CAFE	137	8.22
6	EAST					44.35	2.66
7	EAST					29.65	1.78
8	EAST					14.99	0.9
9	NORTH					234.45	14.07
10	NORTH					9.98	0.6
11	WEST					135.75	8.15
12	WEST					9.98	0.6
13	NORTH					10.25	0.62
14	NORTH					121.76	7.31
15	NORTH					10.25	0.62
16	SOUTH					500	30
17	EAST					23.87	1.43
18							
19							
20							

Figure 20 A Lotus 1–2–3 spreadsheet obtained from just three control statements

Using the OPTIONS Statement to Name the PC Program

Different PC programs have slightly different formatting requirements for the "import files" they accept. Report Writer creates PC files for all of the major PC spreadsheet, database and graphics programs. Just use an OPTIONS statement to tell Report Writer which PC program you want the PC file for. In the example on page 79, we created a PC file to use in a Lotus 1–2–3 spreadsheet. We used this statement:

```
OPTIONS: LOTUS
```

All Report Writer control statements begin in column 1 with the name of the statement (for example, OPTIONS), followed immediately by a colon. What follows next will depend on the particular control statement involved. With an OPTIONS statement, you simply add a keyword that identifies the kind of PC file wanted. In the above example, we specified LOTUS, which is the keyword for a Lotus 1–2–3 file. Here are some other keywords you can use to request PC files for other PC programs.

KEYWORD	**PC PROGRAM**
ACCESS	Access
COREL	Corel Chart
CSV	"Generic" Comma–Separated–Values
DBASE3	dBASE III
DBASE4	dBASE IV
EXCEL	Excel
FOXPRO	FoxPro
HARVARD	Harvard Graphics
LOTUS	Lotus 1–2–3
MS–WORKS	Microsoft Works
PARADOX	Paradox
QUATTRO	Quattro Pro
RBASE	R:Base

Note: if the PC program you want is not listed above, see page 256. It explains how to create a PC file for other PC programs.

In other lessons in this chapter we will use some of the above PC options.

How to Use the INPUT and COLUMNS Statements

The INPUT and COLUMNS statements perform the same functions when creating PC files as they do when creating reports. The INPUT statement names the mainframe file to be used as input for the run. And the COLUMNS statement names the fields from that file that should be written to the PC file.

Importing Your PC File into Lotus 1–2–3

The three control statements discussed above result in Report Writer extracting data from the sales file on the mainframe and turning it into a PC file in Lotus format. Here are a few lines from the actual PC file created by Report Writer:

```
" ","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
...
```

But how did we get this PC file loaded into Lotus as a spreadsheet? After creating this PC file, we simply performed two additional steps to get the spreadsheet shown on page 79:

1) We downloaded the PC file to our PC. Just use whatever file transfer program your company normally uses. For example, if your company uses Attachmate's EXTRA! as its terminal emulator program, use EXTRA!'s file transfer facility.

2) We ran Lotus 1–2–3 and "opened" the downloaded PC file as a comma delimited file. For example, here's how you import a PC file into Lotus 1–2–3 for Windows (Release 5.) From an empty spreadsheet:

   - From the FILE menu, choose OPEN (this brings up an Open File dialog box)
   - Fill in the File Name
   - Choose TEXT for the File Type
   - Click the OK button

   Note: A similar process is used to import PC files into other versions of Lotus 1–2–3. The exact steps may vary a little from version to version. To be sure, just check your PC program's manual (or the online Help) for exact instructions on "importing" comma delimited ASCII files.

Note: the exact steps for importing PC files into various other PC programs are shown in Appendix H, "How to Import PC Files" (page 596.)

Note: the JCL used to create this PC file is shown on page 361 (MVS) and page 373 (VSE).

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, described in Appendix F, "Sample File Definitions" (page 588.) We will create a simple employee directory from that file. We 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: QUATTRO
INPUT:   EMPL–FILE
COLUMNS: EMPL–NUM  LAST–NAME  FIRST–NAME  SEX  SOCIAL–SEC–NUM
         HIRE–DATE  CITY  STATE
```

The OPTIONS statement above specifies that we want a PC file to use in Quattro Pro. The INPUT statement above specifies that the input file for our PC file will be the employee file (EMPL–FILE). The COLUMNS statement specifies the columns of data we want our spreadsheet 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 want. Just leave at least one blank space at the beginning of each continuation line.

The Quattro Pro spreadsheet obtained by using the above statements is shown in **Figure 21**.

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

Using Your Company's Files

You may be wondering how Report Writer knows the names of *your company's* files and fields. The answer is that your company's files are defined to Report Writer by other control statements that are kept in a Report Writer "copy library." For example, the statements used to define the SALES–FILE that we used earlier in this lesson are shown on page 587.

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 Report Writer Copy Library, in a format similar to that shown on page 587.

If you already know the name of the file to use, you can also get a list of all of its fields by adding the SHOWFLDS(YES) parm to your INPUT statement like this:

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

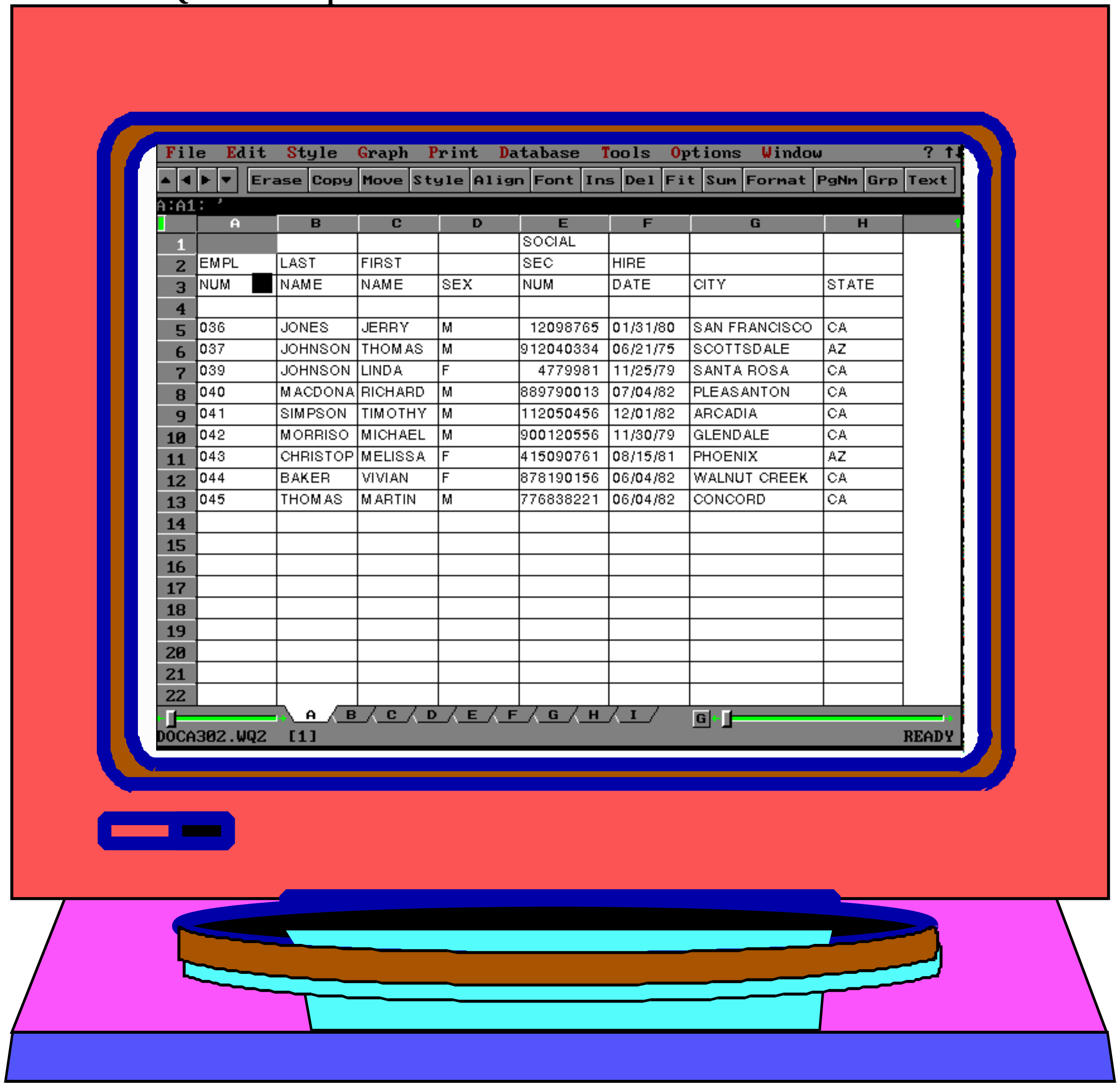
The above statement tells Report Writer to print (in the control statement listing) a list of all of the fields defined for SALES–FILE.

If a file that you need has not yet been defined, see Chapter 5, "How to Define Your Input Files" for information on doing that.

These control statements:

```
OPTIONS: QUATTRO
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:A1: '
```

	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								

```
DOCA302.WQ2   [1]                                   READY
```

Figure 21 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 tells Report Writer which PC program to format the PC file for
- the INPUT statement tells Report Writer which input file contains the data needed in your PC file
- the COLUMNS statement tells Report 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 8, "General Syntax Rules." In that chapter you will learn such things as:

- **how long** each line can be (page 384)
- how to **continue** control statements onto multiple lines (page 385)

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 Chapter 4, "Beyond the Basics." Examples of additional features are:

- how to specify your own **column headings** for a PC file (page 127)
- how to **suppress column headings** in your PC file (page 127)
- how to reserve **more room for numeric columns** in your PC file (page 131)
- how to create a column that contains a **literal text** (page 124)
- how to produce **multiple rows in the PC file** for each input record (page 147)
- how to turn data from **DB2 tables and views** into PC files (page 340)
- how to turn data from **existing reports** into PC files (page 241)

The complete syntax for the OPTIONS, INPUT and COLUMNS statements is given in Chapter 9, "Control Statement Syntax."

(This page left blank intentionally.)

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, Report Writer includes every record from the mainframe file. Use the INCLUDEIF statement to tell Report 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 that might take hours to download (and which might use up half of your hard disk in the process.) 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 on page 79. But this time let's just download the data for the employee named Jones. We simply add the following INCLUDEIF statement to the other control statements:

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

The above INCLUDEIF statement tells Report Writer to "*include*"records from the SALES–FILE "*if*" the EMPL–NAME field is equal to 'JONES'. Report 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 22 shows an Excel spreadsheet produced using the above statement. Only the sales made by Jones appear in that spreadsheet.

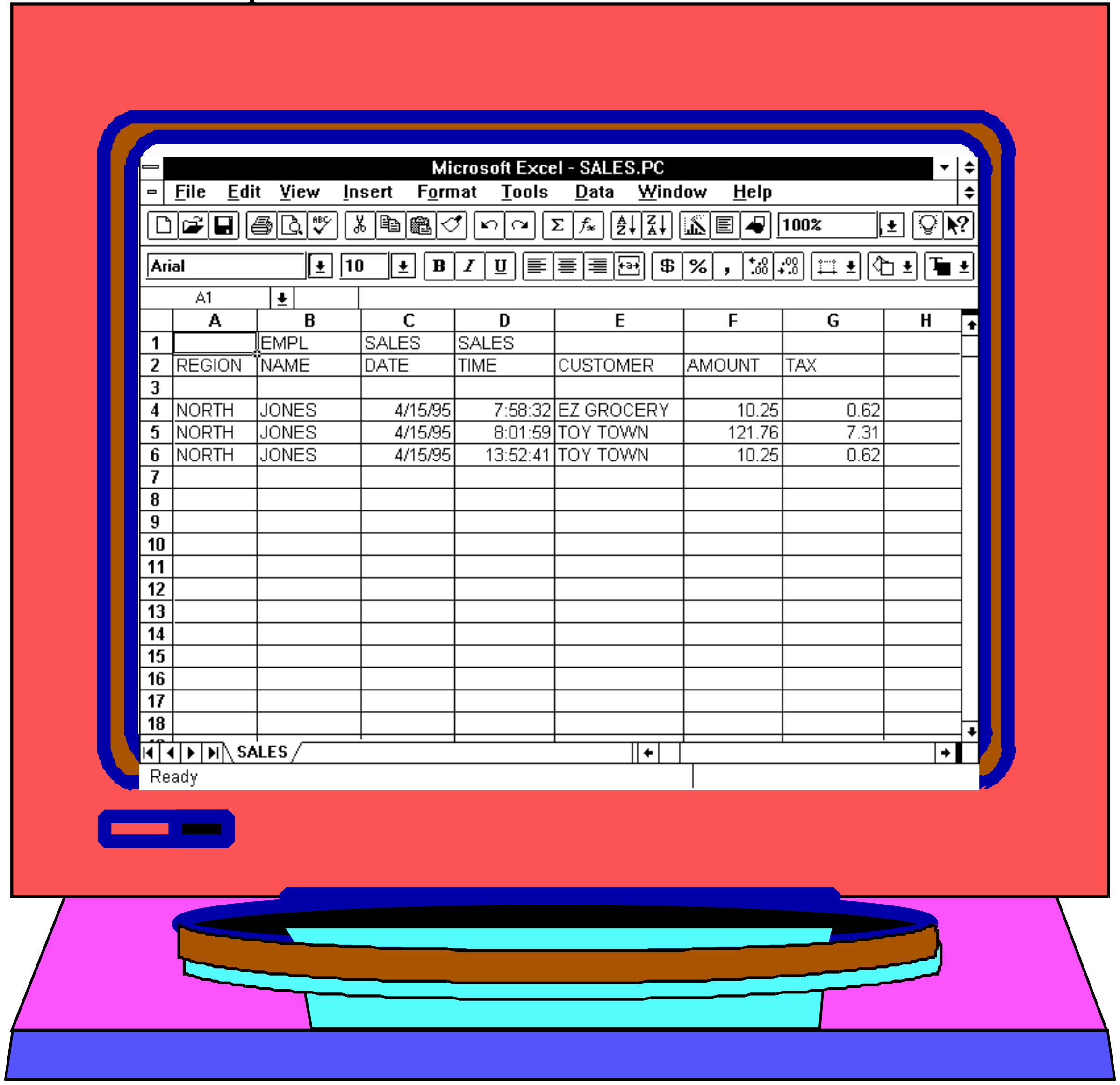
The INCLUDEIF statement may appear anywhere after the INPUT statement. Only one INCLUDEIF statement is allowed per run, but it may contain as many conditions as you like.

By the way, the INCLUDEIF statement can refer to *any* of the fields in the input file. You are *not limited* to just those fields that are listed in the COLUMNS statement.

These control statements:

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

Result in this Excel spreadsheet:


Microsoft Excel - SALES.PC
File Edit View Insert Format Tools Data Window Help
100%
Arial
10
A1
SALES
Ready

	A	B	C	D	E	F	G	H
1		EMPL	SALES	SALES				
2	REGION	NAME	DATE	TIME	CUSTOMER	AMOUNT	TAX	
3								
4	NORTH	JONES	4/15/95	7:58:32	EZ GROCERY	10.25	0.62	
5	NORTH	JONES	4/15/95	8:01:59	TOY TOWN	121.76	7.31	
6	NORTH	JONES	4/15/95	13:52:41	TOY TOWN	10.25	0.62	

Figure 22 Using an INCLUDEIF statement to specify which records to include in a PC file

How to Write Conditional Expressions

The INCLUDEIF statement consists of a **conditional expression**. The complete rules for writing conditional expressions are explained beginning on page 399. 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 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* of the conditions is true. (Of course, records will also be included if *both* 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 literal, are *not* enclosed in quotation marks. Numeric literal also must *not* contain commas. (The rules for writing literal are thoroughly explained beginning on page 389).

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 Report 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.

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.)

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.

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 Chapter 9, "Control Statement Syntax," (page 481.) Examples of additional features include:

- how to use **symbols** rather than the actual words AND and OR in your conditional expressions
- how to **scan** a character field, to see if a certain text exists *anywhere* within the field
- how to specify conditions based on **bit fields**
- how to specify a condition based on a field's raw **hexadecimal** value
- how to specify date literal in DD/MM/YY or DD/MM/YYYY format (page 137), like this:
  ```
      INCLUDEIF: SALES–DATE > 15/4/1995
  ```
- you may also be able to use the **KEYRANGE parm** of the INPUT statement to limit the records included in your run (page 485)

(This page left blank intentionally.)

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 beginning on page 410. Generally, your expression will consist of one or more mathematical operations performed on numeric fields or numeric literal.

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 Paradox table in **Figure 23** was obtained by using the above COMPUTE statement.

> **Note:** COMPUTE statements normally appear *after* the INPUT statement, but must appear *before* any control statement that refers to the field being created. In the example on page 93, the COMPUTE statement for TOTAL–AMOUNT had to come before the COLUMNS statement, since the COLUMNS statement referred to that field.

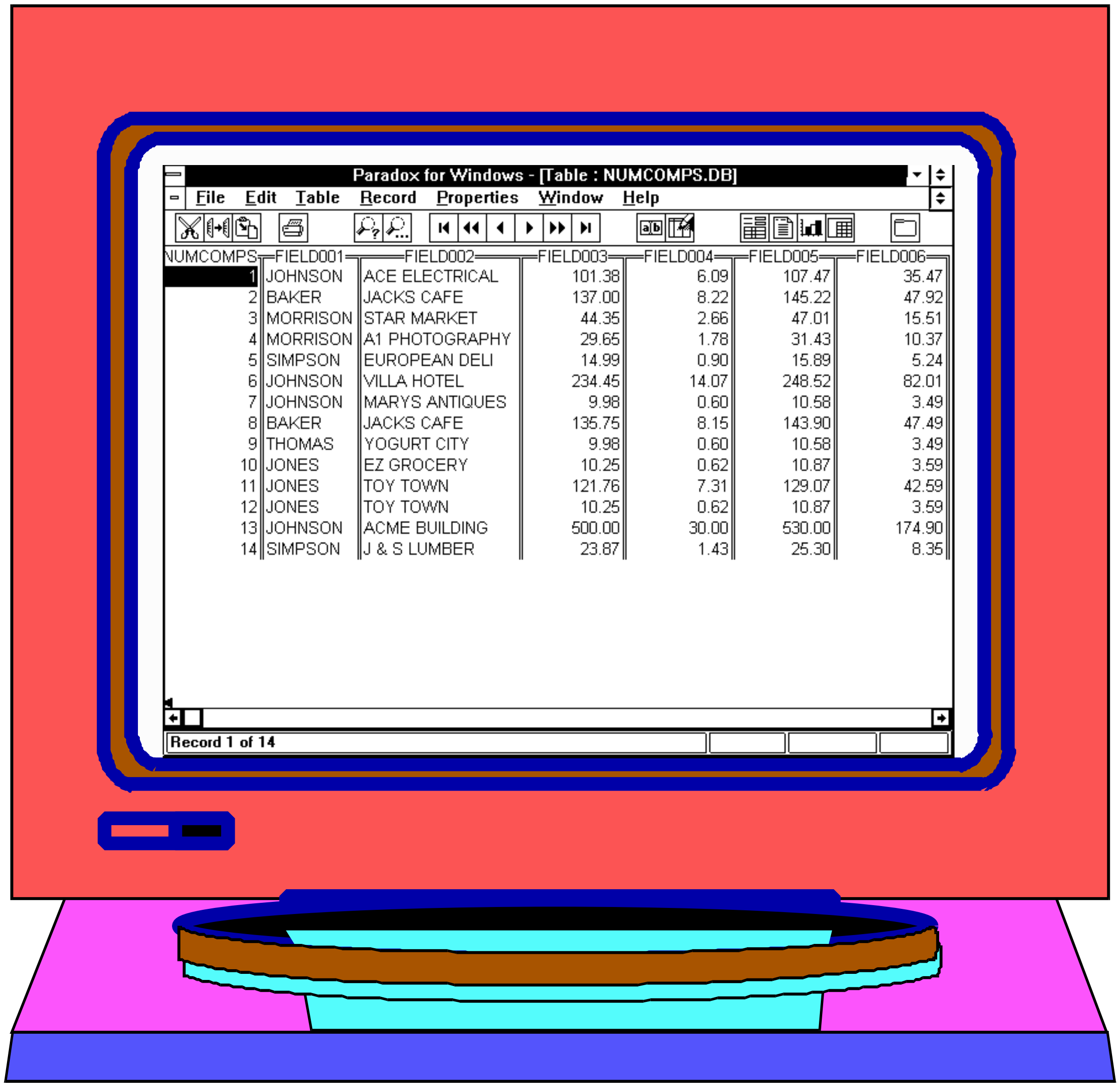
You can perform addition, subtraction, multiplication, and division in the COMPUTE statement. Use the +, –, * and / symbols, respectively. You may also use parentheses as needed to indicate the order in which the operations should be performed.

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

These control statements:

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

Result in this Paradox table:


Paradox for Windows - [Table : NUMCOMPS.DB]
File Edit Table Record Properties Window Help
Record 1 of 14

NUMCOMPS	FIELD001	FIELD002	FIELD003	FIELD004	FIELD005	FIELD006
1	JOHNSON	ACE ELECTRICAL	101.38	6.09	107.47	35.47
2	BAKER	JACKS CAFE	137.00	8.22	145.22	47.92
3	MORRISON	STAR MARKET	44.35	2.66	47.01	15.51
4	MORRISON	A1 PHOTOGRAPHY	29.65	1.78	31.43	10.37
5	SIMPSON	EUROPEAN DELI	14.99	0.90	15.89	5.24
6	JOHNSON	VILLA HOTEL	234.45	14.07	248.52	82.01
7	JOHNSON	MARYS ANTIQUES	9.98	0.60	10.58	3.49
8	BAKER	JACKS CAFE	135.75	8.15	143.90	47.49
9	THOMAS	YOGURT CITY	9.98	0.60	10.58	3.49
10	JONES	EZ GROCERY	10.25	0.62	10.87	3.59
11	JONES	TOY TOWN	121.76	7.31	129.07	42.59
12	JONES	TOY TOWN	10.25	0.62	10.87	3.59
13	JOHNSON	ACME BUILDING	500.00	30.00	530.00	174.90
14	SIMPSON	J & S LUMBER	23.87	1.43	25.30	8.35

Figure 23 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 = 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 Paradox table in **Figure 23** (page 93) also uses the above statement.

In addition to the basic arithmetic operations, there are also a 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 566.)

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:

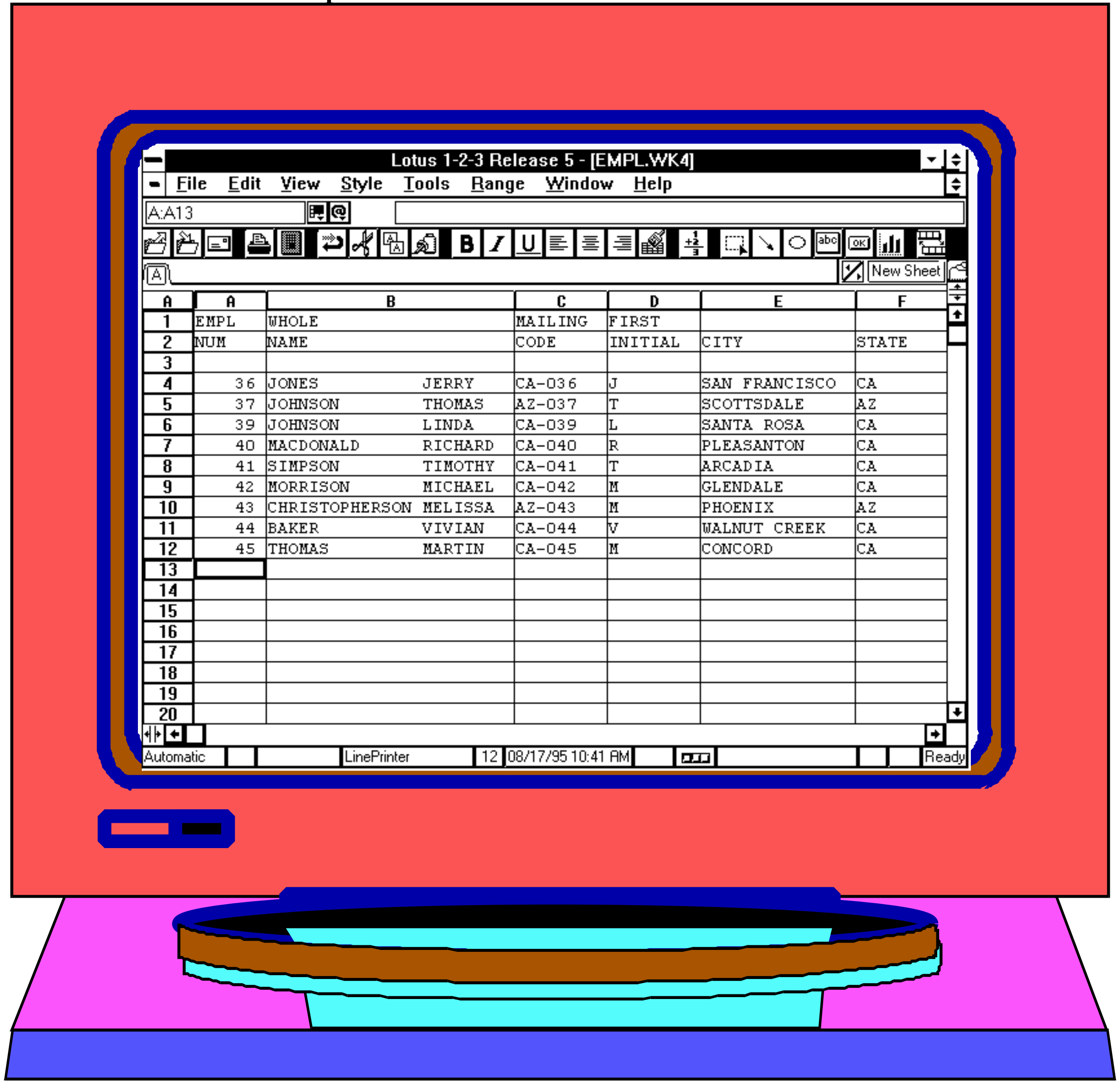
```
COMPUTE: FIRST–INITIAL = #LEFT(FIRST–NAME,1)
```

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.

These control statements:

```
OPTIONS: LOTUS
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:


Lotus 1-2-3 Release 5 - [EMPL.WK4]
File Edit View Style Tools Range Window Help
A:A13
New Sheet
Automatic
LinePrinter
12
08/17/95 10:41 AM
Ready

A	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 24 Using the COMPUTE statement to create character fields for a PC file

There are a number of other built–in functions which can also be used. A complete list of built–in functions is found in Appendix D, "Built-In Functions" (page 566.)

Figure 24 (page 95) shows a spreadsheet obtained by using the COMPUTE statements shown above.

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, Report 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.

You are allowed to 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:

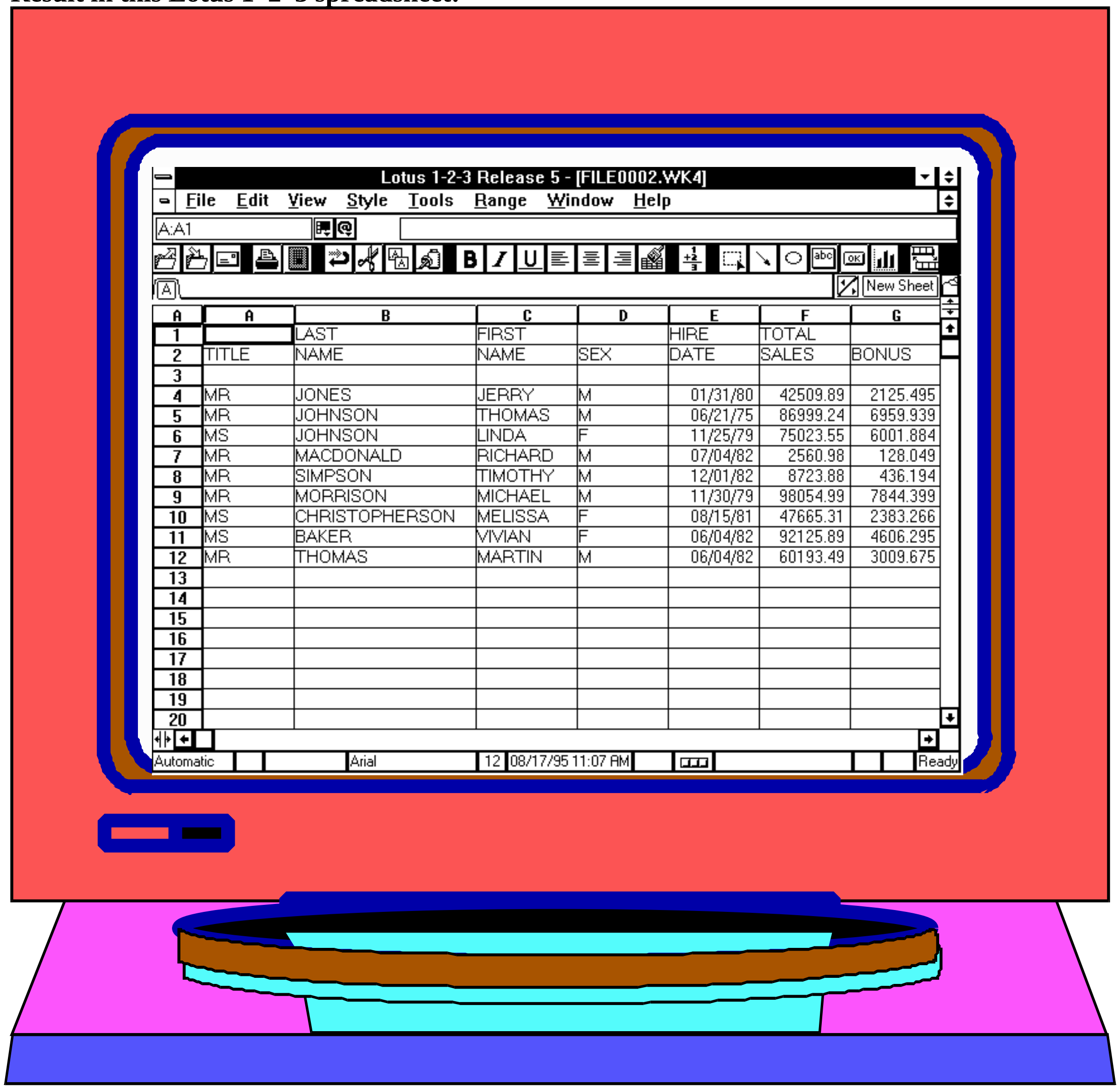
```
COMPUTE: TITLE = WHEN(SEX = 'M')  ASSIGN('MR')
                 ELSE             ASSIGN('MS')
```

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.

These control statements:

```
OPTIONS:  LOTUS
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:


Lotus 1-2-3 Release 5 - [FILE0002.WK4]
File Edit View Style Tools Range Window Help
A:A1
New Sheet
Automatic
Arial
12
08/17/95 11:07 AM
Ready

A	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 25 Assigning values to computed fields based on conditions

Figure 25 (page 97) 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* will be 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 Lesson 2 on page 88.) The complete rules for writing conditional expressions are given beginning on page 399. Additional examples of COMPUTE statements are shown beginning on page 451.

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 the 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 under the COMPUTE statement in Chapter 9, "Control Statement Syntax" (page 444). Other additional features are discussed in Chapter 4, "Beyond the Basics." Examples of the additional topics include:

- how to create **date** type fields (page 452)
- how to create **time** type fields (page 254)
- how to create **bit** type fields (page 452)
- how to specify how many **decimal places** a numeric or time field should contain (page 450)
- how to specify **column headings** for the fields you create (page 449)
- how to specify whether a numeric or time field should be **totalled** at control breaks (page 144)
- how to **retain the value** of a COMPUTE field in certain cases (page 238)

The complete syntax for the COMPUTE statement is given in Chapter 9, "Control Statement Syntax" (page 444).

(This page left blank intentionally.)

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, Report 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 all appeared in sales date order. (For example, see the spreadsheet on page 79.) The EMPL–FILE sample file is a VSAM file stored in EMPL–NUM order. Therefore, the earlier spreadsheet from that file was in employee number order (page 83.)

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. Report 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 26** shows a Microsoft Works spreadsheet obtained by using the above statement.

By default, Report 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

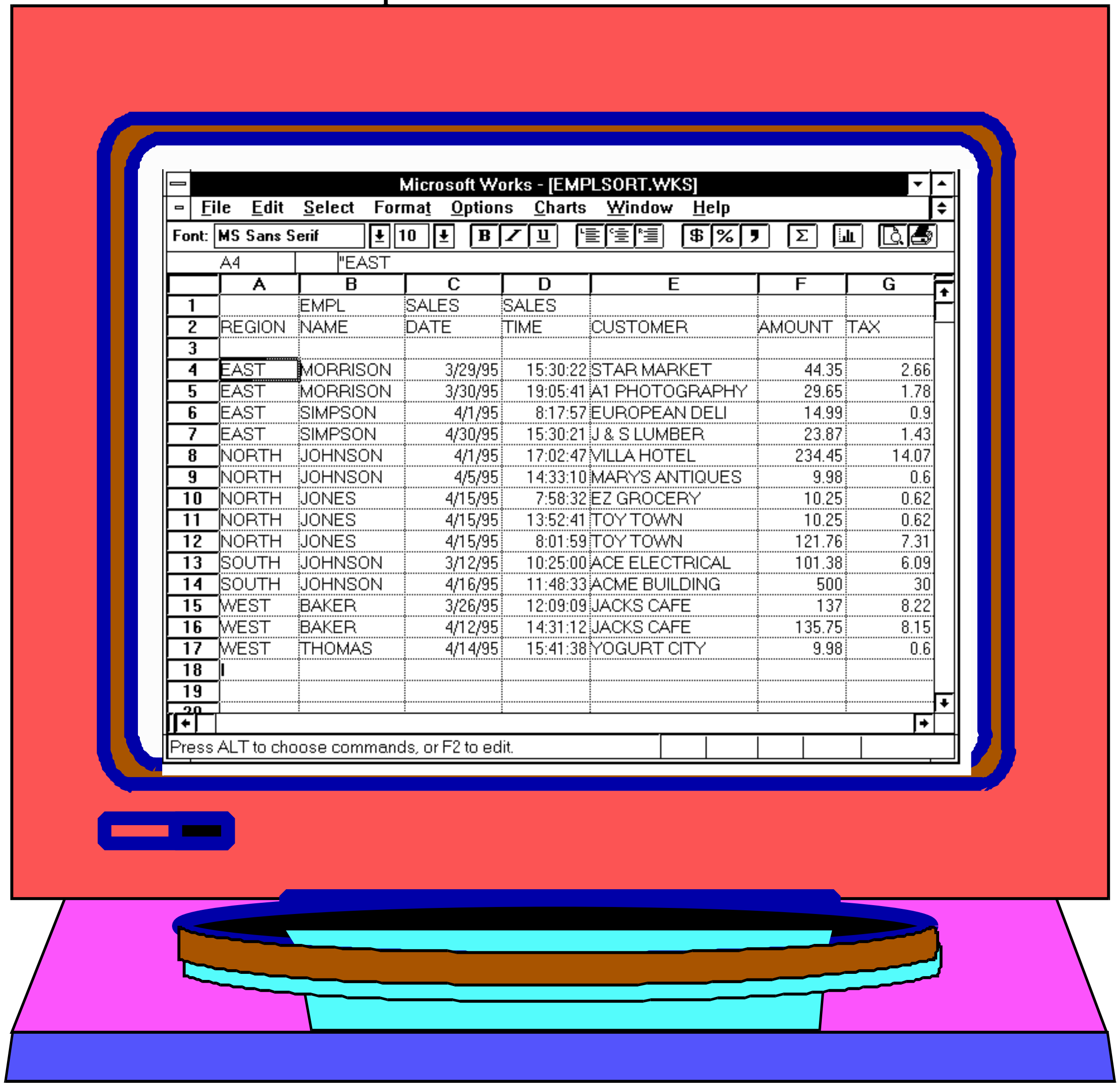
If you prefer, you can let Report Writer *automatically* sort your PC file for you. To have your PC file automatically sorted on its first 5 columns of data, simply specify the AUTOSORT option, like this:

```
OPTIONS: AUTOSORT
```

These control statements:

```
OPTIONS: MS–WORKS
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:


Microsoft Works - [EMPLSORT.WKS]
File Edit Select Format Options Charts Window Help
Font: MS Sans Serif 10
A4 "EAST

	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							

Press ALT to choose commands, or F2 to edit.

Figure 26 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." Examples of additional features include:

- creating a **control break** from the SORT statement (page 182)
- requesting **totals and statistics** in the SORT statement (page 194)

The complete syntax for the SORT statements is given in Chapter 9, "Control Statement Syntax" (page 524).

(This page left blank intentionally.)

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 101 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:

```
BREAK: REGION
```

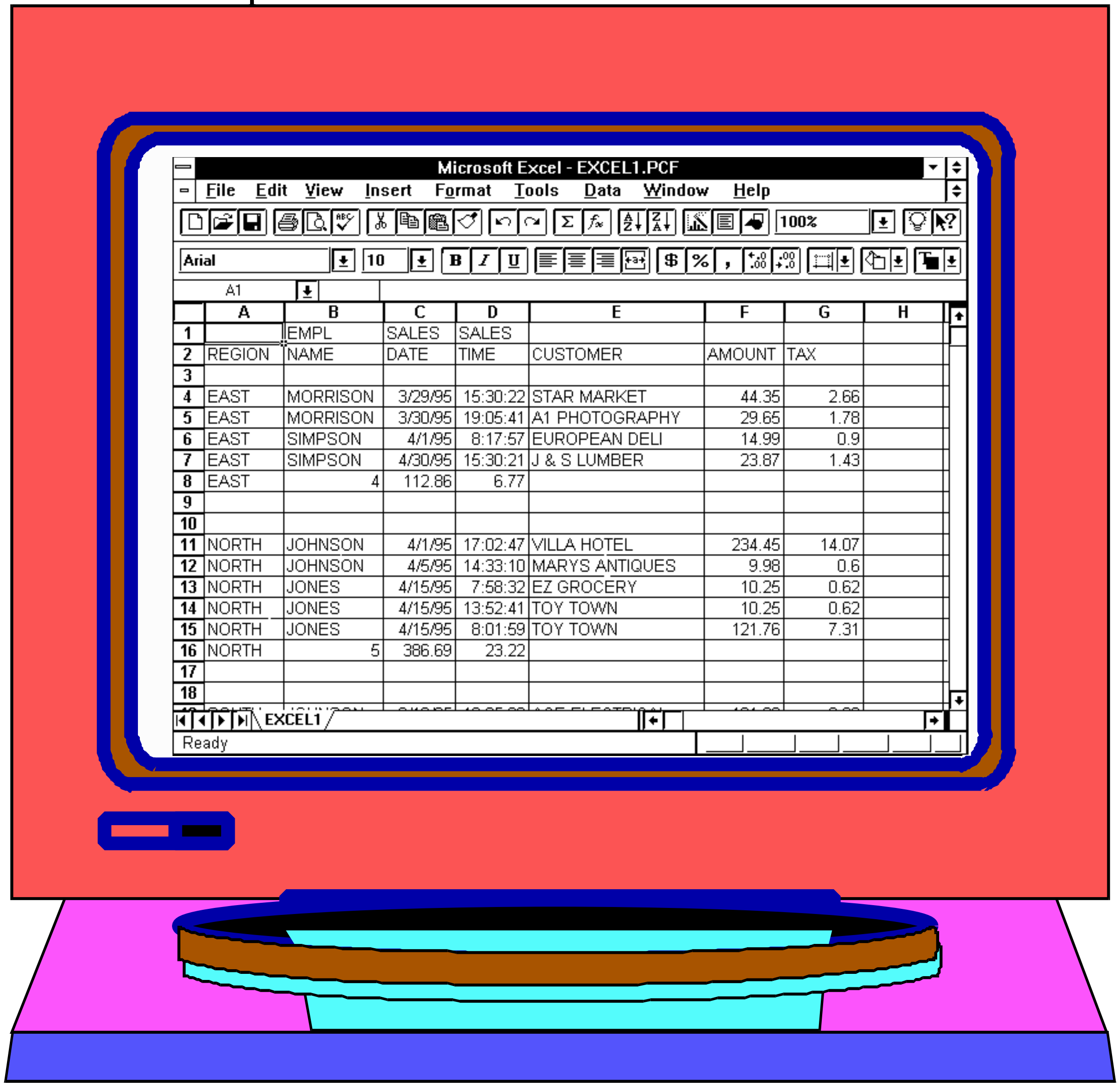
The above statement makes REGION a control break field. Now we will get REGION totals in the resulting spreadsheet whenever one region ends and another region is about to begin.

Figure 27 shows a spreadsheet obtained by using the BREAK statement above to produce a control break.

These control statements:

```
OPTIONS: EXCEL
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:


Microsoft Excel - EXCEL1.PCF
File Edit View Insert Format Tools Data Window Help
100%
Arial
10
A1
EXCEL1
Ready

	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 27 Using the BREAK statement to create a control break with subtotals in a PC file

Customizing the Control Break

The Excel spreadsheet in the previous example (page 105) shows what Report 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 were 4 items in the control group for the East region.) Following this are 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 your 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 Report 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 Report Writer which columns are wanted in the *detail rows*. The FOOTING parm tell Report 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, Report Writer puts two blank rows after the total row at a control break (see page 105). Use the SPACE parm in your BREAK statement to request a different number of blank lines. For example:

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

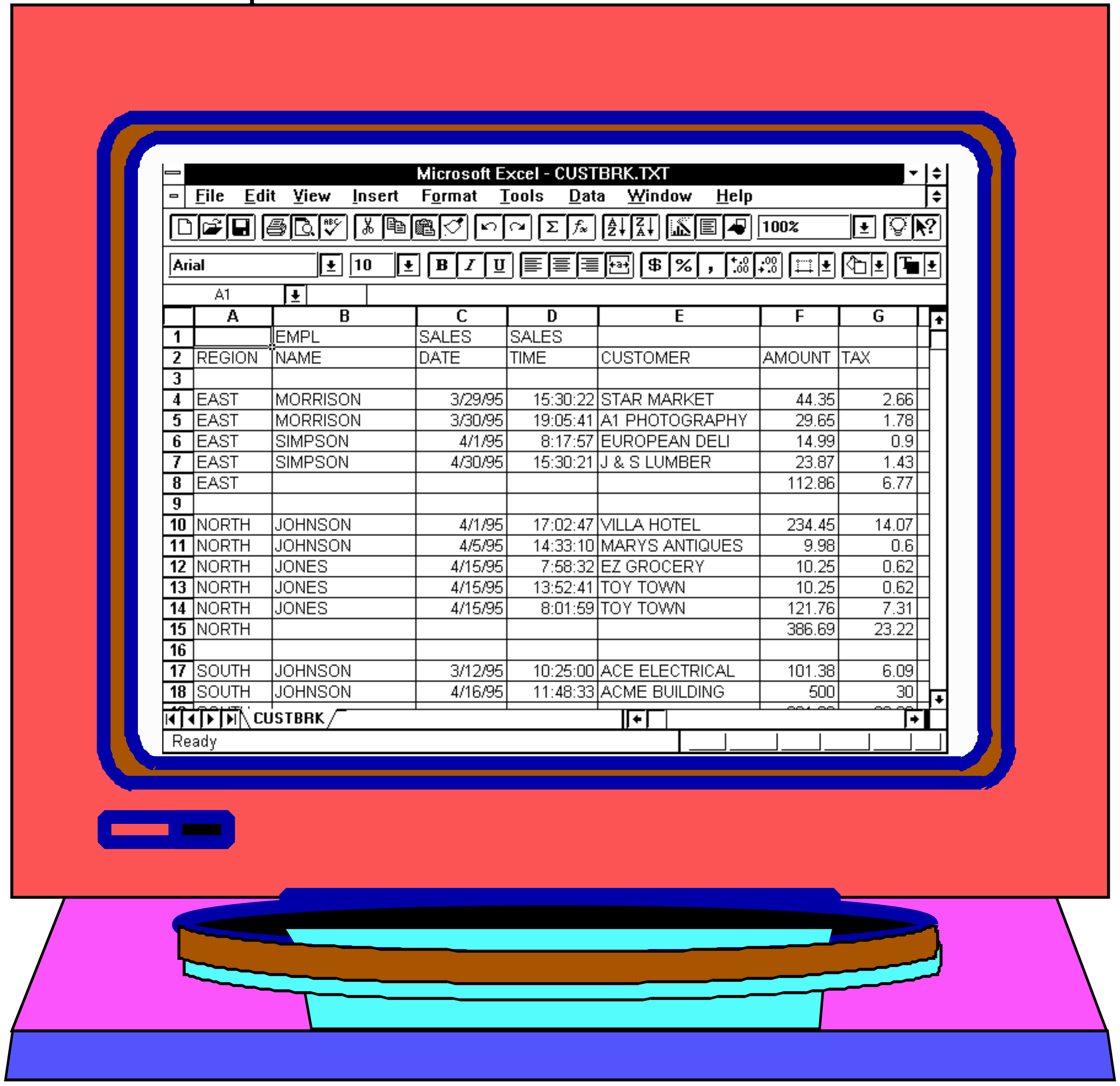
The above statement requests just one blank row at the REGION control break. You may also specify SPACE(0) if you want *no* blank rows in your spreadsheet.

Figure 28 uses the FOOTING, NOTOTALS and SPACE parms to customize the control break.

These control statements:

```
OPTIONS: EXCEL
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:



	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

Figure 28 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." Examples of additional topics include:

- how to write one or more **customized rows at the beginning** of a control break (page 208)
- how to write one or more **customized rows at the end** of a control break (page 196)
- how to **customize the total row**, and the other statistical rows (page 190 and 194)
- how to **suppress the total row** at a control break (page 193)
- how to show various **statistics** at control breaks (page 202)
- how to compute **percentages and ratios** that apply to an entire control group (page 187)
- how to have **multiple levels** of control breaks (page 211)

The complete syntax for the BREAK statement is given in Chapter 9, "Control Statement Syntax" (page 421).

(This page left blank intentionally.)

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. It's a waste of time to download the entire mainframe file and then use your PC program to summarize the data. Instead, let Report Writer perform the summarization for you on the mainframe. Then just download the small summary file to your PC.

Summarizing a mainframe file with Report Writer is very easy.

For example, consider the Excel spreadsheet we created back on page 105. 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.

For this example, 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
```

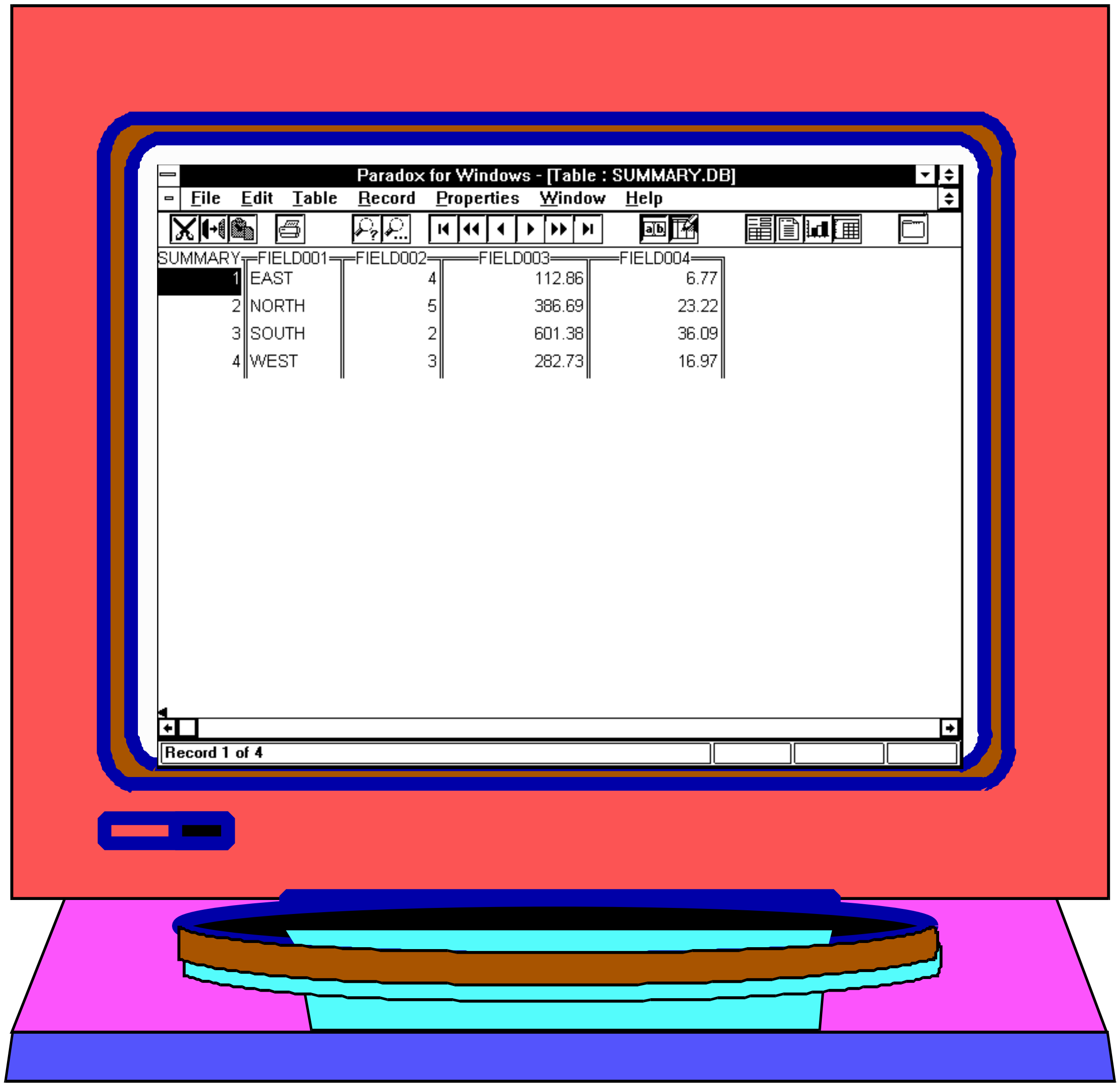
Figure 29 shows a Paradox table obtained by using the above statement. As you can see, the table has just four rows of actual data — one for each region in the mainframe file. The first column in each row contains a region name. The second column shows the number of records that were summarized in order 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 Report 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: PARADOX  SUMMARY
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



Paradox for Windows - [Table : SUMMARY.DB]
File Edit Table Record Properties Window Help
Record 1 of 4

SUMMARY	FIELD001	FIELD002	FIELD003	FIELD004
1	EAST	4	112.86	6.77
2	NORTH	5	386.69	23.22
3	SOUTH	2	601.38	36.09
4	WEST	3	282.73	16.97

Figure 29 A spreadsheet 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 file **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 190)
- **obtaining statistics** (such as averages, maximums and minimums) in your summary file (page 194)
- creating **multiple levels** of summarization (page 211)
- including a **limited number of detail records** in each control group, creating spreadsheets such as "The Top 3 Sales in Each Region" (page 220)

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 Report Writer's most powerful features is its ability to use *any number* of input 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 use 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 Report Writer processes the primary and auxiliary input files. Report 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, Report Writer also reads one record from each of those files. (These files are always read *randomly*, using a key.) At this point, Report 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 other fields can be used

After processing this set of records, Report 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.

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

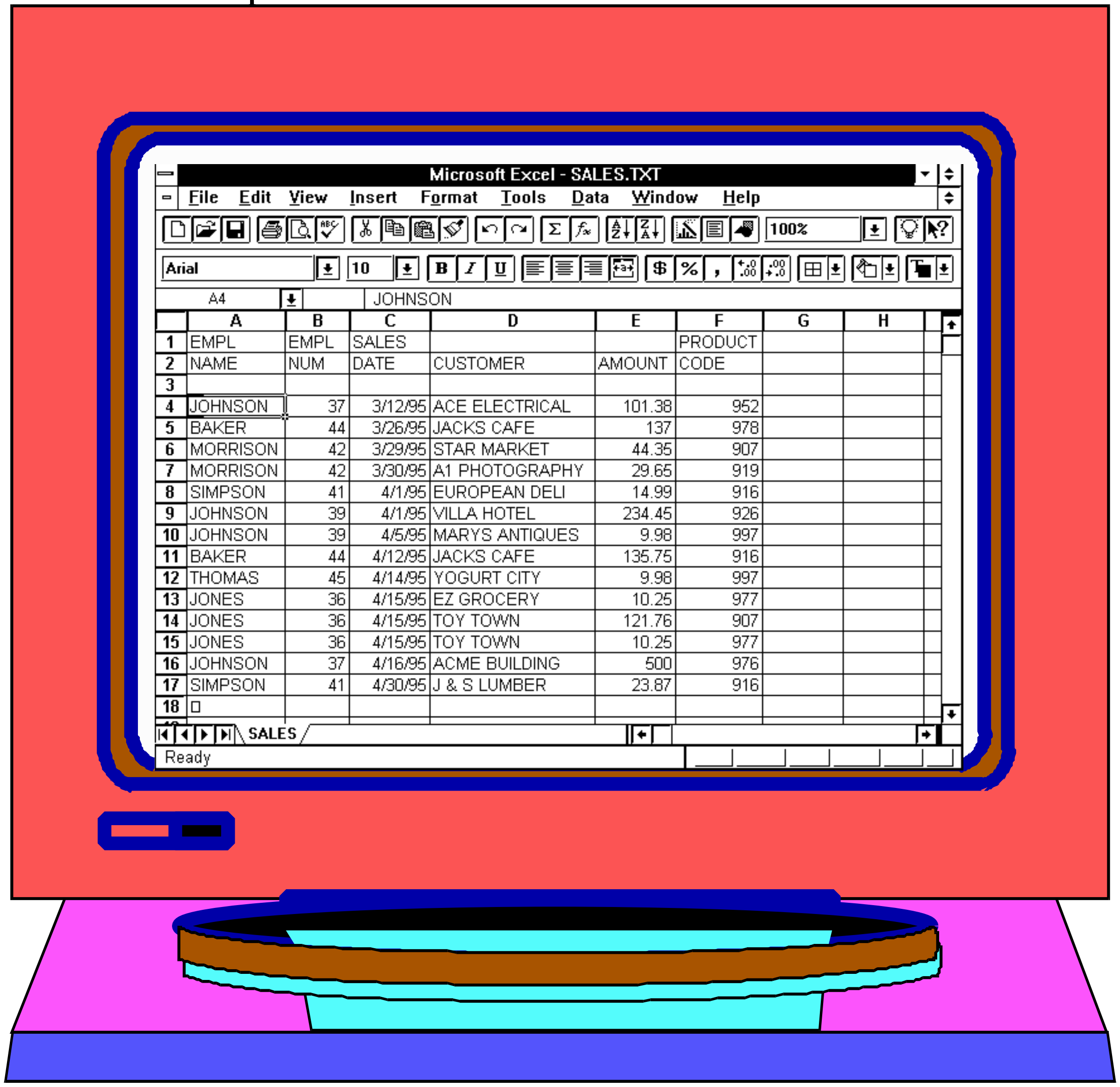
How to Use the READ Statement

Now let's look at a concrete example of how to use the READ statement. Begin by considering **Figure 30**, 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 which product was sold to the customer.

These control statements:

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

Result in this Excel spreadsheet:


Microsoft Excel - SALES.TXT
File Edit View Insert Format Tools Data Window Help
100%
Arial
10
A4
SALES
Ready

A4 JOHNSON

	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 30 A spreadsheet that uses only the primary input file

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 83.) 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 in reading the EMPL–FILE. We can make the EMPL–FILE an auxiliary input file, then, by simply adding this statement:

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

This READ statement tells Report Writer to use the EMPL–NUM field from each record in the SALES–FILE as a key for reading 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
```

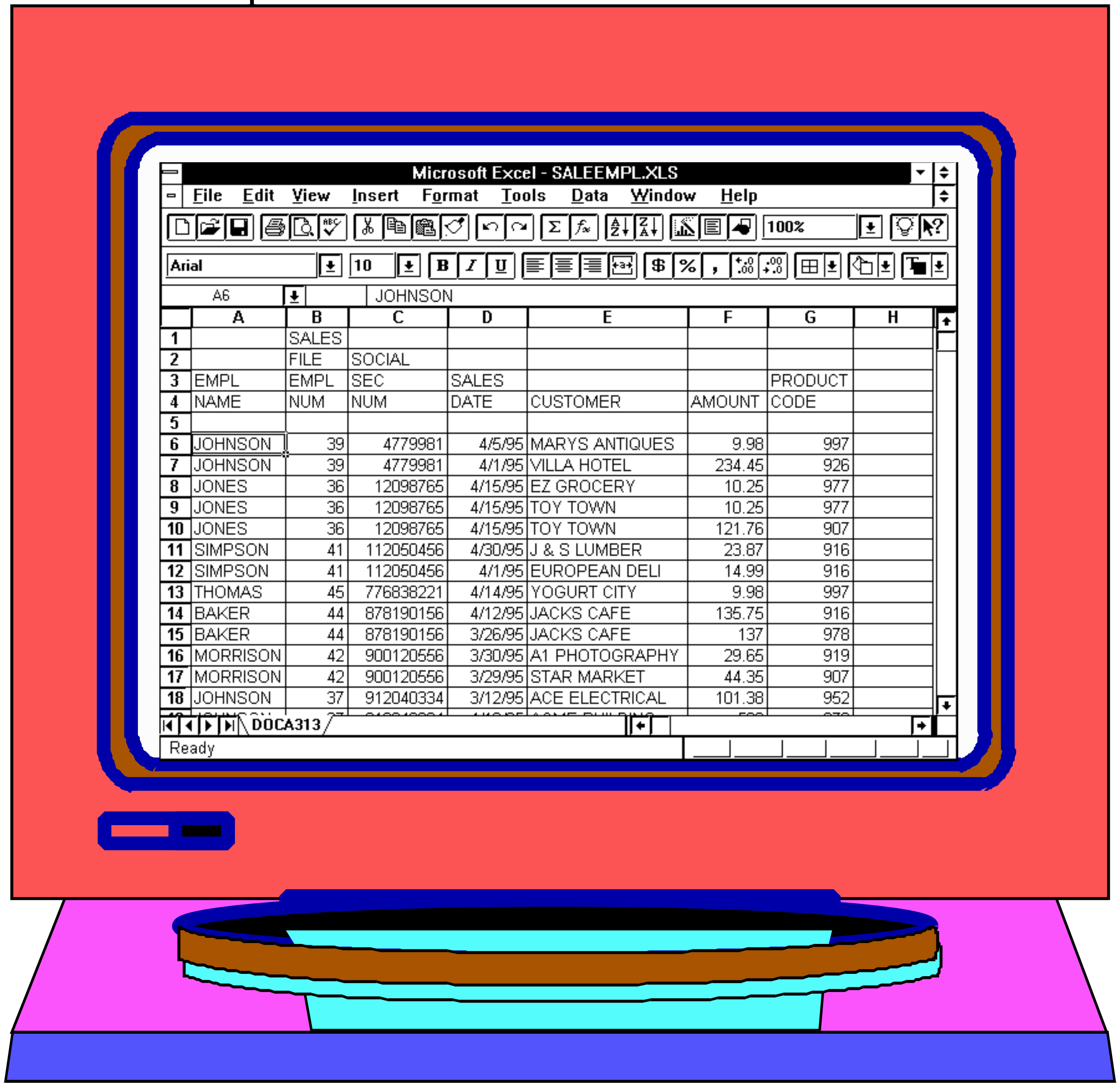
Notice that in the above COLUMNS statement we must now prefix the EMPL–NUM field with a record name (like this: SALES–FILE.EMPL–NUM). This is because after the READ statement, EMPL–NUM is no longer a *unique* field name. A field by that name exists in both the SALES–FILE and the EMPL–FILE. (See Appendix F, "Sample File Definitions.") 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 page 232.)

Figure 31 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 use fields from the primary input file.

These control statements:

```
OPTIONS: EXCEL
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
100%
Arial
10
A6
JOHNSON
DOCA313
Ready

	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	

Figure 31 A spreadsheet that uses a READ statement to specify an auxiliary input file

How to Use Multiple READ Statements

You are allowed to use an *unlimited* number of READ statements in requesting a PC file. For example, the Excel spreadsheet in **Figure 32** 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 an another auxiliary input file. (The PRODUCT–FILE is described in Appendix F, "Sample File Definitions.")

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)
```

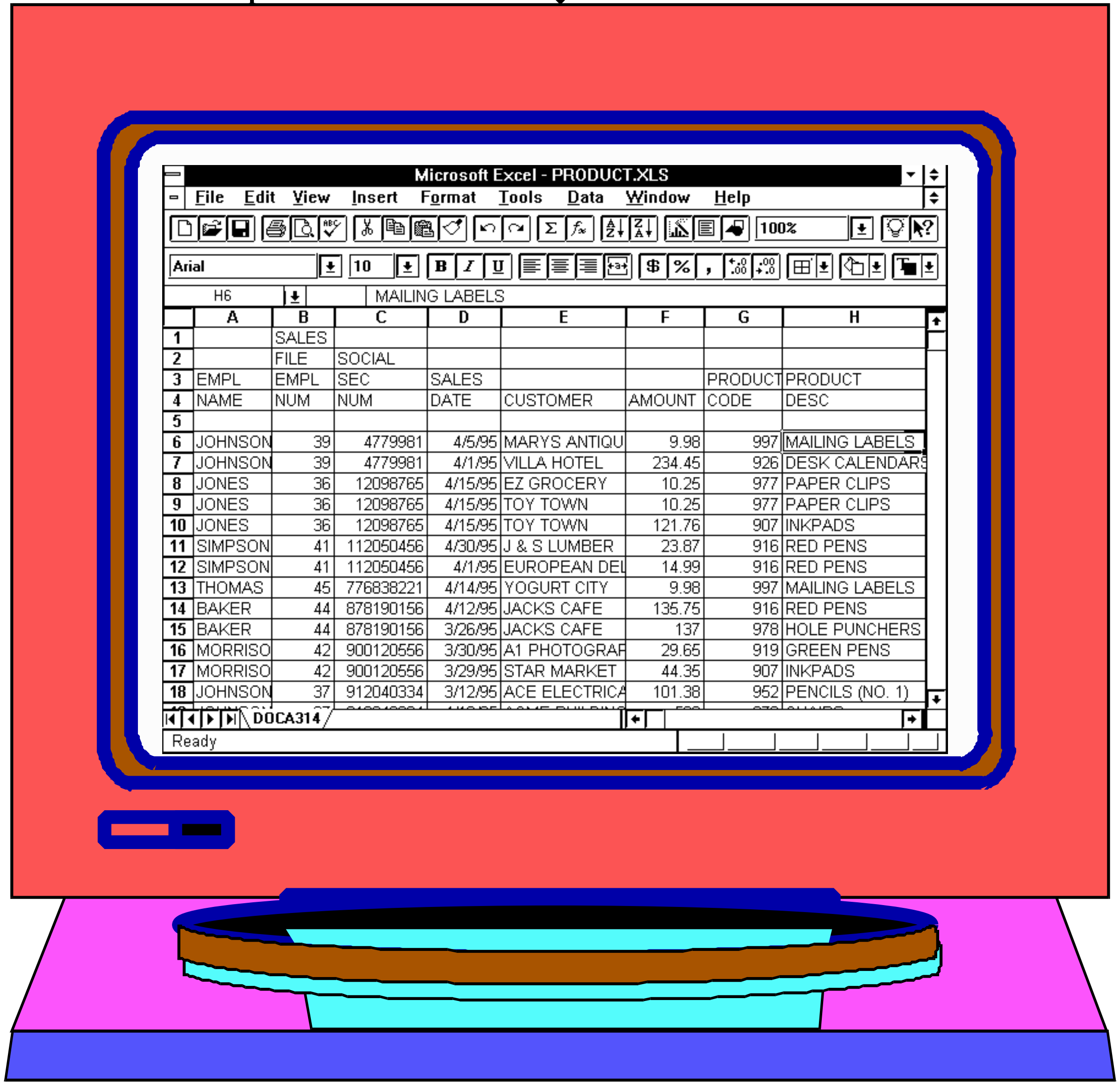
By having two READ statements in addition to the INPUT statement, the PC file now has *three input files*. Data from *all* of these files can be used in any of the subsequent control statements. In the Excel spreadsheet in **Figure 32**, the COLUMNS statement uses two fields from the auxiliary input files. It uses the SOCIAL–SEC–NUM field from the EMPL–FILE, and the PRODUCT–DESC field from the PRODUCT–FILE:

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

These control statements:

```
OPTIONS: EXCEL
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
100%
Arial
10
H6
MAILING LABELS
DOCA314
Ready

	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)

Figure 32 A spreadsheet 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**
- 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." Examples of additional features include:

- how to assign a **record name** to the records read from auxiliary input files (page 232)
- how to read more than one record from the **same auxiliary input file** (page 228)
- how to use **data from one auxiliary input file as the read key** to another auxiliary input file (page 230)
- what happens when **no record is found** for a particular read key (page 233)
- how to determine whether the read for a particular key was **successful** or not (page 233)
- how to use the READ statement to obtain data from a **DB2 table or view** (page 340)

The complete syntax for the READ statement is given in Chapter 9, "Control Statement Syntax" (page 510).

Chapter 4. Beyond the Basics

Chapter Table of Contents

Chapter Table of Contents (Continued)

Chapter 4. Beyond the Basics

This chapter is a user's guide to some of Report Writer's additional features. Many of the control statements introduced earlier in Chapters 2 and 3 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 124)
- how to change the **spacing** between report columns (page 124)
- how to change the **column headings** (page 127)
- how to change the **width** of a column (page 131)
- how to change the way **dates, times and numbers** are formatted (page 132)
- how to format dates, times and numbers for **international users** (page 137)

- how to change the **justification** of data within a column (page 142)
- how to change which columns are **totalled** (page 144)
- how to produce **multi–line reports or multi–row PC files** (page 147)
- how to print **bar graphs** in a report (page 150)
- how to put a **text (such as a vertical line) between report columns** (page 152)
- how to change the report **margins** (page 150)

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 33** on page 125.

The contents of the COLUMNS statement is simply a **print expression**. Print expressions are used in a number of different control statements. They tell Report 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 output records.)

As with other print expressions in Report 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**.

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 an input file** in 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 Report 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" for a complete list of built–in fields)

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 2 blanks to appear between the LAST–NAME and the FIRST–NAME columns, and causes 5 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, use the COLSPACE parm of the OPTIONS statement (page 498.)

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

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)
```

Notes:

- 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 33 Using spacing factors and literal texts in the COLUMNS statement

first parenthesis.) You may use any combination of parms, in any order. Separate the parms with a comma and/or with one or more 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 what parms are available in the COLUMNS statement. Subsequent sections of this chapter explain in detail how to use each of these parms.

COLUMN 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 144. The following example specifies that the TOTAL–SALES column should not be accumulated (and therefore not totalled): `COLUMNS:  TOTAL–SALES(NOACCUM)`
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 127. 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 550.) For more information on using a display format parm, see page 132. 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)`

COLUMN STATEMENT PARMS	
PARM	**DESCRIPTION**
LEFT/CENTER/RIGHT	Specifies how to justify the contents of a column. For more information on using a justification parm, see page 142. 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 131. 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 Report 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 34** on page 129.

If you do not specify a column heading for a field in the COLUMNS statement, Report 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 33** on page 125.)

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" beginning on page 147.

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 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. Some PC keyboards that emulate mainframe terminals do not have a key that shows the straight vertical bar. (The "pipeline" character is *not* the same as the vertical bar.) On many of these keyboards, the right–hand square bracket key (]) is used to send a vertical bar to the mainframe while in emulator mode.

You may use the HDGSEP parm of the OPTION statement to select a **different character** to use as the separator character for column heading texts. (See page 501.) 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')
```

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)
```

Notes:

- 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 34 Specifying your own column headings

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 296) or the COMPUTE statement (page 444.)

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 could 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, Report 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 503.) 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.

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 NOOVERPRINT parm of the OPTION statement (as described on page 504.)

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')
```

How to Change the Width of a Column

This section explains:

- how Report 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 35** on page 133.

Report 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, Report 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 lead to 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 35** (page 133) 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 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 beginning on page 132.) 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. Report 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 36** on page 135.

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
```

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***
```

Notes:

- 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 35 Specifying the width of report columns

Time values can be formatted in the following ways, among others:

```
12:34:56
12:35
```

Report Writer supports many different **display formats** that indicate exactly how to format a field in a report. A complete list of these display formats is found in Appendix B, "Display Formats" (page 550.)

If you do not specify a display format in the COLUMNS statement, Report Writer uses a default display format. This will be:

- the display format (if any) specified when the field was first defined (in a FIELD or COMPUTE statement), or
- the display format (if any) specified in a previous OPTIONS statement's FORMAT parm (see page 500.) (Use the FORMAT option if you want to change the way *all* dates, times or numbers in your report are formatted.)
- the default display format shown in the table on page 559.

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 above 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: you can change the delimiter used to format date fields by using 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
```

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
```

Notes:

- 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 36 Customizing the way dates and numbers are formatted in a report

Note: 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 40** on page 145 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.

Formatting Tips for International Users

This section

- suggests some options that international users may wish to use when creating reports.

The following table contains a number of options of special interest to international users. The report in **Figure 37** (page 139) 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
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

OPTIONS OF INTEREST TO INTERNATIONAL USERS		
OPTIONS Statement Parm	**Description**	**Example**
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 Report 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 Report Writer Copy Library. Then, under MVS, use the SWOPTION DD to point to that member. Report Writer will process the statements in that member before it processes the other control statements (page 368.) 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
```

Notes:

- 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 37 A report with international formatting options

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 control group** is handled

Most of the features discussed in this section are illustrated in the sample report in **Figure 38**.

The NOREPEAT parm in a COLUMNS statement tells Report 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 Report 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

          EMPL       EMPL     SALES
REGION    NAME       NAME      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
```

Notes:

- 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 38 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 39** on page 143.

By default, Report 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: 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 area reserved for the entire column. Justification, if any, is performed only within the (smaller) area actually used to display the field's contents.

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
```

Notes:

- 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 39 Specifying how to justify data within the report columns

How to Specify Which Columns to Total

This section explains:

- how Report 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 40** on page 145.

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, Report 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, "Sample File Definitions.") The special characters in the PICTURE (namely the parentheses) suggest that totals, averages, etc. would not be appropriate for this field.

To state Report 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
```

Notes:

- 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 40 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, "Sample File Definitions.") 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, Report 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:** 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 40**. (For information on specifying PICTURES, see page 393.)

Note: to suppress the entire total line at a control break, see page 193.

Note: to suppress the entire Grand Total line, use the NOGRANDTOTAL parm on the OPTION statement. For more information on customizing the Grand Totals, see page 214.

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 41** on page 148.

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 154.

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.

A few tips will help your multi–line reports look better.

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 124; width parms are discussed on page 131.) Consider the sample report in **Figure 41**. 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 neatly.

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
```

Notes:

- 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

Figure 41 Using multiple COLUMNS statements to print multi–line reports

Use the **DOUBLE parm** of the OPTION statement (page 507) 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 Report 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.)

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 Report 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 override the default column headings as desired.

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. 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.

If you want to **underline** your columns headings, use a final TITLE statement that contains nothing but underscores and blanks. Report 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 41**.) 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 214.

> **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 OPTION statement (discussed on page 502.) 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 3 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 Report 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 the section beginning on page 132, 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 appear as a bar graph in the report. By default, bar graph columns are 20 characters wide. The column will contain a number of

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
```

Notes:

- 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 42 A report with a bar graph column

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 the COMPUTE field using the BAR display format. This is illustrated in **Figure 42** (page 151.)

Also, you may use an override column width parm to increase (or decrease) the default column width of 20 characters. The report on page 151 shows a bar graph column that is 30 characters wide. (The use of the width parm was discussed beginning on page 131.)

How to Print Vertical Lines between Report Columns

Report 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 43** shows a report that uses the above statement.

Note: the vertical bar is the Shift "1" key on most mainframe terminals. Some PC keyboards that emulate mainframe terminals do not have a key that shows the straight vertical bar. (The "pipeline" character is not the same as the vertical bar.) On many of these keyboards, the right–hand square bracket key (]) is used to send a vertical bar to the mainframe.

PC File Note: the COLSEP parm should not be used when creating PC files. Report 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
```

Notes:

- the COLSEP option specifies a 3–character "column separator" text, consisting of a vertical bar surrounded by blanks

Figure 43 A report with vertical lines separating the columns

How to Print a Variable Number of Lines Per Input Record

In some input files the records may contain an unknown, variable number of occurrences of a field. The SKIPZERODET option may be useful in such cases. It causes Report 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 on page 155. 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 3 fields there is an array of 6 sales slots. Each "slot" contains the date and the amount of a sale. But not all 6 slots are actually filled in for each record. As you can see, some records have only 1 slot filled in. Others have 2 or 3. One record has all 6 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 slots. We'll consider two different strategies to accomplish this objective.

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, Report Writer will always print 7 lines for each input record (one line per COLUMNS statement.) The resulting report is shown on page 156. 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 Report 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 on page 157 illustrates this strategy. (Note that we also specified DOUBLE 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)

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.

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)
```

Contents of SALES–HISTORY file:

```
BAKER     BOSTON    2921201004239893010400912250000000000000000000000000000000000000000000000000000
CHAVEZ    MIAMI     1930125018890100000000000000000000000000000000000000000000000000000000000000000
JEFFERSON CHICAGO   2930120006675593012300442340000000000000000000000000000000000000000000000000000
JOHNSON   DALLAS    5921230010081093010200554759301100075065930111002998093011900301620000000000000
JONES     ATLANTA   6921229007110592123000192569301080109023930110005247593011300789129301160120030
MORRISON  NEW YORK  3930102005220093010400919449301060140246000000000000000000000000000000000000000
SHARP     PORTLAND  1930131006001900000000000000000000000000000000000000000000000000000000000000000
SMITH     ST LOUIS  4930119003342393012100708109301240100056930128002007293013100941990000000000000
```

Notes:

- 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 44 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 45 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 46 Strategy 1 — just add the SKIPZERODET 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 2 or more sales slots 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" (page 162) 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 tells Report Writer that it may split the detail lines for a single input record across pages in the report. If you do not specify this option, Report 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 6 sales slots filled in (thus requiring 7 report lines in the example on page 157), Report Writer would skip to the next page if there were not 7 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 Report Writer to fill up each page before going on to the next page of the report.

> **Note:** remember that anytime multiple COLUMNS statements are specified, Report 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 slots in a record are 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, like 99/99/99). 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 contain good data. In our present example, we create a COMPUTE field for each of the 6 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:')

COLUMNS: SALE–ID–1  SALE–DATE–1  SALE–AMT–1
COLUMNS: SALE–ID–2  SALE–DATE–2  SALE–AMT–2
...
```

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. 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 on page 161 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 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
...
```

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 47 Adding literal identifiers to variable lines

If there is no character field for you to test, you can even test the date or amount field itself. Remember that Report Writer considers any conditional expression "false" if one or more of its operands contain invalid data. So, if your slot contains hex zeros for unused slots (which is "invalid data" for YYMMDD fields and for most numeric fields), you could use these COMPUTE statements:

```
COMPUTE: S–DATE–1 = WHEN(SALE–DATE–1 = SALE–DATE–1)  ASSIGN(SALE–DATE–1)
COMPUTE: S–AMT–1  = WHEN(SALE–AMT–1 = SALE–AMT–1)    ASSIGN(SALE–AMT–1)
```

The WHEN parm in the first statement above will be true if SALE–DATE–1 contains any valid date, and will be false if it contains invalid data. Likewise, the WHEN parm in the second statement will be true if SALE–AMT–1 contains any valid value, and false if it contains invalid data.

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)
```

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.

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, Report 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, under MVS, change the JCL to specify a larger LRECL for the SWOUTPUT DD (page 362.) Report 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 374.)

  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 506) for information on doing this.

If you are printing on a regular line printer:

- shorten long column headings, by rewording them, or by breaking the heading up into several lines (see **Figure 34**, page 129.) See the section titled "How to Change the Column Headings" beginning on page 127.
- shorten the width of one or more columns. See the section titled "How to Change the Width of a Column", beginning on page 131.
- 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 56** on page 185.
- use multiple COLUMNS statements to create a report with more than one report line for each input file record. See the section titled "How to Produce Multi–Line Reports" beginning on page 147.

Why Do I See ****X**** in My Report?

This section explains:

- why **asterisks** sometimes appear in your report

Sometimes an error prevents Report Writer from being able to display the desired data in a report. Rather than abandon the whole report, Report 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 582) 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.
`****E****`	Error in definition.
`****F****`	Error computing a field's offset value.
`****I****`	Invalid data.
`****S****`	Size error (not enough room to print all 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 (page 165)
- how to put the **page number, date and time** in your titles (page 172)
- how to change the way **dates, times and numbers are formatted** in the titles (page 170)
- how to split the title into **left, center and right** parts (page 174)

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 Report 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.

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 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 Report 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 the field's value from the *next* record that would print in the report.

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" for a complete list of built–in fields)

Figure 48 (page 167) 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 56** on page 185.)

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)
```

Notes:

- 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 48 A report title that includes data from a file

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. Report Writer has a number of **built–in fields** that can be used for this purpose. You may use these fields in your TITLE statement just like real fields from input files. The built–in fields available are:

BUILT–IN FIELD NAME	**CONTAINS**
#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 (formatted 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 Report Writer

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)
```

Notes:

- 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 49 A title that shows the current day of the week, date, time and page number

The sample report in **Figure 49** 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 line up 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 180.)

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
- **justify** the contents of fields printed in the title

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 with one or more 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 what parms are available in the TITLE statement. The sample report in **Figure 50** (page 173) illustrates the use of each these parms.

TITLE STATEMENT PARMS

PARM	DESCRIPTION
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" (page 550.) This parm works just like the display format parm in the COLUMNS statement, which is explained in more detail beginning on page 132. 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)`
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 142. The section titled "How to Split the Title into Left, Center, and Right Parts" (page 174) 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 131. 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)`

How to Change the Appearance of Items in the Title

If a field is specified in a TITLE statement without any parms, Report Writer chooses a default width, display format and justification.

Notice in the sample report in **Figure 50** 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.

For a similar problem that can arise when dates are lined up over the *right margin* of a report, see page 177.

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)
```

Notes:

- 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 50 Using width, display format and justification parms in the title

How to Split the Title into Left, Center, and Right Parts

This section explains:

- how to split the title into **left, center and right 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 our 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 the section titled "How to Include Data from a File in the Title" on page 165 for a review of what a print expression is.

Each print expression is called a **title part**. Report 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 51** shows an example of such a title.

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 52** (page 176) shows an example of a title that has 3 parts.

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)
```

Notes:

- the slash in the TITLE statement splits the title into left and right parts

Figure 51 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)
```

Notes:

- 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 52 A report with left, center, and right title 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 where one of the parts is 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 53** (page 178) 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 41** (page 148.)

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 Report 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 54** (page 179) shows an example of this.

The reports on pages 178 and 179 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.

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. 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 54** on page 179 shows a sample report that uses this technique to correctly right align the current date in a title.

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 50**

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)
```

Notes:

- the built–in fields #TODAY, #TIME24, and #PAGENUM are utilized
- 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 53 Titles with the date, 24–hour time, and page number on the left side of the report

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)
```

Notes:
- 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 54 A title with the date (spelled out), time, and page number on the right side of the report

(page 173) 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 title parts that are aligned over the left or right margins. In these cases, Report Writer shifts the center title part to prevent overlap.

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 55** 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 165) 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. (See page 172)
- separate the footnote line into left, center, and right aligned parts, by using slashes within the FOOTNOTE statement. (See page 174)
- 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. (See page 165)
- specify exactly how data should be formatted in the footnote, by using the width, display–format, and justification parms. (See page 170)

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
```

Notes:

- 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 55 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 be named 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 183)
- how the **default total line** looks, and tips on getting the most out of it (page 186)
- how to print **break–wide percentages and ratios** in the total line (page 187)
- how to **customize the total line** using the TOTAL parm (page 190)
- how to **suppress totals** at a control break (page 193)
- how to **print statistical lines** using the AVERAGE, MAXIMUM, MINIMUM, NZAVERAGE and NZMINIMUM parms (page 194)
- how to print **customized "footing" lines** at the end of a control group using the FOOTING parm (page 196)
- how to print the **number of items contained** in a control group (page 206)

- how to print **customized "heading" lines** at the beginning of a control group using the HEADING parm (page 208)

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, Report Writer prints two blank lines whenever a control break occurs. (These blank lines print *after* any footing lines, total lines and statistical lines have printed.) For example, the sample report in **Figure 12** (page 53) 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 the next page.) By coding the appropriate value for this parm, you can request that some other 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 *5 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 56 (page 185) 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 a laser printer, using both sides 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 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 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 page number to "one".
NEWSHEET	Skips to a **new sheet** of paper. In order for this feature to work, you must also use the OPTION 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 starting on page 506.)
NEWSHEET1	Works like NEWSHEET, but also resets 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 page number to "one".

PC File Note: only the **n** spacing parm (for "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.

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)

Notes:

- 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 56 A BREAK statement that requests a page break and resets the page number

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, Report Writer prints one total line at every control break. The report in **Figure 56** (page 185) 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 the section that begins on page 211.)
- 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 the section beginning on page 144.)

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, **Report Writer skips to a new line** to print the column totals.

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 67** (page 219) 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 190).

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 131.) 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 190.) Be sure to specify a width parm (page 201) 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 beginning on page 206.

Computing True Percentages and Ratios at Control Breaks

By default, Report 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 Report 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 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 57 (page 189) 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)
  ```

- Neither the numerator nor the denominator may be literal values. Each must be either a field or an expression. That is, 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.

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                                                     PERC       PERCENT
   NAME     REGION    CUSTOMER        TAX        AMOUNT     TAX         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
```

Notes:

- 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 57 Using the DIVTOTS parm to get accurate percentages at control breaks

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

Report 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, Report 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 58** shows a sample report that uses the above BREAK statement.

The contents of the TOTAL parm is actually a **print expression**. Print expressions tell Report 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.

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 196) 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.

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
```

Notes:

- the total line now begins with the text "REGION TOTALS", as specified in the TOTAL parm of the BREAK statement

Figure 58 A report with a customized total line at the control breaks

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 196.) 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 214.

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 just suppress totals for a **particular column**, see page 144.

Note: to suppress the **Grand Total** line, see the section beginning on page 214.

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 59** (page 195) 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

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.

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

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
```

Notes:

- the print expression in parentheses after each statistical parm determines the initial wording of the statistical lines
- to customize the Grand Total statistical lines, we could add another BREAK statement (see page 214)

Figure 59 A report that prints statistical lines (average, maximum, minimum) at control breaks

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 196) 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 144.

Note: notice the statistical lines after the Grand Totals on page 195. They still begin with the default wording (****** AVERAGE VALUE, etc.) To customize the statistical lines at the Grand Totals, see page 214.

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 106.

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
```

Notes:

- the footing line (specified in the BREAK statement) prints before the total line at each control break

Figure 60 Using the FOOTING parm to print a customized line at a control break

Report 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 do that.

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 60** (page 197) 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 pages 190 and 194.) 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 208.

The contents of the FOOTING parm is simply a **print expression**. Print expressions tell Report 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.

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 Report 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)
- a **built–in** field (see Appendix C, "Built-In Fields" 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 60 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 with one or more 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 what parms may be used in BREAK statement print expressions:

BREAK STATEMENT PRINT EXPRESSION PARMS	
PARM	**DESCRIPTION**

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

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 550). This parm works just like the display format parm in the COLUMNS statement, which is explained in more detail beginning on page 132. 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 142. 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	
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 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 131. 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 width, BIZ, display–format and justification parms specify *how* a data field will appear 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 in the previous 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 61 (page 203) shows a sample report which uses the above statement.

Notice that there are two different ways to use the statistical keywords TOTAL, AVERAGE, MAXIMUM, etc:

- 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 pages 190 and 194 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 Report Writer to print the average value of AMOUNT for the control group.

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
```

Notes:

- the footing line contains the AMOUNT field's *average* value for each region
- the example on page 205 shows how to remove the excess space that appears between the text and the average value in the footing line

Figure 61 A report which prints a field's average value in a footing line

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 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 61** (page 203.) **Figure 62** (page 205) 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 (see the section beginning on page 193.)
- 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 in **Figure 62** 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 same order as they appear 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 62** 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)

Notes:

- 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 62 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.
#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.
#ITEM–ENDING	This 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 the word.

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 63 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
```

Notes:

- 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 63 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 64 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 same order as the HEADING parms appear in.

The contents of the HEADING parm is simply a **print expression**. Print expressions tell Report 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.

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" (page 196) 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 with numeric and time fields in the HEADING parm's print expression. The reason, of course, is that Report 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 printed

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
```

Notes:

- 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 and total lines stand out

Figure 64 A report that prints control group headings

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. Such heading lines print after the report titles and column headings, and before the first detail line of the report. The value used for all fields appearing in a repeated heading line is taken each time from the *next* detail record after the heading line.

Printing a "Line Number" in Your Report

You have already seen how to use the #ITEM built–in field in BREAK statements. 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 68** (page 221) uses the #ITEM built–in field.

Note: #ITEM may also be spelled #ITEM1.

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. Report 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 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.

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 65** (page 213) 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 Report 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, Report 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 Report 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, Report 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 183). For example:

```
SORT:  REGION  EMPL–NAME  CUSTOMER
BREAK: REGION     SPACE(PAGE)
BREAK: EMPL–NAME  SPACE(1)
```

Or, to specify the same spacing parms in the SORT statement:

```
SORT: REGION(PAGE)  EMPL–NAME(1)  CUSTOMER
```

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
```

Notes:

- 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 65 A report with two levels of control breaks

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

Report 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, Report Writer prints a total line at this special control break. This break total line is what appears as the "Grand Total" line.

You may customize the Grand Total control break by using a BREAK statement, just like you do for regular control breaks. Use the special field name #GRAND on the 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 66** shows a sample report that uses this BREAK statement

You may use any of the BREAK statement parms except for SPACE in the BREAK statement for #GRAND. See the section titled "Customizing the Control Breaks" (page 182) 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. (Use the FOOTNOTE statement to print lines at the bottom of *each* page.)

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 headings). If the REPEAT parm is also specified, the HEADING lines will be repeated at the top of each page of the report.

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
```

Notes:

- 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 66 A report with customized Grand Totals

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 real 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 real 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 breaks specified a SPACE parm of NEWSHEET, then the Grand Totals will also be printed on a new sheet of paper. Otherwise, if any real control breaks specified ODDPAGE, then the Grand Total will also go on the next odd page. Otherwise, if any real control break specified PAGE, then the Grand Totals will go on a new page.

In addition, if the NEWSHEET1, ODDPAGE1, or PAGE1 parm was used in any of these cases, 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 68 (page 221) 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
```

(This page left blank intentionally.)

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 62.) And a lesson in Chapter 3, "How to Request a PC File" showed how to create summary PC files (page 110.)

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 will print. This is the same as specifying:

  ```
  OPTIONS: DETAIL(0)
  ```

  The only lines that print in such a report are lines 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. This prevents two blank lines from appearing between every line in the summary report (the default break spacing value.)

Figure 67 on page 219 shows another example of a summary report. This report contains *two* levels of breaks. It is very similar to the detail report shown earlier in **Figure 65** (page 213). The main difference is that in **Figure 67** 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 natural since only numeric columns can be totalled, or "summarized." Therefore, in this report we eliminated the non–numeric columns such as REGION, EMPL–NAME, SALES–DATE, etc. 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.

> **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 a report.

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
```

Notes:

- 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 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 67 A summary report that uses two levels of control breaks

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 Report 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 68** 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 68**, the "SOUTH" region is an example of this. There are only 2 sales for that region.)
- the control group *totals* will still contain the total value of the entire control group — not just the total of the records that are printed. You can use the NOTOTALS parm in the BREAK statement to suppress the totals if you prefer (as we did in **Figure 68**).
- 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, the first nnn records of the report will print.

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
```

Notes:

- the DETAIL(3) option causes only 3 detail lines per control group to print
- the #ITEM built–in field 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 68 "Top 3 Sales in Region" report

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 69** and **Figure 70** 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 69** 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 70** (page 224) 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. You would have to run a different report, for example, to count the number of male and female employees.

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
```

Notes:

- 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 69 Counting how many times something occurs 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
```

Notes:
- 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 70 Breaking down "count" statistics further

(This page left blank intentionally.)

How to Total a Field by "Category"

This section explains:

- how to compute totals "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 a 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 0 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 0. 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 71 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 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.

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
```

Notes:

- in the detail lines, MALE–SALES and FEMALE–SALES each contains either 0 or the value from the TOTAL–SALES field.
- the totals for those fields show the total sales made by male and female employees

Figure 71 Accumulating fields by a category (such as gender)

Working With Multiple Input Files

The following sections discuss various topics involving runs that use multiple input files. The topics discussed are:

- reading **more than one record from the same** auxiliary input file (page 228)
- how to use a field from one auxiliary input file as the **READKEY for another auxiliary file** (page 230)
- how to assign and use **record names** (page 232)
- how **"missing" records** are handled (page 233)
- how to read records using **generic and KGE** (key greater than or equal) keys (page 234)
- how to perform **"one–to–many"** reads by reading more than one record for each READKEY value (page 235)

Using Multiple READ Statements for the Same File

This section explains:

- how to read more than one record from the same auxiliary input file

In Chapter 2, "How to Request a Report" we learned how to produce a report using two auxiliary input files. (See **Figure 19** on page 71.) 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. Report Writer allows you to have an unlimited number of READ statements for the same file. The sample report in **Figure 72** 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

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)
```

Notes:

- 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

Figure 72 A report with multiple READ statements for the same file

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. Report Writer allows file chaining to any level.

Let's look at an example of file chaining. In the sample report in **Figure 72** (page 229), 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, "Sample File Definitions.") By reading the STATE–FILE record we can obtain the full state name for use in our report. **Figure 73** 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 Report 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.

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)
```

Notes:

- 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 73 A report with chained READ statements

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

Report 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. For example, a field named EMPL–NUM exists in both the EMPL–FILE and in the SALES–FILE (see Appendix F, "Sample File Definitions.") If a particular report uses *both* of these files as inputs, simply specifying EMPL–NUM as a field name would be *ambiguous*. You 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.

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, it is *not required* that you always use it when referring to fields from that file. Just use it when 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 are allowed to 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 72** (page 229) illustrated the use of record names.

How Missing Records Are Handled

This section explains:

- what happens when no record is found for a particular read key
- how to test whether or not a read was successful

Sometimes the auxiliary input file will *not* contain a record with a key equal to the read key's value. When this happens, Report 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:

FIELD TYPE	DEFAULT VALUE FOR MISSING RECORDS
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

A simple way to determine whether a record was successfully read or not is to examine the contents of some field in that record. Pick a character field from the record that would *not* normally be blank— perhaps the key field itself.

For example, assume we are using the SALES–FILE as the primary input for a report. In the report we also want to print the salesperson's department number. The DEPT–NUM field is in the EMPL–FILE, so we have a READ statement for the EMPL–FILE. Now assume that some employee numbers found in the SALES–FILE will *not* have corresponding records in the EMPL–FILE.

We need to test whether we have successfully read an EMPL–FILE record for each person. For those people who do not have EMPL–FILE records, we want the report to list their department number as "9". You could use the following statements to do this:

```
INPUT:   SALES–FILE
READ:    EMPL–FILE  READKEY(EMPL–NUM)
COMPUTE: DEPARTMENT = WHEN(LAST–NAME = ' ')  ASSIGN(9)
                      ELSE                   ASSIGN(DEPT–NUM)
COLUMNS: EMPL–NAME  DEPARTMENT  SALES–DATE  CUSTOMER
```

Notice that in the COLUMNS statement above we did *not* list the DEPT–NUM field from the EMPL–FILE. Instead we computed a new field named DEPARTMENT. Its value is equal to the DEPT–NUM field from the EMPL–FILE, *except when the EMPL–FILE record is missing.* To determine if that record is missing, we examine the LAST–NAME field (which is in the

EMPL–FILE.) When the LAST–NAME field is equal to blanks, we know that the read was not successful. In that case, we assign a value of "9" to the DEPARTMENT field.

Using Generic and KGE Keys

This section explains:

- how to use **generic** READKEYs
- how to read records that are **greater than or equal** to the READKEY

By default, Report Writer assumes that the value in the READKEY parm specifies an entire, exact key. When performing the READ, Report 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, Report 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 indicated result:

```
                                                   KEY OF RECORD
STATEMENT                                              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 Report 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:

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 Report 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 Report 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, Report 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, Report Writer will now read *all of* the SALES–AIX records that match the EMPL–NUM in the EMPL–FILE record. The report in **Figure 74** uses the above statements.

Here's how Report Writer processed the input files in **Figure 74**. It first read a record from the primary input file, EMPL–FILE. That record had an EMPL–NUM of 036.

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

                                         EMPL SALES
                                         FILE  AIX
     LAST           FIRST        HIRE    EMPL EMPL   SALES
     NAME           NAME         DATE    NUM   NUM    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
```

Notes:

- the MULTI parm in the READ statement causes Report Writer to read multiple records from the SALES–AIX file for each record read from the EMPL–FILE

Figure 74 A report that uses the MULTI parm

Report 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", Report Writer then produced one line of the report.

Then, *before* reading the next record from the EMPL–FILE, Report 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, Report 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.

For a more complete description of how Report Writer processes MULTI–type READ statements, see the Notes section of the READ statement in Chapter 9, "Control Statement Syntax" (page 520).

> **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). This may 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 lines to produce your Report 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, Report 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 Report 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), Report 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, Report Writer would set the SAVE–NAME field to blanks whenever the REC–TYPE field was not equal to "A".)

Figure 75 (page 239) shows a sample "batch" type file with header and detail records. The lower box on that page shows the Report Writer definition statements for the file. **Figure 76** (page 240) shows a PC file produced from this sample batch file.

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. (Report 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 Report Writer report (or PC file). This is because you don't want to write out a report line containing just the 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.)

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)
```

Notes:

- 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 Report 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 75 An input file with header and detail records, and its definition statements

These control statements:

```
OPTION:     LOTUS
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
```

Notes:

- 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 Lotus output file
- The COLUMNS statement creates a column in the Lotus file for the SAVE–NAME field taken from the header record, as well as for the two fields from the detail records

Figure 76 A Lotus file produced from an input file with header and detail records

- 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, Report 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.

Creating PC Files from Existing Reports

This section shows how to:

- turn **existing mainframe reports** into PC files for your favorite PC program
- how to use the **RETAIN parm** in the COMPUTE statement

Normally Report 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 program such as Lotus 1–2–3. Report Writer can let you automate that process, increasing accuracy and saving hours of manual work.

The technique used is to first write your existing report to a *file* (rather than to a printer). Then simply define this "report file" to Report Writer as if it were any other input file. Consider the sample mainframe report in **Figure 77**. 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 Report 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 77** shows Report Writer definition statements for this sample report. (We'll explain shortly the other fields defined in this 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 77** 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.

```
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 77 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 78 Report Writer statements to define the "report file" shown above

- 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 77** 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 Report 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 Report Writer ignore certain lines in your report.
- 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 77** 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 Report 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 Report 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 **Figure 77**, an easy way to do that is with the following statement:

```
INCLUDEIF: COL40 = '/'
```

The above statement tells Report 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 77** 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 77** 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 Report Writer to capture data from the title lines as they are processed and "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 thru 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 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  ...
```

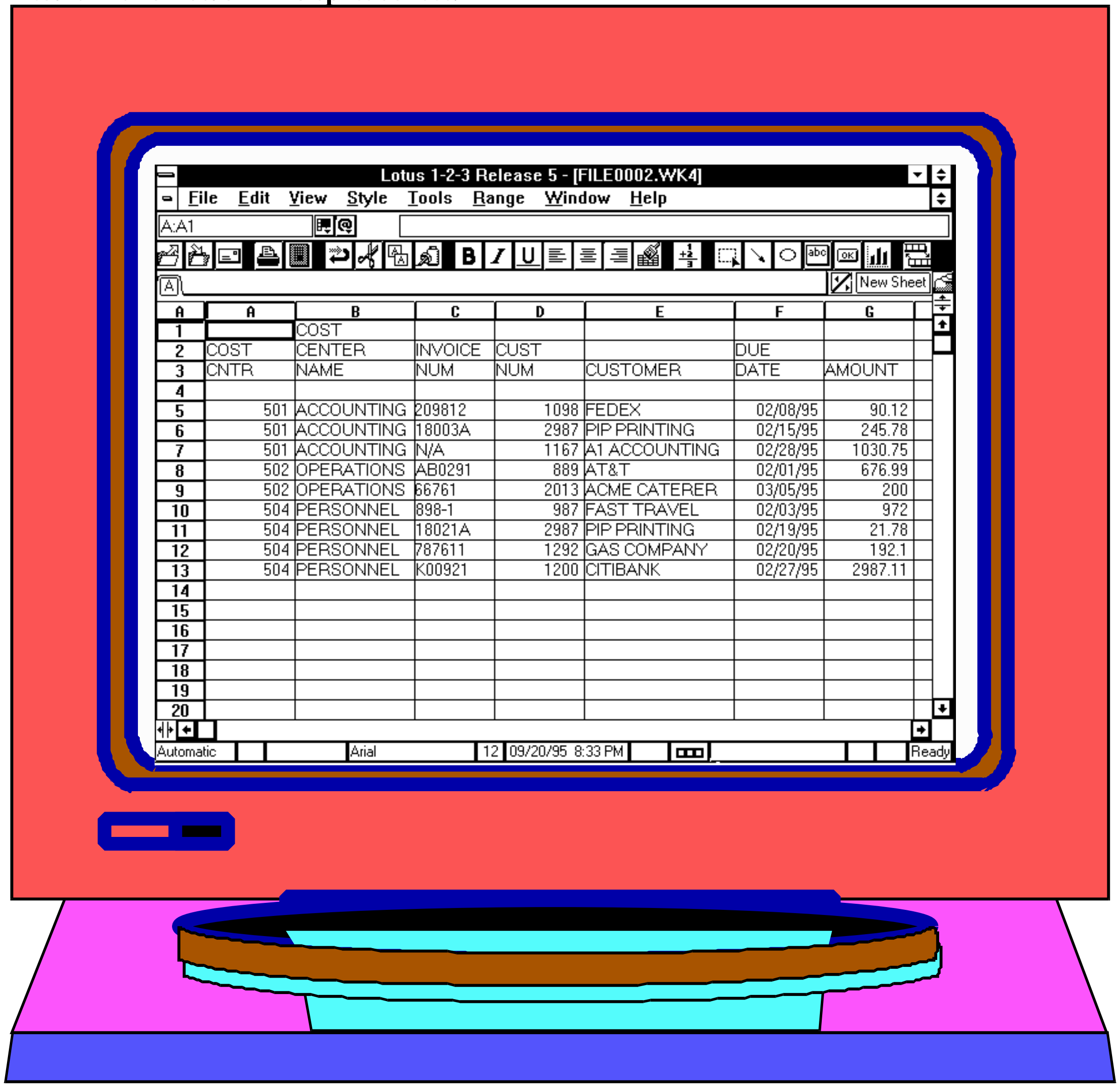
Why couldn't we simply put TITLE–COST–CNTR and TITLE–CC–NAME directly in our COLUMNS statement? Remember that our INCLUDEIF statement is written to include only the detail report records in our PC file. And the cost center is not present in the detail records. The same columns where the cost center appears 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."

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:



Lotus 1-2-3 Release 5 - [FILE0002.WK4]
File Edit View Style Tools Range Window Help
A:A1
New Sheet
Automatic
Arial
12
09/20/95 8:33 PM
Ready

A	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 79 Creating a Lotus 1–2–3 spreadsheet from a mainframe report

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 Report Writer's extracting and PC–formatting power to the existing reports in your shop.

Figure 79 (page 245) shows an actual example of creating a Lotus 1–2–3 spreadsheet from the report shown in **Figure 77** (page 242). Notice that we had Report Writer re-sort the PC file into cost center and due date order.

Working with SMF Records

You can use Report Writer to produce many useful reports from your shop's SMF files. In addition, Report 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 Report Writer with SMF files.

The SMF files are among the most complicated files in any shop. But Report 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 80** (page 247.)

- SMF records can be *big.* So to be safe, specify Report 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 Report Writer allocates a big enough I/O area to handle the largest SMF records.

- You should not need to specify DCB information in your DD statement. Report 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.

- Report Writer normally ignores the 4–byte RDW (record descriptor word) at the beginning of variable–length records (such as SMF records.) That is, Report 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
  ```

  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.

```
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 80 File definition of selected fields in SMF type 30 records

- When defining fields to Report 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
```

- Report 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 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 < 13:02:00)
```

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 Report 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 247.)

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. Report 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 = #MAKEDATE(#MAKENUM(#TODAY) –1)
    INCLUDEIF: REC–TYPE = 30  AND  SMF–DATE = YESTERDAY
```

- 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 Report 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 any 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 Report Writer.

Control Statements to Produce a "Daily ABEND" report (shown on page 251)

```
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
```

Control Statements to Produce a "TSO Sessions" Report (shown on page 252)

```
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
```

```
                                   BATCH JOB STEPS THAT ABENDED ON 04/15/94
                                       (522 AND 622 ABENDS NOT INCLUDED)
                                                                                 STEP      STEP        STEP
  JES    STEP                            COMP  JOB                PGMR          START      START        END
 JOBID    NUM JOBNAME  STEPNAME PGMNAME  CODE CLASS DPRTY         NAME           DATE      TIME         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 81 SMF "Daily ABEND" report produced by the control statements on page 250

```
                                             TSO SESSIONS ON 05/04/94
                                                                                                    STEP     STEP
                 PGMR          START      START        END          SESSION  SESSION    NUM     NUM    TCB      SRB
JOBNAME          NAME           DATE      TIME        TIME          MINUTES    COST   TPUTS   TGETS    SECS     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
```

Figure 82 SMF "TSO Sessions" report produced by the control statements on page 250

Working with Time Fields

This section offers some tips that you may find useful when working with time fields.

Report Writer supports two dozen different types of time fields commonly found in data files. These "time data types" are listed in Appendix A, "Data Types" (page 545.) For information on defining the time fields in your input files, see the section beginning on page 286.

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 like this:

```
HH:MM
```

A number of time display formats are available if you want to format your time fields differently. The time display formats are listed in Appendix B, "Display Formats" (page 557.) For example, you can specify the HH–MM display format if you want a time field to be displayed without showing the seconds. Report 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'Z9:99:99.99999')
```

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'ZZ,ZZ9:99:99')
```

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 the field is defined as having (which is usually 0.) 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 3 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 as HH:MM:SS with optional decimal parts of seconds also allowed. Here are some examples of using time fields and time literals in INCLUDEIF statements:

```
INCLUDEIF: START–TIME > END–TIME
INCLUDEIF: START–TIME > 12:00:00
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 the END–TIME and the 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 as 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`.

Report Writer has a built–in field named #HHMMSS which contains the system time that Report Writer began running. You can use this field like any other time field in creating reports or PC files.

> **Note:** Report 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)
```

Producing Files for Other PC Programs

Chapter 3, "How to Request a PC File" showed how to use Report Writer's control statements to produce PC files for various PC programs. Appendix H, "How to Import PC Files" gives specific information on how to import PC files into a number of PC programs.

The specific PC programs discussed in those areas are *not* the only ones that will accept files formatted by Report Writer. Most PC programs have very similar criteria for the files that they will import.

We will discuss three methods that you can use to create PC files for use in other PC programs. These three methods are:

- use the PC option to create a "standard" delimited ASCII file for PCs
- use your own combination of special options to specify in detail how the PC file is to be formatted
- 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 will import Lotus spreadsheets, use Report Writer to create a Lotus file. Then import the PC file into Lotus and save it as a spreadsheet file, which your PC program can then use. See page 600 for an example of this.

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
```

Figure 83 shows a sample PC file created using the above statement. The output 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

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", "DIF files", "DOS files" "ASCII files" or "text files".

These control statements:

```
OPTIONS:   PC
INPUT:     EMPL–FILE
COLUMNS:   LAST–NAME  HIRE–DATE   SALES–QTR1  SALES–QTR2
                                  SALES–QTR3  SALES–QTR4
```

Produce this PC File:

```
"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
```

Notes:

- 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 83 SMF "TSO Sessions" report produced by the control statements on page 250

Custom PC File

If your PC program does not import delimited ASCII 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 correctly. By specifying these options in various combinations you can create an output file in just about any format. (Each of these options is discussed in more detail in Chapter 9, "Control Statement Syntax," beginning on page 494.)

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 Excel does), try this statement:

```
OPTIONS: COLSEP(X'05')
```

FORMAT This option allows you to specify any display format you want as the *default display format*. 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.

Use this option to select 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 550.) For example, if your PC program requires that dates be formatted in Julian YYDDD format, you might use this statement:

```
OPTIONS: FORMAT(QCHAR, NOCOMMA, YYDDD)
```

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 dataset.

NOCOLHDGS This option suppresses all column headings from the report.

NOGRANDTOTAL This option suppresses the Grand Totals from the report.

NOTITLES This option suppresses all titles, footnotes and page breaks from the report.

OUTPUT You may specify the OUTPUT option parm, like this:

```
OPTIONS: OUTPUT
```

The OUTPUT option tells Report 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
- 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 NOCOLHDG 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
- 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 on–line Help (or its printed manual) for instructions on how to import fixed format files. Try using such keywords as "import", "fixed", "format", "ASCII", and "record".

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 84 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.

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)
```

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
```

Notes:

- 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 84 An output file created with the MAINFRAME option

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: 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.

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: FORMAT(YYYYMMDD)
COLUMNS: HIRE-DATE
```

The above statements cause the HIRE-DATE field (and any 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 550.)

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. Use the following options to suppress any or all of these items:

```
OPTIONS: NOTITLES  NOCOLHDGS  NOGRANDTOTAL  NOCC
```

When creating mainframe output files, you may want your records to be larger (or smaller) than the standard 133–byte output record. Chapter 7, "Operating System Considerations" explains how to specify any record length you want for your output file. See page 362 (MVS) or page 374 (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 Report Writer to sort mainframe files. One advantage of using Report 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.

(This page left blank intentionally.)

Chapter 5. How To Define Your Input Files

Chapter Table of Contents

Chapter 5. How To Define Your Input Files

This chapter is intended primarily for programmers "setting up" new files for Report 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.

Report 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 Report 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 Report Writer file definition simply consists of a single FILE statement, followed by a number of FIELD statements. (Appendix F, "Sample File Definitions" 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 Report 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 different column heading.) 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, you won't need to specify it 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 (page 269)
- the second section explains how to use FIELD statements to define each individual field within the file (page 275)
- the third section describes how to store these statements in Report Writer's copy library, to make requesting reports easy (page 301)
- the fourth section shows how to use Cobol or Assembler record layouts to define your files to Report Writer. You can use such record layouts *in place* of a Report Writer file definition. Or, you can use the record layouts *to create* a standard Report Writer file definition. (Page 311.)

Sometimes a picture is worth a thousand words. So, before we get into the details of how to define files, notice the box on the following page. It shows a typical Cobol definition of a file, and how the same file would be defined to Report Writer.

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 Report Writer (MVS) 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)
```

Notes:

- the FILE statement for Report 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, Report Writer always requires the *length (in bytes)* that a field occupies in the input record, rather than 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 4 individual fields for Report Writer

Figure 85 Converting a Cobol copybook to Report Writer definition statements

How to Define a File

This section explains:

- how to use the FILE statement to **define a file** to Report Writer
- how to later **override aspects of a file definition** in the INPUT or READ statement

Input files are defined to Report Writer with the FILE statement. If desired, the INPUT or READ statements can also be used to provide, or to modify, a file definition (for a single run.) (As a reminder, an INPUT statement is required for all runs, and specifies the primary input file for a run. READ statements are optional, and identify any additional input files required for a particular run.)

The following sections show how to use these statements to define your input files to Report Writer.

The parms used in the FILE statement differ between Report Writer MVS and Report Writer VSE. Please refer to the correct section for your operating system:

- for MVS, see below.
- for VSE, see page 273.

How to Use the FILE Statement –– MVS

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 470.) Only a few of these parms are actually *required*. The others are optional, and are only needed in unusual cases.

The four things that Report Writer *must* know about a file are:

- the **file name** (that is, the "user friendly" name by which it will be referred to in other Report 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 Report 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 **file name**. 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 388.) This is the name that will be used in Report Writer control statements when referring to this file. It does *not* have to be the actual DSNAME ("data set name") 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 Report Writer what type of file is being defined. This tells Report Writer which access method to use when performing I/O to the file. Report 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. Report 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 Report Writer is a VSAM file:

```
FILE:  EMPL–FILE  TYPE(VSAM)
```

The above statement defines a file named EMPL–FILE as being a VSAM file. Report Writer supports KSDS, ESDS and RRDS VSAM files.

Note: you can also use other types of files with Report Writer. However, you will need to write an I/O Exit program in order to do that. I/O Exits are discussed in Appendix K, “I/O Exits.“

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. Report 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 Report 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 Report Writer to provide a 5000–byte I/O area to use when reading records from this file. If no LRECL parm is present, Report Writer reserves a 1000 byte I/O area as a default.

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.

> **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.

Records in variable length SEQ 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, Report Writer ignores this RDW. Thus, a field defined as beginning in 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 any. If for some reason you want column 1 of your record to refer to the RDW, use the **KEEPRDW parm** in the FILE statement. For example:

```
FILE:  SALES–FILE  DDNAME(SALESDD)  KEEPRDW
```

The above statement tells Report Writer to consider the RDW as part of the input record's user data. Thus a field defined as starting in column 1 will point to the RDW within the record.

The only other parm available in the FILE statement is the **EXITPARM parm**. This parm is not normally used. However, if any of the fields defined for this file use a data exit program (see page 297), you may want to use this parm. Whenever a data exit program is called, it is passed certain information to assist it in preparing the data to return to Report Writer. One item of information that is passed to the data exit program is the contents of the FILE statement's EXITPARM parm. For example,

```
FILE:  SALES–FILE  DDNAME(SALESDD)  EXITPARM('ABCDEFG')
```

The above statement specifies the 7–byte text 'ABCDEFG' as the file's exit parm data. If any fields defined for the SALES–FILE are created in a data exit program, the string 'ABCDEFG' will be passed to that exit program when it is called. The exit program could then use this data in any way it wanted.

How to Override a File Definition –– MVS

Remember that the FILE statement simply *defines* a file to Report Writer for later use. It *does not* make that file an input file to a report. The INPUT and READ statements request a file as input for a particular report. When an INPUT or READ statement specifies a particular file, Report 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 just one particular run. You may do this by specifying one or more file definition parms directly in the INPUT or READ statement. These parms will override any such parm that may also have been specified in the FILE statement— but only for the current run. The file definition parms that can be specified in the INPUT and READ statements are:

- DDNAME
- TYPE
- LRECL
- KEEPRDW
- EXITPARM

For example, assume that the FILE statement for EMPL–FILE stated that the DDNAME to use was "SWINPUT." But, for one particular report you want to use a DDNAME of "EMPLOYEE" instead.

There is no need to change the FILE statement just to run this particular report. You would simply code an override DDNAME parm directly in the INPUT statement:

```
INPUT:  EMPL–FILE  DDNAME(EMPLOYEE)
```

The above example lets you use the EMPLOYEE DD for the current report without having to change the FILE statement (which may be located in the copy library and difficult to modify.)

Similarly, you can override the file's TYPE parm in an INPUT or READ statement. For example, assume that the FILE statement defined EMPL–FILE as being a "sequential" file. But you may have loaded a VSAM file from the sequential file and want to use that VSAM file as an auxiliary input for a report. You would override the file type, for that report only, like this:

```
READ:  EMPL–FILE  READKEY(EMPL–NUM)  TYPE(VSAM)
```

The above example causes the EMPL–FILE to be opened as a VSAM file, not as a normal sequential file.

You can also specify a different LRECL from the one specified in the FILE statement. Here is an example of specifying an override LRECL parm in an INPUT statement:

```
INPUT:  EMPL–FILE  LRECL(3000)
```

Similarly, if you need to specify a different exit parm text from the one specified in the FILE statement, do that in the INPUT or READ statement like this:

```
INPUT:  EMPL–FILE  EXITPARM('ABCXYZ')
```

How to Use the FILE Statement –– VSE

The FILE statement's ATTR parm is used to describe the attributes of a VSE file to Report 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
- the blocks in this file are 160 bytes long

Note: the complete syntax of the ATTR parm is shown on page 470.

Here is another example of defining a VSE file with the ATTR parm. In this example, we define a VSAM file to Report 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(DASD,'FILEIN',V,100,5000)
```

The file defined above is a SAM file on DASD. The DLBL name used in the JCL is FILEIN. 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.

Records in variable length SAM 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, Report Writer ignores this RDW. Thus, a field defined as beginning in 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 any. If for some reason you want column 1 of your record to refer to the RDW, use the **KEEPRDW parm** in the FILE statement. For example:

```
FILE:  VAR–FILE  ATTR(TAPE,'FILEIN',V,100,5000)  KEEPRDW
```

The above statement tells Report Writer to consider the RDW as part of the input record's user data. Thus a field defined as starting in column 1 will point to the RDW within the record.

The only other parm available in the FILE statement is the **EXITPARM parm**. This parm is not normally used. However, if any of the fields defined for this file use a *data exit program* (see page 297), you may want to use this parm. Whenever a data exit program is called, it is passed certain information to assist it in preparing the data to return to Report Writer. One item of information that is passed to the data exit program is the contents of the FILE statement's EXITPARM parm. For example,

```
FILE:  SALES–FILE  ATTR(DASD,'SALEFIL',80,160)  EXITPARM('ABCDEFG')
```

The above statement specifies the 7–byte text 'ABCDEFG' as the file's exit parm data. If any fields defined for the SALES–FILE are created in a data exit program, the string 'ABCDEFG' will be passed to that exit program when it is called. The exit program could then use this data in any way it wanted.

How to Override a File Definition –– VSE

The ATTR parm can also be used in the INPUT and READ statements. This temporarily changes the way a file is defined for a single Report 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
- the blocks in the file 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 Report Writer

There are five general types of fields used in Report 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 for the FIELD statement is given beginning on page 460.

> **Note:** Report Writer MVS and Report 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 86** on page 277.

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 282 for more on this subject.

How To Define a Character Field

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 388.

After the fieldname, the other parm(s) may appear in any order in the FIELD statement.

The **LENGTH parm** is required to tell Report 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)
```

Report 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 beginning on page 292. 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)
```

By default, whenever a field appears as a column in a report, the field name itself is used as the column heading. To specify a different column heading, use the **HEADING parm** in the FIELD statement. The use of the HEADING parm is discussed beginning on page 296. As an example, we could specify a column heading for the LAST–NAME field like this:

```
FIELD:  LAST–NAME  LENGTH(15)  HEADING('EMPLOYEE LAST NAME')
```

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 it is when you have a character field that you normally want to display in its hexadecimal representation. (A status byte might be an example of such a field.) 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)
```

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)
```

Notes:

- 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 as column headings

Figure 86 A report with FIELD statements that define character fields

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 87** on page 281.

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 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 Report Writer exactly how a field 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 with any other numeric field. Report Writer automatically takes care of any conversion that may be necessary. You may also mix any combination of numeric fields (packed, binary, etc.) 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 388.

After the fieldname, the other parm(s) may appear in any order in the FIELD statement.

When defining a numeric field to Report 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. Report 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 in Appendix A, "Data Types" (page 540.) 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.) Report 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 Report 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 85** on page 268.) 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.

Report 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 beginning on page 292. 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, Report 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 Report 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, as well as control break totals at each control break. 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 are accumulated and appear in the total lines, see page 144.

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 3 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 in Appendix B, "Display Formats" (page 552.)

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. 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
```

Page 393 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.

By default, whenever a field appears as a column in a report, the field name itself is used as the column heading. To specify a different column heading, use the **HEADING parm** in the

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
```

Notes:

- we used abbreviations for the COLUMNS, LENGTH and DECIMAL parms. See page 461 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 87 A report with FIELD statements that define numeric fields

FIELD statement. The use of the HEADING parm is discussed beginning on page 296. As an example, we could specify a column heading for the TOTAL–SALES field like this:

```
FIELD:  TOTAL–SALES  TYPE(NUM)  LENGTH(7)
        HEADING('YEARLY SALES TOTAL')
```

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 575.) 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 570.) 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 88** on page 285.

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 Report Writer exactly how a field 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. Report Writer automatically takes care of any conversion that may be necessary.

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 388.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

When defining a date field to Report 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. Report 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 in Appendix A, "Data Types" (page 542.) 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, Report 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 beginning on page 542. If Report 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.

Report Writer assumes that the HIRE–DATE field defined in the preceding example occupies the 6 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 beginning on page 292. 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)
```

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 in Appendix B, "Display Formats" (page 554.)

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.

By default, whenever a field appears as a column in a report, the field name itself is used as the column heading. To specify a different column heading, use the **HEADING parm** in the FIELD statement. The use of the HEADING parm is discussed beginning on page 296. As an example, we could specify a column heading for the HIRE–DATE field like this:

```
FIELD:  HIRE–DATE  TYPE(YYMMDD)  HEADING('DATE HIRED')
```

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)
```

Notes:

- 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 88 A report with FIELD statements that define date fields

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 89** (page 288).

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 time fields as binary hundredths of seconds since midnight. The S/370 STCK machine instruction represents times as the number of "timer units" since the beginning of the century.

Report Writer supports all of these kinds of time fields and about two dozen others. The FIELD statement's TYPE parm tells Report Writer exactly how a 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. Report 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:

- DECIMAL
- ACCUM/NOACCUM

The **fieldname** is always the first item in a FIELD statement. The rules for assigning field names are given on page 388.

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 in Appendix A, "Data Types" (page 545.) 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.

The **LENGTH parm** is generally not required for time fields. Depending on the particular data type, Report Writer assumes a default length for each time field. For example, the default length of a time field in HHMMSS format is 6 bytes. The default length of a time field in P–HHMM format is 3 bytes, and so on. The default length of each time data type is also shown in the table beginning on page 545. If Report 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.

Report Writer assumes that the SALES–TIME field defined in the preceding example occupies the 6 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 beginning on page 292. For example, if SALES–TIME begins in the 38th column of a record, we could define it like this:

```
FIELD:  SALES–TIME  TYPE(HHMMSS)  COLUMN(38)
```

You may also use the **DECIMAL parm** in the FIELD statement. Do this when 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 store their time 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, as well as control break totals at each control break. 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:  RESPONSE–TIME  TYPE(HHMMSS)  LENGTH(8)  DEC(2)  ACCUM
```

The above statement specifies that the RESPONSE–TIME field should be accumulated when it appears as a column in a report. Therefore, the RESPONSE–TIME column will be totalled at control breaks and at the end of the report. For a more detailed discussion about which fields are accumulated and appear in the total lines, see page 144.

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) will be formatted like this:

```
HH:MM
```

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
```

Notes:

- 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 89 A report with FIELD statements that define time fields

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(HHMMSS)
```

The above example specifies that whenever the SALES–TIME field is printed in a report, the HHMMSS format should be used. The HHMMSS format does not use colons to separate the hours, minutes and seconds (for example, "131059"). Or, you might specify the HH–MM display format if you want a time field to be displayed without showing the seconds. Report Writer will round the time 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 in Appendix B, "Display Formats" (page 557.)

> **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.

By default, whenever a field appears as a column in a report, the field name itself is used as the column heading. To specify a different column heading, use the **HEADING parm** in the FIELD statement. The use of the HEADING parm is discussed beginning on page 296. As an example, we could specify a column heading for the SALES–TIME field like this:

```
FIELD:  SALES–TIME  TYPE(HHMMSS)  HEADING('TIME OF SALE')
```

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 90** on page 291.

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 388.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

The **BIT parm** is required to tell Report Writer which specific bit (within a byte) the field refers to. Every byte contains 8 bits, which are numbered from 1 to 8, starting with the leftmost ("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 (or "high order") bit within the byte.

Report 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 the FULL–TIME bit is located, use the **COLUMN** or the **DISP parm**. The use of these parms is discussed beginning on page 292. 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 on the default location, see "How to Specify a Field's Location in a Record" (page 292.)

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')
```

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)
```

Notes:

- 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 90 A report with FIELD statements that define bit fields

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.

By default, whenever a field appears as a column in a report, the field name is used as the column heading. To specify a different column heading, use the **HEADING parm** in the FIELD statement. The use of the HEADING parm is discussed beginning on page 296. As an example, we could specify a column heading for the FULL–TIME field like this:

```
FIELD:  FULL–TIME  BIT(1)  HEADING('FULL TIME STATUS')
```

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 covered include:

- how a field's **default location** is determined
- how the default location works when defining **bit type fields**
- how to use the **COLUMN or DISP parm** to specify a field's location
- how to use the **FILE parm** to specify the file in which a field is located

Subsequent sections show:

- how columns are counted in **variable length** (VB) input files (page 294)
- how to use the **OFFSET parm** for variably located fields (page 295)

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 1. 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.

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 **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*. Report 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 some *other 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" is handled a little differently when working with bit fields. 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. 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.

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" — that is, 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   COLUMN(10)
FIELD:  FIRST–NAME  COLUMN(19) LENGTH(15)
```

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.)

Field Location in Variable Length Files

Records in non–VSAM variable length files begin with a 4–byte record prefix called the record descriptor word (RDW). This RDW appears before the actual user data in each record.

By default, Report Writer ignores the RDW in variable length input files. It treats your variable length input records as beginning immediately *after* the 4–byte 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, Report 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 Report Writer to ignore the RDW, use the KEEPRDW keyword in the FILE statement (or in the INPUT or READ 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 Report Writer to treat the RDW as part of each 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, "NAME", now starts in column 5.

Variably Located Fields

Some records contain fields that do not always begin at a fixed column in the record. In such cases there is usually another field within the record that tells the "offset" to the variably located field. Report Writer's OFFSET parm lets you easily define such fields.

The OFFSET parm can contain any numeric expression. Report Writer computes the value of the OFFSET parm for each input record. It *adds* this value to the value of your COLUMN or DISP parm and thus determines where the field is located within the input record.

For example:

```
FIELD:  ADDR–OFFSET  DISP(26)    TYPE(HALFWORD)
FIELD:  ADDR–LINE–1  LENGTH(30)  OFFSET(ADDR–OFFSET)
FIELD:  ADDR–LINE–2  LENGTH(30)
...
```

In this example, our input record contains a halfword value at displacement 26. This value is the offset within the record to an "address section" of the record. The address section consists of two 30–byte address lines.

Here are some points to keep in mind about the OFFSET parm:

- The "default location" value is reset to displacement 0 each time an OFFSET parm is encountered. ("Default location" means the default displacement assumed when you do not specify a DISP or COLUMN parm in the FIELD statement.) Thus ADDR–LINE–1 is treated as if it had a DISP(0) parm. Therefore, if ADDR–OFFSET contains a value of 100 in a particular record, ADDR–LINE–1 will be located at displacement 100 in that record. Of course, you can still specify your own explicit DISP (or COLUMN) parm if you don't want the default value. For example:

  ```
  FIELD: ADDR–LINE–X OFFSET(ADDR–OFFSET) DISP(15) LENGTH(30)
  ```

  This statement would cause ADDR–LINE–X to be located at displacement 115 in our example.

- An OFFSET parm remains in effect for *all subsequent* FIELD statements, until another OFFSET parm is found. Thus, the location of ADDR–LINE–2 is also determined by using the value in ADDR–OFFSET. Since there is no DISP parm present, the "default location" value is assumed. The default location is 30 for this field (since ADDR–LINE–1 took up 30 bytes in the record.) Thus ADDR–LINE–2 would be located at displacement 130 in the same record.

Use OFFSET(0) if you later want to define fields that do not need any OFFSET value. Remember that specifying OFFSET(0) also resets the default record location value to zero.

If you use an OFFSET parm in a member of the Report Writer Copy Library, it is a good idea to have a final FIELD statement that contains an OFFSET(0) parm. That way there will be no "surprises" if someone later adds more FIELD statements "inline" for a report request. They might not be aware that an OFFSET value was still in effect for their additional FIELD statements.

- A `***F***` error indicator in your report means that an "Offset Error" occurred for a 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 record size specified in a 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 contains invalid data.

The examples above used a single field as the OFFSET value. You are also allowed to use numeric expressions in the OFFSET parm. For example, to define a field that appears after an array of variable size, you might use this statement:

```
FIELD:  LAST–FIELD  OFFSET(75 + (NUM–SLOTS * 12))  LENGTH(10)
```

How to Specify a Field's Column Heading

This section explains:

- how to use the **HEADING parm** to specify column headings

The HEADING parm can be used when a 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.

Note: the vertical bar is the Shift "1" key on most mainframe terminals. Some PC keyboards that emulate mainframe terminals do not have a key that shows the straight vertical bar. (The "pipeline" character is not the same as the vertical bar.) On many of these keyboards, the right–hand square bracket key (]) is used to send a vertical bar to the mainframe.

You can also use the HDGSEP parm of the OPTION 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")
```

If no HEADING parm is specified when a field is defined, the **field name itself** will be used as the default column heading. All dashes or underscores in the field name will be used to separate the name into different column heading lines.

Note that the HEADING parm simply specifies the *default* column headings that will be used for the field. Override column headings can be specified in the COLUMNS control statement to change the column heading for a particular run.

For more complete information on specifying column headings, see page 127.

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 Report 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 Report Writer does not directly support
- data that exists in files that Report Writer cannot read directly, such as certain data base files

Even in such situations, Report 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 Report 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 date field that Report Writer can process
- perform its own I/O on the data base file, and pass back to Report Writer data from that file

Note: data exit programs are not included with Report Writer. They must be written by a programmer at your company. Most companies will not use data exit programs at all. The data exit interface is merely provided to give the maximum ability to use Report Writer at any shop — even those with very "non–standard" types of files or data. Appendix G, "Sample Data Exit Program" shows a sample data exit program and a run that uses it.

When Report 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 Report 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 Report 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 `C'0'` or `C'1'`

Once an exit program has passed data back to Report 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 Report 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

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 388.

After the fieldname, the other parm(s) may be specified in any order in the FIELD statement.

The **TYPE parm** is required to tell Report 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 **DXPROG parm** is required to tell Report Writer the name of the program that should be called to create the field's data.

The **DXRETLEN parm** is required for *character* fields that are created by a data exit program. This parm specifies the length of the character data that will be returned to Report Writer by the exit program.

The **DXRETDEC parm** is required for numeric and time fields that are created by a data exit program. This parm specifies the number of decimal digits that will exist in the packed number returned to Report 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, Report Writer must call an exit program. Therefore, the field is defined with the TYPE(CHAREXIT) parm. This specifies that a data exit program will be used, and that the exit program will return *character* type data to Report Writer.

The **DXPROG parm** supplies the name of the exit program to call. In this example, a program named DCRYPROG will be called. Under MVS, a load module by this name must exist in the library named in the STEPLIB DD in the JCL. Under VSE, a phase by this name must be in a sublibrary named in the "`// LIBDEF PHASE,SEARCH=...`" statement in the JCL.

When Report Writer calls that program it will pass it the 20 byte encrypted name, which begins in column 15 of the record. This is specified by the **COLUMN** and **LENGTH** parms.

The **TYPE** and **DXRETLEN** parms tell Report Writer to expect an 18–byte character value back from the exit program. It is this 18 byte character field 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 exists that can convert this date into the standard 4–byte `X'YYYYMMDD'` format date that Report Writer uses internally. The following statement could be used to define the field:

```
FIELD:  SPECIAL–DATE  COLUMN(17)  LENGTH(2)  TYPE(DATEEXIT)
        DXPROG('DATECONV')  FORMAT(LONG1)
```

The above statement defines a field named SPECIAL–DATE that can be used just as any other date field in Report Writer. It can be compared to other dates, printed using any date display format, etc. In this example, we have also specified the optional **FORMAT parm**. It specifies that this date field should be displayed using the LONG1 format, by default.

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 Report Writer cannot read the data base file directly, an exit program named READCOST is called to read a record from the file and return the unit cost as a 16–byte packed number. The 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')
```

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 preceding sections have shown how to write FILE and FIELD statements. (These statements are 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 **Report 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 Report 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 91** shows an MVS 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 92 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 one or more READ statements) be sure to include the definition statements for *each* of the input files.

This JCL:

```
//SPECTWTR JOB 'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,    REPORT WRITER REPORT
//                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
```

Notes:
- 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 91 A Report Writer report that does not use a copy library — MVS

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
```

Notes:

- 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 92 A Report Writer report that does not use a copy library — VSE

A Better Way: Using the Copy Library

There is a better way to handle the definition statements. Report Writer can *automatically* access the definition statements it needs for a particular report by using a "copy library." In MVS, 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 MVS programmers: the Report Writer Copy Library works in much the same way as the following programming language libraries:

- the Cobol copybook library (SYSLIB)
- the PL/1 INCLUDE library
- the SYSLIB macro and copy library, for assembler programs

We suggest you create a new PDS to serve exclusively as your Report Writer Copy Library. However, you can use any 80–byte PDS. Use the SWCOPY DD (in your execution JCL) to tell Report Writer what PDS you are using as the copy library. Chapter 7, "Operating System Considerations" gives more information on the SWCOPY DD and on setting up file definitions in your copy library (page 364.)

Note to VSE programmers: the Report Writer Copy Library works in much the same way as the following programming language libraries:

- the Cobol copybook library
- the PL/1 INCLUDE library
- the macro and copy library, for assembler programs

We suggest you define a separate sublibrary to serve exclusively as your Report Writer Copy Library. However, you can use any sublibrary you choose. Use the SUBLIB parm (in an OPTIONS statement) to tell Report Writer the name of your copy library. The *member type* for all members should be SPECTWTR. (Use the OPTIONS statement MEMTYPE parm if you need to use a different name for the member type.) Chapter 7, "Operating System Considerations" gives more information on setting up file definitions in your copy library (page 376.)

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 beginning on page 308.) After you have created a member in the copy library for a file, simply save its FILE and FIELD statements there. You can also add any COMPUTE statements that are commonly used with the file. That's all there is to adding a file to the Report Writer Copy Library.

Once a file's definition statements have been stored in the copy library, Report 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, Report Writer attempts to copy control statements from the copy library member that corresponds to that file. Those control statements then define the file for Report 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 93 (MVS) and **Figure 94** (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. To see the contents of the copy library members for the sample files used in this manual, see Appendix F, "Sample File Definitions."

By default, the control statements copied from the copy library *are not* printed in the control listing along with the other control statements. 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 library can also be used to store any other 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 455.)

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 does all of that for them.

Under MVS, the COPY statement can also copy sequential datasets that are *not* partitioned datasets. 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. 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)
```

Notes:

- 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 (which defines the EMPL–FILE) is automatically copied into this report request
- the following line appears in the SWALIAS member of the copy library:

```
EMPL–FILE = EMPLDEF
```

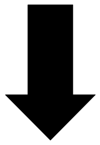
- the following statements are stored in the EMPLDEF member of the copy library:

```
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 93 A report which uses Report Writer's Copy Library — MVS

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)
```

Notes:

- the OPTIONS statement names LIB.SPECTWTR as the Librarian sublibrary to use as the Report Writer Copy Library for this run.
- as the INPUT statement is processed, the EMPLDEF.SPECTWTR copy library member (which defines the EMPL–FILE) is automatically copied into this report request
- 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 94 A report which uses Report 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, Report 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 itself 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.

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 Report 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 two names are the same, 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 Report 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 Report 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 Report 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 Report Writer as the member name for reading the definition statements from the copy library.

Consider the following statement:

```
INPUT: HEADQUARTERS–PAYROLL
```

When Report 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. Report 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 to remember 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, "Sample File Definitions" 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* to those kept in the copy library.

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 Report 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 (that is, 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, all of the your FIELD statements will be assumed to belong to the *most recently defined file*. (That is the file named in the most recent FILE statement.)

Using Cobol and Assembler Record Layouts

This section explains how to use Cobol or Assembler record layouts with Report Writer.

Earlier in this chapter you learned how to use FIELD statements to define an input file to Report Writer. Report 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 Report Writer's Cobol and Assembler interpreter.

1. In a "live" run. Provide a Cobol or Assembler record layout to Report Writer and produce a custom report or PC file in the same run. With this method, you never create a standard Report Writer file definition.

2. In a "conversion" run. Provide a Cobol or Assembler record layout to Report Writer and let it write corresponding FIELD statements to an output file. Save these FIELD statements in your Report 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 Report 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.)

Report Writer has two special control statements that are used when working with Cobol or Assembler record layouts. The COBOL statement tells Report Writer that a Cobol record layout is about to follow. The ASM statement tells Report 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 Report Writer control statement by that name. Thus "COBOL statement" means the Report 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 95 (page 314) shows an example of a report produced using a Cobol record layout. Let's examine the Report Writer control statements shown in the top box in that figure.

A FILE statement is always required when a file is about to be defined. It tells Report Writer the name of the file being defined. 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 Report Writer that subsequent control statements will be in the Cobol language, rather than in Report Writer's 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.) Report Writer processes the Cobol record layout, noting the names of the Cobol fields and their characteristics. Internally, Report Writer creates the equivalent of a FIELD statement for each Cobol field in the record layout.

Note: see "Technical Notes on Cobol Support" on page 328 for certain limitations on the Cobol syntax that Report Writer accepts.

Report Writer continues treating each subsequent line as Cobol code until it reaches a line that begins with a Report Writer control statement prefix. In this example, the line beginning "INPUT:" is recognized as a Report 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 under "The Scope of the COBOL and ASM Statements" on page 328.)

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, in the normal way.

Fields defined by a Cobol record layout can be used in all of the same ways as fields defined with FIELD statements. For example, you can use the Cobol field names in SORT statements, COMPUTE statements, BREAK statements, and so on.

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—that is, 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 on handling date and time fields, see "Handling Date and Time Fields" on page 318.

Note: To see a listing of the internal FIELD statements that Report Writer creates from a Cobol layout, add the SHOWFLDS(YES) parm to the COBOL statement. For example, assume we had added the SHOWFLDS(YES) parm to the COBOL statement on page 314:

```
COBOL:   SHOWFLDS(YES)
```

In that case, the following FIELD statements would have been printed 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)
```

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 95 A report produced using a Cobol record layout

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 96 (page 317) shows an example of a report produced using an Assembler record layout. Let's examine the Report Writer control statements shown in the top box in that figure.

A FILE statement is always required when a file is about to be defined. It tells Report Writer the name of the file being defined. 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.

The ASM statement tells Report Writer that subsequent control statements will be in the IBM S/370 Assembler language, rather than in Report Writer's 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.) Report Writer processes the Assembler record layout, noting the names of the Assembler fields and their characteristics. Internally, Report Writer creates the equivalent of a FIELD statement for each Assembler field in the record layout.

> **Note:** see "Technical Notes on Assembler Support" on page 330 for certain limitations on the Assembler syntax that Report Writer accepts.

Report Writer continues treating each subsequent line as Assembler code until it reaches a line that begins with a Report Writer control statement prefix. In this example, the line beginning "INPUT:" is recognized as a Report 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 under "The Scope of the COBOL and ASM Statements" on page 328.)

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, in the normal way.

Fields defined by an Assembler layout can be used in all of the same way as fields defined with FIELD statements. For example, you can use the Assembler field names in SORT statements, COMPUTE statements, BREAK statements, and so on.

> **Note:** To see a listing of the internal FIELD statements that Report Writer creates from an Assembler layout, add the SHOWFLDS(YES) parm to the ASM statement. For example, assume we had added the SHOWFLDS(YES) parm to the ASM statement on page 317:

```
ASM: SHOWFLDS(YES)
```

In that case, the following FIELD statements would have been printed 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)
```

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 96 A report produced using an Assembler record layout

Handling Date and Time Fields

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 314. The file actually contains a YYMMDD date for this field. But it is defined in Cobol simply as PIC 9(6). Report 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 would have. 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 file actually contains a HHMMSS time value for this field. But since it is defined in Cobol as PIC 9(6), it's just another numeric field to Report 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 on page 320. 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 know what column the field actually starts in. It starts in whatever column the SALES–DATE field starts in. 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 true 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 on page 320 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. Report Writer would treat this field like any other 5–digit packed number. But you could create a true Report 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 (stored in the 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 the Cobol record layout. Such a member would include everything you see in the top box on page 320 before 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 normal Report 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.

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 REGION           PIC X(5).
             05 PRODUCT-CODE     PIC X(3).
             05 AMOUNT           PIC 9999V99.
             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 BACKUP-EMPL-NUM  PIC X(3).
             05 TAX              PIC 99V99.
             05 FILLER           PIC X(5).
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 97 Creating true date and time fields from a Cobol layout

How Report Writer Handles Arrays

Report Writer requires that each field have a unique name. You can not define a field as being an "array" to Report Writer. Therefore, when Report Writer encounters a Cobol field with an OCCURS clause, it creates a separate field for each occurrence of the item. Report 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).
```

Report 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 address line in your report, you would specify:

```
COLUMNS: ADDR–LINE–2
```

Report 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
```

Report 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)
```

Report Writer also supports nested arrays in Cobol. Report 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).
```

Report 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 Report Writer has assigned, use the SHOWFLDS(YES) parm on your COBOL or ASM statement That way you will see a complete listing of the internal FIELD statements that Report Writer has created from your record layout.

> **Note:** by default, Report 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 the COBOL or ASM statement. (See page 434.) 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, Report Writer creates fields for the maximum possible number of occurrences (subject to the MAXOCCURS limit just described.)

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 Report 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 Report Writer before.

However, for input files that will be used often with Report Writer, it may be better to create a standard Report Writer file definition (consisting of a FILE statement and many 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 Report Writer to perform a one–time conversion of your Cobol or Assembler layout into FIELD statements. Report 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 Report 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 Report Writer write those same FIELD statements to a separate output file. **Figure 98** (page 324) shows an example of converting a Cobol record layout to FIELD statements. (That example assumes an MVS operating system.)

Let's examine the control statements in the top box on page 324. Once again, a FILE statement is required because fields must always belong to a particular file. Report Writer won't process record layouts or FIELD statements unless it has a file it can associate the 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 name you like.

The COBOL statement tells Report 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 Report Writer the name of a DD statement in the JCL where the FIELD statements should be written. In this example, we told Report 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 435. 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.

Report 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 use an INPUT statement or any other Report Writer statements. Report Writer writes the FIELD statements to the output file, and then ends execution. (You will see a message saying that no report was produced because no INPUT statement was found. That is normal.) The middle box on page 324 shows the FIELD statements produced by Report 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 on page 324 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 TYPE(NUM) to TYPE(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 will also add an appropriate FILE statement ahead of the FIELD statements. When you're satisfied with your file definition, save it in your Report 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 Report Writer file definition for your file.

These control statements:

```
FILE: DUMMY
COBOL: OUTDDN(FLDOUT)
       01    SALES–REC.
             05 EMPL–NAME        PIC X(10).
             05 EMPL–NUM         PIC X(3).
             05 REGION           PIC X(5).
             05 PRODUCT–CODE     PIC X(3).
             05 AMOUNT           PIC 9999V99.
             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 BACKUP–EMPL–NUM  PIC X(3).
             05 TAX              PIC 99V99.
             05 FILLER           PIC X(5).
```

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: REGION          LEN(5)
FIELD: PRODUCT-CODE    LEN(3
FIELD: AMOUNT          LEN(6)  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: BACKUP-EMPL-NUM LEN(3)
FIELD: TAX             LEN(4)  TYPE(NUM)     DEC(2)
FIELD: FILLER#001      LEN(5)
```

(File definition after sample 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: REGION          LEN(5)
FIELD: PRODUCT-CODE    LEN(3)
FIELD: AMOUNT          LEN(6)  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: BACKUP-EMPL-NUM LEN(3)
FIELD: TAX             LEN(4)  TYPE(NUM)     DEC(2)
FIELD: FILLER#001      LEN(5)
```

Figure 98 Converting a Cobol record layout to Report Writer FIELD statements

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.

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 within the Report Writer control statements. But normally Cobol and Assembler record layouts are stored as members in copy libraries, to be used by their respective compilers. Report Writer also allows you to copy such record layouts directly from those libraries. Just use Report 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. What library is searched for the member named SALEREC?

Under MVS, COPY statements normally read members from the Report 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 Report Writer definitions (and even in different libraries from each other.) Therefore, when processing Cobol code, Report Writer performs copies from the PDS named by the COBLIB DD in the JCL, if one is present. When processing Assembler record layouts, copies are performed from the library pointed to by the ASMLIB DD, if one is present. If the appropriate DD (COBLIB or ASMLIB) is not present, Report Writer attempts to perform the copy from the standard copy library (SWCOPY DD.) You can also override these defaults and specify any DDNAME you like directly in the COPY statement. Use the PDSDDN parm:

```
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.

In MVS, 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 job step ahead of the Report Writer step to write the desired member to a sequential dataset. Then have Report Writer copy that sequential dataset by using the DDNAME parm in the COPY statement:

```
COPY: DDNAME(SALEREC)
```

The above example causes Report Writer to read in the records from the sequential file pointed to by the SALEREC DD in the JCL.

VSE Note: under VSE, the COPY statement names a member to be copied from a Librarian sublibrary. It can also optionally name the member type and/or the sublibrary to use.

For example, the following COPY statements are all valid:

```
OPTION: COBLIB('PROD.COBCOPY')
COBOL:
COPY:   SALEREC
COPY:   SALEREC.COBOL
COPY:   SALEREC SUBLIB('TEST.COBCOPY')
```

The member type used will be:

- the type specified in the COPY statement itself, if any, or
- "C" (if within the scope of a COBOL statement), or
- "A" (if within the scope of an ASM statement.)

The sublibrary used will be:

- the sublibrary from the SUBLIB parm in the COPY statement itself, if any, or
- the sublibrary named in an OPTIONS: COBLIB parm, if any (if within the scope of a COBOL statement), or
- the sublibrary named in an OPTIONS: ASMLIB parm, if any (if within the scope of an ASM statement), or
- the sublibrary named in the OPTIONS: SUBLIB parm.

Note: we mentioned in an earlier section that Cobol processing begins immediately after the COBOL statement and ends when the next Report 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.

Note: you may use multiple copy statements. You may also intermix in–line code and 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
...
```

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, Report Writer assigns a file's "default location" value to the first item within a Cobol record layout. 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. If you do have preceding FIELD statements (or preceding COBOL or ASM statements), 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 first field in Assembler code is also handled in the way just described.

The "Default Location" After a Cobol or Assembler Layout

Report 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, Report 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 Report 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 Report Writer control statement, that line will end the scope of the COBOL statement.)

2. A Report 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 Report Writer is treating a particular input line as Cobol, Assembler, or native Report Writer, check the control listing. The word "COBOL" or "ASM" will appear beside each line that Report Writer is interpreting as Cobol or Assembler code. Use the LIST(YES) option on any COPY statements to ensure that the copied lines are also printed in the control listing.

Other Features Available in COBOL and ASM Statements

There are a number of parms available in the COBOL and ASM statements that we have not discussed. These parms let you control various options used in processing the record layouts. The complete syntax for the COBOL statement begins on page 432. The complete syntax for the ASM statement begins on page 419.

Technical Notes on Cobol Support

Report Writer will accept the vast majority of Cobol record layouts used in most shops. Still, Report Writer is not a complete Cobol compiler and there are some valid Cobol features that Report Writer does not support at the present.

Even when Report Writer doesn't support a particular Cobol statement, you will still save much time by using the FIELD statement output from Report 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. Report 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 Report Writer. Use the Cobol compiler for that purpose and use only clean, tested record layouts in Report Writer.

In general, if a record layout would be accepted in the Record Description entry of an FD (File Description), Report Writer will also accept it. In addition, Report Writer accepts many types of edited PICTURES (like PIC $$$,$$9.99). This means that Report 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 Report Writer.

Level Indicators

Report 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, Report 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. Report 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, Report 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, Report Writer would use DATE for the first item and DATE#001 for the second item. A message is printed in the control listing whenever Report Writer modifies a name in this way to make it unique.

Handling FILLER

Report Writer does not support FILLER as a special field name. Therefore, Report Writer *always* appends a tiebreaker to FILLER fields. No message is printed when this happens, 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

Report Writer does not process 88 level field definitions automatically. However, it is not difficult to create Report Writer equivalents for 88 items yourself. Following is an example of how several 88 level items would be defined with Report 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, Report 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 Report 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 Report 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 Report 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

Report Writer supports the SIGN IS SEPARATE clause for elemental items, but not for group items.

Technical Notes on Assembler Support

Report Writer will accept most of the Assembler record layouts used in most shops. Still, Report Writer is not a complete assembler and there are some valid Assembler features that Report Writer does not support at the present.

Even when Report Writer doesn't support a particular Assembler statement, you will still save much time by using the FIELD statement output from Report 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. Report 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 Report Writer. Use the assembler for that purpose and use only clean, tested record layouts in Report Writer.

In general, Report 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
```

Report Writer can only treat this AMOUNT field as a 6–byte character field. There is nothing to tell Report 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 Report 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
```

Report 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 Report 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 Report Writer, it may be better to create a standard Report Writer file definition for it. Use Report Writer to convert the record layout into FIELD statements. Then modify the FIELD statements as necessary to correctly define the numeric fields.

Decimal Digits

Report 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)
```

Report 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 Report 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

Report 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 Report 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

Report 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)
```

Report 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
```

Report Writer prints a warning message when an expression like the one above is encountered. You can simplify such expressions, if desired, so that Report 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 Report Writer.

Multiple Operands

Report 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 Report Writer:

```
TABLE    DS    4AL4
```

Handling EQUs

When Report 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 Report 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, Report Writer defines a 1–byte character field whose name is the DSECT name

Unique field names

Report 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, Report 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, Report Writer would use DATE for the first item and DATE#001 for the second item. A message is printed in the control listing whenever Report Writer modifies a name in this way to make it unique.

Chapter 6. Working with Databases

Chapter Table of Contents

Chapter 6. Working with Databases

At present, Report Writer supports the following databases:

- DB2

Using Report Writer with DB2 Databases

Report Writer's DB2 Option lets you use DB2 data with Report 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 Lotus 1–2–3, Excel, Access, Paradox, Harvard Graphics, and many other PC programs.

Report 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 Report 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 Report 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 Report 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 OPTION statement
- the INPUT statement
- the READ statement (not required)
- the FILE statement (not required)

For most reports and PC files, you won't even use the READ or FILE statements.

> **JCL note:** when using DB2 tables with Report 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 Report Writer to request reports and PC files. These sections explain the few differences that you need to know in order to use DB2 data in Report Writer.

The following sections show you how to:

- create a custom report from a DB2 table (page 338)
- create a PC file from a DB2 table (page 340)
- use the WHERE parm in the INPUT statement, as an alternate way to select certain rows from the DB2 table (page 342)
- use the ORDERBY parm in the INPUT statement, as an alternate way to sort your report or PC file (page 344)
- use data from multiple DB2 tables in a single run (page 345)
- customize your DB2 fields' columns heading and display format (page 352)
- create and save a Report Writer DB2 file definition (page 353)

Using DB2 Data in Reports

Let's begin by looking at an actual Report Writer report that uses DB2 data. Notice the sample report in **Figure 99**. 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 Report 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 Report 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 "Report 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" is 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. (The sample Project table is named DSN8310.PROJ in Release 3.1 of DB2.)

The INPUT statement does two things.

- it associates an actual DB2 table with a user–friendly Report Writer "file name." (This association is not permanent— it lasts only during the one Report Writer run.)
- it makes that DB2 table the primary input for your Report 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 INPUT statement parms. (The complete syntax for the INPUT statement appears on page 485.)

> **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 Report Writer statements in any way you like. Refer to the DB2 fields by using their standard, unqualified DB2 names. Report Writer will automatically recognize these DB2 names. For example, in the COLUMNS statement in **Figure 99**, we referred to the following DB2 fields from the project table: PROJNO, PROJNAME, DEPTNO, RESPEMP, PRSTDATE and PRSTAFF.

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 99 A Report Writer DB2 report

You can also use the DB2 fields in the SORT statement, COMPUTE statements, INCLUDEIF statements, BREAK statements, and all the other Report 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 Report Writer! Here's a review of the differences from non–DB2 Report Writer requests:

- no data definition of your DB2 file is necessary (that is, no FILE or FIELD statements are required)
- no Report Writer Copy Library is required
- use an OPTION statement with the DB2SUBSYS parm
- use the DB2NAME parm in your INPUT statement

Note: Report Writer supports character, numeric, date and time fields from DB2 tables. DB2 "timestamps" are treated as 26–byte character fields by Report 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 Report Writer. It's just as easy to turn your DB2 data into PC files with Report Writer. Simply add the appropriate PC option to the OPTION statement. An example of using DB2 data in a Lotus 1–2–3 spreadsheet is shown in **Figure 100**. This example shows the same "project table" data being used 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. Report 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)
```

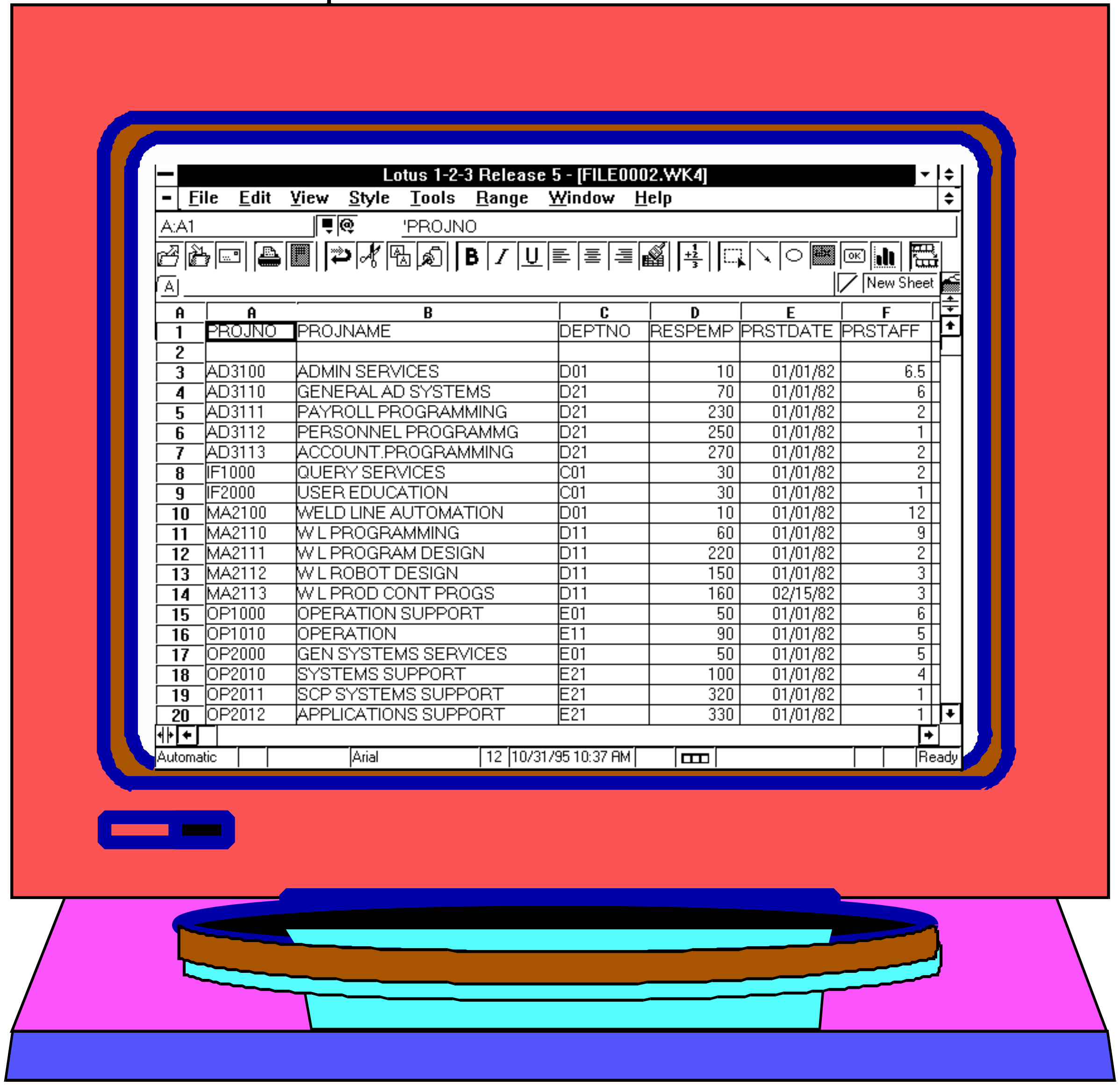
The above statement causes a list to be printed showing each DB2 field available from the DSN8230.PROJ table. This list appears in the Report Writer control statement listing. The list also indicates the data type (character, numeric, date or time) of each of the DB2 fields.

The SHOWFLDS parm can also be used in the READ statement.

These control statements:

```
OPTION:  LOTUS    DB2SUBSYS('DB2T')
INPUT:   PROJECT  DB2NAME('DSN8230.PROJ')
COLUMNS: PROJNO
         PROJNAME
         DEPTNO
         RESPEMP
         PRSTDATE
         PRSTAFF
```

Result in this Lotus 1–2–3 spreadsheet:


Lotus 1-2-3 Release 5 - [FILE0002.WK4]
File Edit View Style Tools Range Window Help
A:A1
'PROJNO
New Sheet
Automatic
Arial
12
10/31/95 10:37 AM
Ready

A	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 100 Using DB2 data in a Lotus 1–2–3 spreadsheet

Using the WHERE Parm

Here's how Report Writer interacted with the DB2 subsystem in order to produce the report on page 339. Report 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, Report 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 Report Writer is to use an INCLUDEIF statement, like this:

```
INCLUDEIF: DEPTNO = 'D21'
```

And that method works just fine! If you use this statement, Report Writer would again fetch *all* rows from the DB2 table. Report 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 Report 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, Report 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 Report 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 Report Writer to include all the rows that DB2 returned to it.

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 on page 343 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, such as what percentage of records will be 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 Report Writer all rows that meet the WHERE conditions. Of those rows, Report Writer will then include in the report only the ones that meet the INCLUDEIF statement conditions.

The WHERE parm is discussed in more detail under "WHERE Parm Syntax" on page 350.

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
```

Notes:

- we could have achieved the same result by leaving out the WHERE and ORDERBY parms, and adding these statements:

```
INCLUDEIF: DEPTNO = 'D21'
SORT:      PROJNAME
```

Figure 101 Using the WHERE parm to select certain rows from a DB2 table

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 Report Writer serves the same function as the ORDER BY clause in a DB2 "SELECT" statement. It tells DB2 what order to pass us the rows in. If your INPUT statement contains an ORDERBY parm, Report 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 Report Writer the rows in an "arbitrary" order.)

Use this parm if you want DB2 to pass its rows to Report Writer in a certain order. You may wish to use this parm rather than using a SORT statement. When no SORT statement is used, Report Writer outputs the data in the same order that DB2 passes it to Report Writer in.

The example on page 343 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 Report 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 Report 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:** 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 Report Writer in the order specified in the ORDERBY parm. Report 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." Report Writer always reads this primary input file sequentially. Then, each time a record is read from the primary file, Report Writer reads one additional record from each VSAM file named in a READ statement. The READKEY parm (in the READ statement) tells Report 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. Instead of using a READKEY parm, use the WHERE parm to identify which row you want to read. (The WHERE parm was introduced in "Using the WHERE Parm " on page 342. Its syntax is discussed in "WHERE Parm Syntax" on page 350.)

Let's start with the DB2 report on page 339 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 could 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 Report 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 Report Writer will process the above statements. The primary input to the report is the project DB2 table. So, Report Writer will retrieve all rows from the DB2 project table. For each row from the project table, Report 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 *any DB2 field* in either the project or 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 Report Writer's most powerful features. All it takes is a single READ statement.

The report in **Figure 102** (page 347) 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 Report 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 Report 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, Report Writer always uses the *first* row returned by DB2.

Note: if you want to use *all of the rows* that meet the WHEN parm conditions, add the MULTI parm to your READ statement. When the READ statement has the MULTI parm, Report 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 the Notes section of the READ statement in Chapter 9, "Control Statement Syntax" (page 520.)

Additional information on the ORDERBY parm appears under "Using the ORDERBY Parm" on page 344.

Note: the complete READ statement syntax is shown on page 510.

Note: for simplicity's sake, in this discussion we implied that Report Writer *always* reads a row from each READ file. In some cases, Report 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.

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 102 A report that uses data from 2 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. Report 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 on page 347. 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 last READ statement above. 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 any 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 the section beginning on page 350.)

The report in **Figure 103** 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 103 A report that uses data from 3 different DB2 tables

WHERE Parm Syntax

The syntax allowed within the WHERE parm is close 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:** Report Writer allows you to prefix any field name in the WHERE parm with a Report Writer record name (to eliminate possible ambiguity)
- **Date and Time Literals:** you may use either Report 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"**
- a **literal value**

Report 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 Report 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. Report 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, Report 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 232. 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 will be the same as the file name.) To tell Report 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 Report 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:** you may wonder if this Report Writer prefix will confuse DB2. The answer is no. Because when you do use a record name prefix in the WHERE parm, Report Writer removes it before passing the WHERE parm on to DB2.

> **Note:** don't confuse Report Writer's record name prefix with a DB2 qualifier. DB2 qualifiers are not necessary and are *not allowed* within Report 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." Report 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, Report 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 Report 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 on page 349. 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 Report 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, Report 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. Report 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 Report 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 Report 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, Report Writer's passes them in the WHERE clause to DB2 unchanged. When you use a Report Writer literal, Report 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 Report 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 we have shown, no FILE or FIELD statements are needed to define the fields in a DB2 input file. Report 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 Report 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 Report Writer Copy Library. (See the section beginning on page 301 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, Report 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 Report Writer without necessarily even knowing it comes from a DB2 table.

DB2 Restrictions

DB2 has certain restrictions which Report 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. (Report Writer allows a precision of 31 digits.) This means, for example, that any Report 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 contain decimal digits, the allowed range of values is reduced even further.

Chapter 7. Operating System Considerations

Chapter Table of Contents

Chapter 7. Operating System Considerations

This chapter discusses various topics that are related to the specific operating system under which Report Writer is executed. It is intended primarily for programmers who are setting up the job control language (JCL) needed to run Report Writer jobs.

The following operating systems are discussed:

- MVS (page 357)
- VSE (page 370)

MVS Operating System Considerations

The following sections discuss operating environment considerations for executing **Report Writer MVS**. Report Writer MVS runs under all MVS systems, including MVS/SP, MVS/XA, MVS/ESA and OS/390. The following topics are presented:

- sample execution JCL for custom reports (page 358)
- sample execution JCL for output files, including PC files and mainframe files (page 360)
- sample Report Writer PROC (page 362)
- specifying the access method and LRECL to use for Report Writer's output records (page 362)
- setting up file definitions in a Copy Library (page 364)
- the Copy Library DD statement (page 366)
- the input file DD statements (page 367)
- the DD statement available for start–up options (page 368)
- the jobstep completion codes (page 369)

Execution JCL for Reports –– MVS

This section explains:

- the JCL needed to produce Report Writer reports

Chapter 2, "How to Request a Report" explained how to use Report Writer's control statements to request custom reports. The JCL needed to produce such a report is very simple. **Figure 104** shows sample JCL for producing a Report Writer report.

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 Report Writer control statements (SYSIN) will be different in each run.

Here is a description of the DD statements used by Report Writer MVS.

DDNAME	REQUIRED?	USED FOR
SYSIN	Yes	Control statements describing the desired report or PC file
SWLIST	Yes	Report Writer writes the control statement listing, error messages, and end–of–job statistics here.
SWOUTPUT	Yes	Report Writer writes the actual report or PC file here.
SWCOPY	No	Points to the Report Writer Copy Library
SWOPTION	No	Used for installation–wide options. Points to a dataset containing Report Writer control statements.
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.)
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.

This JCL:

```
//SPECTWTR JOB 'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,    PRODUCE REPORT 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
```

Note:

- the Report Writer control statements in SPECTWTR.COPYLIB(SALES) would automatically be processed by Report Writer during this run. Appendix F, "Sample File Definitions" 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 104 Sample Report Writer JCL for reports — MVS

Execution JCL for PC and Mainframe Files –– MVS

This section explains:

- the JCL needed to produce Report Writer output files, including PC files and mainframe files

Chapter 3, "How to Request a PC File" explained how to use Report Writer's control statements to request PC files. Chapter 4, "Beyond the Basics" included a section on creating mainframe output files.

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 a subsequent mainframe program.

Figure 105 shows sample JCL for writing a PC file to disk.

You may specify any LRECL (and corresponding BLKSIZE) that you want in the SWOUTPUT DD. Pick a record length that will be big enough to hold all of the columns you will be writing 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" on page 362.

This JCL:

```
//SPECTWTR JOB  'REQUESTER'
//*
//SPECTWTR EXEC  PGM=SPECTWTR,    PRODUCE REPORT 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.LOTUS.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: LOTUS
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
//
```

Produces this PC 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
```

Note:

- only the SWOUTPUT DD statement is different from the JCL used to produce a report (page 359.)

Figure 105 Sample Report Writer JCL for PC files — MVS

Report Writer PROC –– MVS

You may wish to create a PROC for Report Writer. That makes it much easier to set up new Report Writer jobstreams. A PROC also makes it easier for non–technical users to run Report Writer jobs. Here is a an example of how such a PROC might look:

```
//SPECTWTR PROC COPYLIB='NULLFILE'
//SPECTWTR EXEC PGM=SPECTWTR,                    PRODUCE REPORT 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 –– MVS

This section explains:

- the default **access method** Report 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, Report 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 several hundred bytes long in order to hold all the desired data.

Report Writer supports output records up to approximately 16,000 bytes wide. Here is how Report Writer determines what record length (LRECL) to use in a particular run.

When writing QSAM output (Report 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, Report 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, Report 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 Report 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 Report Writer to allow up to 200 characters in the report (again reserving 1 byte for the carriage control character.) Report 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, Report 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, Report 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=LOTUS.FILE,DISP=(NEW,CATLG),
//            DCB=(LRECL=300,BLKSIZE=3000,RECFM=FB),
//            SPACE=(CYL,5),UNIT=SYSDA
```

In the above example, Report Writer would only truncate fields that extended beyond column 300.

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, Report Writer will use that LRECL as the size of its output records. If no OUTLRECL option is specified, Report Writer again defaults to writing 133–byte records. However, if the VSAM dataset was defined with a maximum record size less than 133, then Report Writer defaults to writing records the size of the maximum record size defined for the file.

Setting Up File Definitions –– MVS

Before running Report Writer, some one–time setup is required. This setup consists of creating a Report Writer Copy Library PDS, and then storing descriptions of your company's files in it. This is necessary before Report Writer can produce reports or PC files from your company's data.

The following setup steps are needed:

Step 1.
Allocate a new PDS to be used as your Report 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 Report Writer. (A Report 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 Report Writer Copy Library.

Step 2.
Create a new member in the copy library for the first file that you want to define to Report Writer. For example, if you want to define your company's payroll file, you might create a new member named PAYROLL. Within this member, type 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 Report 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" on page 269.

Next, type 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 275.) 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 Report Writer, you do not need to define those fields. Just use the COLUMN parm where needed in subsequent FIELD statements to tell Report Writer exactly which column a field begins in.

When you are finished, the copy library member should contain a single FILE statement, followed by a number of FIELD statements. (Appendix F, "Sample File Definitions" shows some examples of copy library members and their file definition statements.) 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 Report 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 Chapter 5, "How to Define Your Input Files" beginning on page 311. To begin with, though, we suggest you define one or two small files manually (as described above) to get a clear idea of how Report Writer works. That will make it easier for you to later see how Report Writer's Cobol and Assembler interpreter fits into the picture.

Step 3.
Add an alias entry for your file. 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 that example.

The purpose of an alias is to relate the Report 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 definition is stored. When the two names are the same, no alias is needed. But you can also use longer, more user–friendly file names if you like. You'll just need to add an alias entry to a 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), and just add an alias entry like this within SWALIAS:

```
HEADQUARTERS–PAYROLL = HQPAYROL
```

The above line tells Report 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 Report 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 Report Writer as the member name for reading the definition statements from the copy library. Appendix F, "Sample File Definitions" 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 Report Writer.

Step 5.
In your execution JCL, make sure the SWCOPY DD points to the copy library that you just set up (containing each file's definition statements, and containing the SWALIAS member.)

Your Report 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 –– MVS

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 Report 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 –– MVS

This section explains:

- how to write the DD statement(s) for the input file(s)

In order for Report Writer to produce a report (or PC or mainframe 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), Report Writer must also open and read from these files.

Make sure that the JCL used to run a Report Writer report contains one DD statement for each input file used in the report.

How does Report 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 Report 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 Report Writer Copy Library.)

An *override* DDNAME parm can also be specified directly in the INPUT or READ statement. When this happens, Report Writer uses the override DDNAME, rather than the one from the FILE statement.

Speed–Up Tip: Random reads to VSAM files can be relatively slow. VSAM maintains two types of buffers (data and index) while processing Report Writer's requests. 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 auxiliary 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, using the `AMP=('AMORG,BUFNI=nn,BUFND=nn')` parm in the DD statement, 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 609.

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 Report Writer is reading that file, there is a possibility that Report 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 Report 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 Report 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 Report Writer) that will read that file.

Specifying Shop–Wide Options –– MVS

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 Report 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 Report 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, Report 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 –– MVS

Upon completion, Report Writer exits back to the operating system with one of the following completion codes:

COMPLETION CODE	MEANING
0	No errors or warning messages issued. Report Writer produced its output normally. (Some informatory messages may have been printed.)
4	Only warning messages were issued. Report Writer produced its output as well as it could.
12	Error messages were issued. No output (or only a partial output) was produced.
16	Security error. Report Writer has expired or some other error was detected in the authorization codes. No output was produced.

VSE Operating System Considerations

The following sections discuss the JCL needed to execute **Report Writer VSE**. Report Writer VSE runs under DOS/VSE, VSE/SP and VSE/ESA. The following topics are presented:

- sample execution JCL for custom reports (page 370)
- sample execution JCL for output files, including PC files and mainframe files (page 372)
- specifying the type and record size of the output file (page 374)
- various methods of downloading PC files (page 375)
- setting up file definitions in a Copy Library (page 376)
- the DLBL/TLBL statements required for input files (page 379)
- routing the control statement listing (page 380)
- specifying the SIZE parm in the EXEC JCL statement (page 380)
- using sort work files (page 381)
- the jobstep completion codes (page 381)

Execution JCL for Reports –– VSE

This section explains:

- the JCL needed to produce Report Writer reports

Chapter 2, "How to Request a Report" explained how to use Report Writer's control statements to request custom reports. The JCL needed to produce such a report is very simple. **Figure 106** shows sample JCL for producing a Report 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 Report Writer control statements (SYSIPT) will be different in each run.

Here is a list of the logical unit assignments used by Report Writer:

SYSIPT	the Report Writer control statements are read from SYSIPT
SYS010	a "control listing" is written to this logical unit. It includes a listing of your Report 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 (see page 374.)

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'
// 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
```

Note:

- the Report Writer control statements in member SALES.SPECTWTR of LIB.SPECTWTR would automatically be processed by Report Writer during this run. Appendix F, "Sample File Definitions" 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 106 Sample Report Writer JCL for reports — VSE

Execution JCL for PC and Mainframe Files –– VSE

This section explains:

- the JCL needed to produce Report Writer output files, including PC files and mainframe files

Chapter 3, "How to Request a PC File" explained how to use Report Writer's control statements to request PC files. Chapter 4, "Beyond the Basics" included a section on creating mainframe output files.

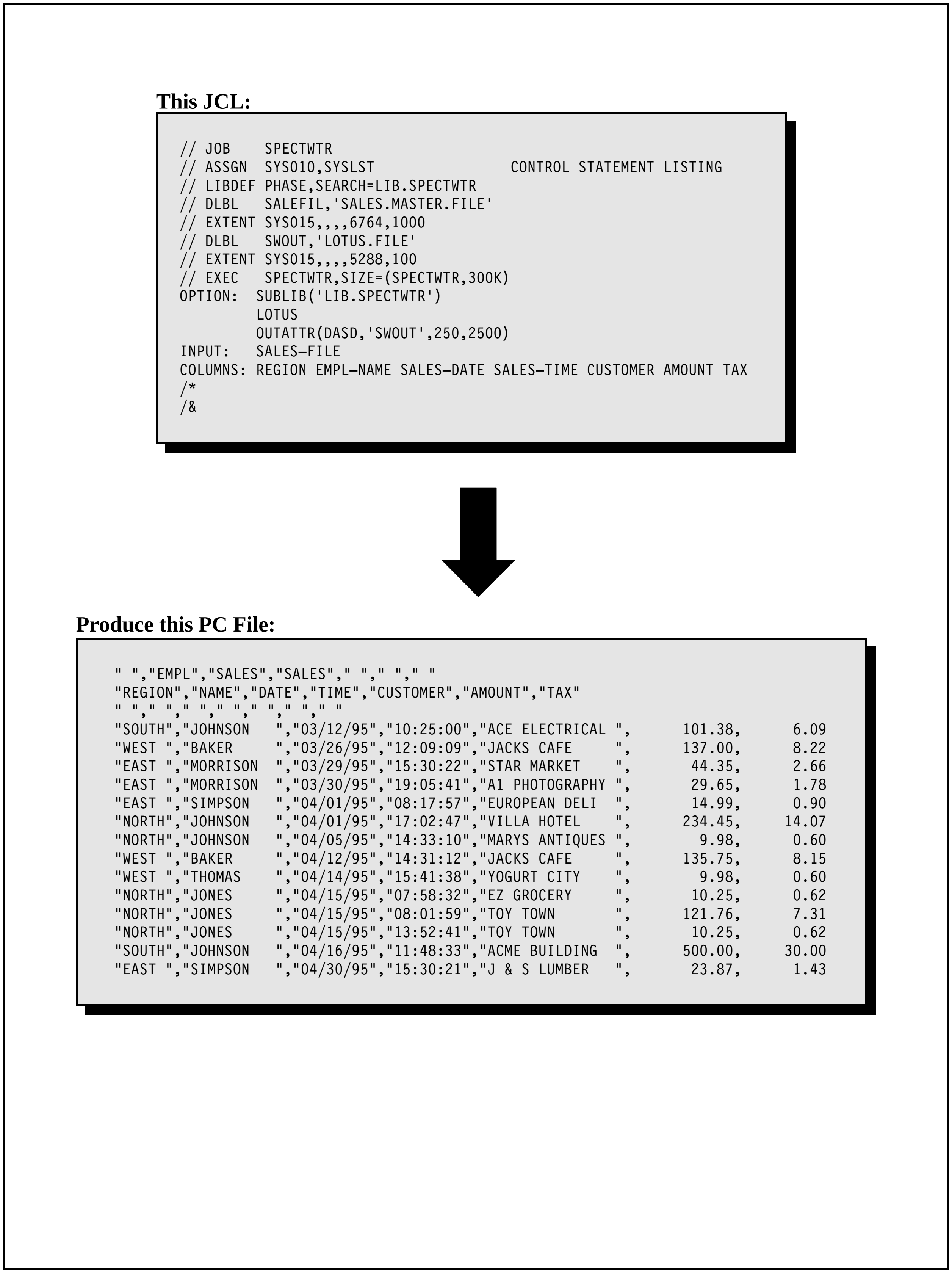
The only JCL difference when creating PC (or mainframe) files concerns where the output will be written. By default, Report Writer writes output file records to the "printer" at SYS011 (just as it writes report lines.)

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 371.)

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 a subsequent mainframe job.

Figure 107 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 Report 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 beginning on page 374.



This JCL:

```
// JOB    SPECTWTR
// ASSGN  SYS010,SYSLST                CONTROL STATEMENT LISTING
// LIBDEF PHASE,SEARCH=LIB.SPECTWTR
// DLBL   SALEFIL,'SALES.MASTER.FILE'
// EXTENT SYS015,,,,6764,1000
// DLBL   SWOUT,'LOTUS.FILE'
// EXTENT SYS015,,,,5288,100
// EXEC   SPECTWTR,SIZE=(SPECTWTR,300K)
OPTION:  SUBLIB('LIB.SPECTWTR')
         LOTUS
         OUTATTR(DASD,'SWOUT',250,2500)
INPUT:   SALES-FILE
COLUMNS: REGION EMPL-NAME SALES-DATE SALES-TIME CUSTOMER AMOUNT TAX
/*
/&
```

Produce this PC 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 107 Sample Report Writer JCL for PC files — VSE

Output File Options –– VSE

This section explains:

- the default **access method** Report 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 Attribute") option lets you give Report Writer explicit information about how and where to write its output. If no OUTATTR option is specified, Report 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 Report 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 Report 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). For reports, you will probably use 133, since that is the maximum size most printers support. When creating output files, you can specify any record size that is big enough to hold all the data you plan to write.

The figure on page 373 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 Report 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 corresponding 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, Report Writer assumes a default record size of 133 bytes.

You can also use the OUTATTR option to have Report 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 Report 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 Report Writer's output to a tape file:

```
OPTIONS: OUTATTR(TAPE,'OUTFILE',SYS009,200,12000)
```

The above statement tells Report 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 at page 494.

Downloading PC Files –– VSE

After Report Writer creates your PC file on the mainframe, just download it to your PC and import it into your favorite PC program. Appendix H, "How to Import PC Files" shows the commands used to import PC files for many popular PC programs.

You can use whatever download method you're most familiar with. Here are some of the common methods of downloading datasets from VSE to a PC.

Downloading from a POWER output queue.
Use Report Writer's default to write your PC file as if it's going to a printer. In your JECL, choose a POWER class that allows the output to remain in the queue (rather than being printed right away.) Then, use VSE's Interactive User Interface (IUI) menus to download the POWER queue entry to your PC. By default, Report Writer limits your output records to 133 byte. Use the OUTATTR option to specify a larger record size if 133 is not big enough to hold all the columns you intend to download. For example, to create a 200–byte entry in the POWER output queue, specify:

```
OPTIONS: OUTATTR(PRT,SYS011,200)
```

Downloading from a CICS VSAM file
If you prefer, you can download a CICS VSAM file. Use the following option to have Report Writer write its PC file to a VSAM file:

```
OPTIONS: OUTATTR(VSAM,'SWOUT',200)
```

You will need to define the dataset (as an ESDS dataset) ahead of time. That dataset will also need to be defined to CICS (via an FCT.) Then, use IUI to copy the contents of your VSAM dataset to the Host Transfer File. You can then download it to your PC from the Host Transfer File.

Downloading under VM
If you are running VSE under VM, you may prefer to do the download from VM (CMS). Use your 3270 emulator package's file transfer command (probably RECEIVE) to do this. You have a couple of options as far as getting the PC file from VSE to your VM session. You can have Report Writer create a printer output file, which you would then spool to your VM session as a reader file. You can then read your reader file into a CMS dataset and download from there. Or, you could have Report Writer write to a SAM disk file on a VSE pack. Then, link from VM to the VSE pack containing your PC file, and download that dataset to your PC.

Using Third–Party Products
Your shop may also have a third–party product that makes it easy to download mainframe files to PCs. Products that have been mentioned to us by users include: BIM–PC/TRANSFER, pcMainframe, PC–Link, and Outbound.

Setting Up File Definitions –– VSE

This section explains:

- how to set up a Librarian sublibrary to serve as the **Report Writer Copy Library**
- how to use the **SUBLIB option** to tell Report Writer the name of the copy library

Before you run Report Writer using your own files, some one–time setup is required. This setup consists of storing descriptions of your company's files in the Report Writer Copy Library. This is necessary before Report Writer can produce reports or PC files from your company's data.

The following setup steps are needed to define your company's files to Report Writer:

Step 1.
Pick a Librarian sublibrary to use as your Report 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 Report Writer Copy Library.

Some shops may want to use *multiple* copy libraries with Report Writer. (Perhaps one for each department in the company.) It is fine to do that. You will tell Report 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 Report 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 Report 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 273.)

Next, the member should contain one FIELD statement for each field in the payroll file. (For more information on writing FIELD statements, see page 275.) 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 Report Writer, you do not need to define those fields. Just use the COLUMN parm where needed in subsequent FIELD statements to tell Report Writer exactly which column a field begins in.

When you are finished, the copy library member should contain a single FILE statement, followed by a number of FIELD statements. (Appendix F, "Sample File Definitions" 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 Report 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 Chapter 5, "How to Define Your Input Files" (page 311.) To begin with, though, we suggest you define one or two small files manually (as described above) to get a clear idea of how Report Writer works. That will make it easier for you to later see how Report Writer's Cobol and Assembler interpreter fits into the picture.

Step 3.
Add an alias entry for your file. 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.

The purpose of an alias is to relate the Report 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 definition is stored. When the two names are the same, no alias is needed. But you can also use longer, more user–friendly file names if you like. You'll just need to add an alias entry to a special member named SWALIAS.SPECTWTR 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), and just add an alias entry like this within SWALIAS.SPECTWTR:

```
HEADQUARTERS–PAYROLL = HQPAYROL
```

The above line tells Report 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 Report 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 Report Writer as the member name for reading the definition statements from the copy library. Appendix F, "Sample File Definitions" 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.

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 Report Writer.

Step 5.
In your Report Writer control statements, always begin with an OPTIONS: SUBLIB statement. This will tell Report Writer the name of the 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 Report 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.

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 Report 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 the READ statement), Report Writer must also open and read from those files.

How does Report 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 Report Writer with a FILE statement. The ATTR parm in the FILE statement specifies which DLBL (or TLBL) Report Writer should use when reading the file. The ATTR parm also tells Report Writer other important information about the file, such as its record size and block size.

Note: The FILE statement is normally kept in the Report Writer Copy Library.

Note: The syntax of the FILE statement is shown on page 470.

An *override* ATTR parm can also be specified directly in the INPUT or READ statement. When this happens, Report Writer uses the override DLBL (or TLBL) name, rather than the one from the FILE statement.

Make sure that your Report 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.)

Speed–Up Tip: Random reads to VSAM files can be relatively slow. VSAM maintains two types of buffers (data and index) while processing Report Writer's requests. 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 auxiliary 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 609.

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 Report Writer is reading that file, there is a possibility that Report 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 Report 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 Report 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 Report Writer) that will read that file.

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

Report 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.

Report 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 Report Writer phase itself (about 250K)
- the Librarian program (if you will be using Report 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 Report Writer jobs will involve an internal sort. This is required in order to put your report or output file into the order specified by the SORT control statement. Report 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 Report 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)
```

Completion Codes –– VSE

Report Writer exits back to the operating system with one of the following completion codes:

COMPLETION CODE	**MEANING**
0	No errors or warning messages issued. Report Writer produced its output normally. (Some informatory messages may have been printed.)
4	Only warning messages were issued. Report Writer produced its output as well as it could.
12	Error messages were issued. No output (or only a partial output) was produced.
16	Security error. Report Writer has expired or some other error was detected in the authorization codes. No output was produced.

(This page left blank intentionally.)

Part 2. Reference Manual

Chapter 8. General Syntax Rules

Chapter Table of Contents

Chapter 8. General Syntax Rules

This chapter describes the general syntax rules that apply to all control statements. The following topics are covered:

- the **overall format** of control statements
- how to continue a long control statement onto **multiple lines**
- how to include **comments** among the control statements
- how to force a **page break** in the control statement listing
- the rules governing **names** used for files, fields, and records
- how to write **literal values** (for character, numeric, date and time data)
- the rules governing the **PICTURE display format**
- the rules for writing **conditional expressions**
- the rules for writing **computational expressions**

Control Statements

What Is a Control Statement?

Control statements are the means by which you describe a desired report or PC file to Report 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, Report 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 Report 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 given in Chapter 9, "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, Report 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 Report 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. Report 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 Report 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 Report Writer but provide good documentation to other people looking at your control statements. There are two ways to include comments in your control statements.

- use an entire **comment line**, by putting an asterisk (*) in column 1 of the line
- 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 Report 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 Report 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:

- all characters 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 (about 70 characters.) Names may not be split across lines.

Note: it is recommended that you do not name your fields beginning with the pound sign (#). This is to avoid confusion with Report Writer's built–in fields and functions, which all begin with a pound sign. For example, Report 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. 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 under "How to Name the Input File Records" on page 232.

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 too long. Thirty to forty characters long is probably a good maximum length for field names.

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 Report Writer's control statements.

There are five types of literals, corresponding to the five types of data recognized by Report 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 Report 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

Report Writer knows what kind of data exists in a particular field from the TYPE parm specified in its FIELD statement. Report 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 generally be of the same type. Also, the operations allowed will depend on the data type of the operands.
- in print expressions, the display format parms used 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 385 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. Report 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. Leading zeros in the month and day are optional. For the year, you may use either all four digits, or just the last two digits. By default, date literals with 2–digit years are assumed to be in the 20th century (1900–1999). However, you may use the CENTURY option (page 497) to specify a cutover year which will allow you to use YY–type dates for both the 20th and 21st centuries. Date literals must specify a date between January 1, 1901 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    =  7/4/92
INCLUDEIF: HIRE–DATE   <  2/01/97
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 are 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 HH:MM:SS format. A leading zero in the hour portion of the time is optional. Time literals 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     =    8:30:00
COMPUTE:   END–DATE       =   17:00:00
INCLUDEIF: SALES–TIME     >=  12:00:00  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 are written.

Bit Literals

There are no true bit literals in Report 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 399.)

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 (described in Appendix F, "Sample File Definitions".) 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 the section beginning on page 282.

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

Examples of PICTUREs

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.

	FORMATTED	**FORMATTED**
PICTURE	**POSITIVE VALUE**	**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`
`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.

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'))
```

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 *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.

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.)

Any literal that appears *after* all zero suppression symbols in a PICTURE will always be moved to the result. This also includes all literals in PICTUREs where no zero suppression symbols are used (such as `PIC'(999) 999–9999'`). This also means that *all trailing literals* are always moved to the result. Trailing literals 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%'`). (As described earlier, trailing plus or minus signs also have special meanings.)

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.)* 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.)* 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.)* 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.)* 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.

(continued on next page)

MEANING OF SYMBOLS WITHIN A PICTURE (CONTINUED)

SYMBOL	MEANING
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.

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 are always formatted into the result.

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, Report 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 the 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, "Sample File Definitions."

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 405.)

The relation **operator** may be any of the following:

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 403.)

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 400 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–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, Report 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, Report Writer is actually comparing the LAST–NAME field with a 15–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 469.) 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. Report Writer automatically handles any data type conversion that may be 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.")

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:

```
INCLUDEIF:   #MAKENUM(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):

```
INCLUDEIF:   #MAKENUM(TIME–ON–PHONE) > 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. Report 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 Report Writer take care of the details necessary in making 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 a little 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:

```
INCLUDEIF: HIRE–DATE = X'404040404040'
```

The above statement compares the 6–byte HIRE–DATE field to 6 spaces (hexadecimal 40).

How to Specify a Bit Field Condition

The relation condition (described beginning on page 400) 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.

Report 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. The following table shows the result of the above conditional expression with various values for the SALES–QTR1 and the HIRE–DATE fields:

SALES–QTR1 VALUE	HIRE–DATE VALUE	CONDITIONAL EXPRESSION IS:
5000	5/1/1960	TRUE
5000	6/3/1999	FALSE
1000	5/1/1960	FALSE
1000	6/3/1999	FALSE

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 3 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. The following table shows the result of the above conditional expression for various values of the SALES–QTR1 and HIRE–DATE fields:

SALES–QTR1 VALUE	HIRE–DATE VALUE	CONDITIONAL EXPRESSION IS:
5000	5/1/1960	TRUE
5000	6/3/1999	TRUE
1000	5/1/1960	TRUE
1000	6/3/1999	FALSE

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** 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

Examples of conditional expressions are found under the INCLUDEIF statement's syntax (page 481).

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 ]  ...
```

Note: 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 as the first. 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 389.)
- 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")
- a **built–in function's result** (a complete list of built–in functions is found in Appendix D, "Built-In Functions")

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.)

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, Report Writer first converts each time value into a numeric value (which corresponding to the 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 (see page 575) to convert a date field to a numeric value. You can then perform addition, subtraction, etc. with this numeric value. Then, use the #MAKEDATE built–in function (page 579) to convert the modified numeric value back to a date field. An example of this is shown on page 414.

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 Report 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 Report 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" on page 187.

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.

(This page left blank intentionally.)

Chapter 9. Control Statement Syntax

Chapter Table of Contents

Chapter 9. Control Statement Syntax

This chapter contains the complete syntax information for each Report Writer control statement. The statements appear in alphabetical order.

Syntax Notation

In the syntax boxes throughout this chapter, the following conventions are used.

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	<u>underlined items</u> 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. Report 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 Report 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) Report Writer treats input lines as Assembler code. The Assembler code is assumed to end when the next Report Writer control statement prefix is encountered. The only exception is that Report 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 parms is found under the similar COBOL statement on page 432. 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 311

ASM

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)                                       (MVS 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 432) 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 211.

> **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 subsequent pages
- 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 52
- a lesson on using the BREAK statement in PC files begins on page 104
- advanced uses of the BREAK statement are discussed beginning on page 182

- using the BREAK statement to produce summary reports is discussed beginning on page 62
- using the BREAK statement to produce summary PC files is discussed beginning on page 110
- customizing the Grand Totals with a BREAK statement is discussed beginning on page 214

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 428.

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 (see page 214.)

EXAMPLES:

```
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.

```
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 428. The use of the AVERAGE parm is discussed on page 194.

EXAMPLES:

```
BREAK: REGION  AVERAGE
```

The above example causes a default average line to print whenever the REGION field changes value.

```
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 break–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 428. The use of the FOOTING parm is discussed on page 196.

EXAMPLES:

```
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.)

```
BREAK: REGION
       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.

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 428. The use of the HEADING parm is discussed on page 208. Specifying the REPEAT parm (in the BREAK statement) causes all of the HEADING lines to also be repeated at the top of each page of the report (following 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 428. The use of the MAXIMUM parm is discussed on page 194.

EXAMPLES:

```
BREAK: REGION  MAXIMUM
```

The above example causes a default maximum line to print whenever the REGION field changes value.

```
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 428. The use of the MINIMUM parm is discussed on page 194.

EXAMPLES:

```
BREAK: REGION  MINIMUM
```

The above example causes a default minimum line to print whenever the REGION field changes value.

```
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 428. The use of the NZAVERAGE parm is discussed on page 194.

EXAMPLES:

```
BREAK: REGION  NZAVERAGE
```

The above example causes a default non–zero average line to print whenever the REGION field changes value.

```
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 428. The use of the NZMINIMUM parm is discussed on page 194.

EXAMPLES:

```
BREAK: REGION  NZMINIMUM
```

The above example causes a default non–zero minimum line to print whenever the REGION field changes value.

```
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 the HEADING parms) should be repeated at the top of each new page (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.

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.

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 page number to "one".
NEWSHEET	Skips to a **new sheet** of paper. In order for this feature to work, you must also use the OPTION 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 starting on page 506.)
NEWSHEET1	Works like NEWSHEET, but also resets 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 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 break–wide statistics for numeric and time fields. The syntax of the print expression is shown on page 428. The use of the TOTAL parm is discussed on page 190.

EXAMPLES:

```
BREAK: REGION  TOTAL
```

The above example causes a default total line to print whenever the REGION field changes value. (Since the *default* is to print a total line, the TOTAL parm in this example was not necessary.)

```
BREAK: REGION  TOTAL('TOTALS FOR'  REGION)
```

The above example specifies that the total line should 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 196.

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[(  [   BIZ                             ]
             [   display-format                  ]
             [   LEFT/CENTER/RIGHT               ]
             [   TOTAL/AVERAGE/MAXIMUM/MINIMUM/
                 NZAVERAGE/NZMINIMUM             ]
             [   width                           ]   )]

'literal'[(      width                               )]
```

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

(Within a print–expression). 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 it.) 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 Report 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" 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 206.) These special built–in fields are:

BUILT–IN FIELD	TYPE	DESCRIPTION
#ITEMS	Numeric	Contains the number of items (records) included in the control group that has just ended.
#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.
#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".

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'

(Within a print–expression). 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

(Within a print–expression.) 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.

BIZ

(Within a print–expression.) 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

(Within a print–expression.) 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 550). If this parm is not specified, Report 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 559

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. For example:

```
END OF EMPLOYEES HIRED ON MAY 1, 1995
```

LEFT/CENTER/RIGHT

(Within a print–expression.) Specifies how the data should be justified within the space allocated for it in the print line.

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

(Within a print–expression.) *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 196.

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

(Within a print–expression.) 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 language record layout follows. Report 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 Report 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) Report Writer treats input lines as Cobol code. The Cobol code is assumed to end when the next Report Writer control statement prefix is encountered. The only exception is that Report 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 311

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)                                    (MVS 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 these same parms.

COLUMN[(ALL)]/DISP[(ALL)]

Specifies whether the COLUMN parm or the DISP parm should be used in the FIELD statements that Report 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 Report 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.

EXAMPLES:

```
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.

```
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/<u>*</u>)

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/<u>100</u>)

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 321 for a discussion 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. Report 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:** even when NOSEQ is *not* specified, Report Writer only prints warning messages for the first 5 sequence number errors encountered.

EXAMPLE:

```
COBOL: NOSEQ
```

The above statement specifies that Report 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:

type This parm is required. It tells Report 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 Report 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)

MVS 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 Report 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, Report 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 Report 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 item defined will start in the "default location" for the file it belongs to. If this is the first item defined for a file, the default location will be column 1. 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 should begin 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 PC 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 17
- a lesson on using the COLUMNS statement in PC files begins on page 78
- advanced uses of the COLUMNS statement are discussed beginning on page 123
- the use of multiple COLUMNS statements is discussed beginning on page 147

COLUMNS

SYNTAX

COLUMNS STATEMENT SYNTAX

```
COLUMNS:  print–expression
```

Note: the syntax for the print-expression is shown on page 439.

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 line in the report or PC file.

COLUMNS STATEMENT SYNTAX

```
COLUMNS:  print-expression
```

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 either a **fieldname** or a **literal text**. Each item can optionally be followed by a parm list in parentheses:

```
fieldname[(  [   ACCUM/NOACCUM                ]
             [   BIZ                          ]
             [   display-format               ]
             [   'heading1|heading2...'       ]
             [   LEFT/CENTER/RIGHT            ]
             [   NOREPEAT/NOREPEATPAGE        ]
             [   width                        ]   )]

'literal'[(  [   'heading1|heading2...'       ]
             [   width                        ]   )]
```

Standard Spelling	Alternate Spellings
ACCUM	ACC
CENTER	CJ
COLUMNS	COLUMN, COLS, COL
LEFT	LJ
NOACCUM	NOACC
NOREPEAT	NOREP
NOREPEATPAGE	NOREPPAGE
RIGHT	RJ

fieldname

Names a field that should appear as a column in the report or PC file. The field must be available to Report 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" 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.

'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.

EXAMPLES:

```
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 148 for a similar report 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 125 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 124.

> **Note:** to change the default spacing factor, use the COLSPACE parm of the OPTION statement (see page 498.)

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.

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, Report 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 do 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 144.

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.

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 have zero values.

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" on page 550. If you do not specify a display format in the COLUMNS statement, Report Writer uses a default display format. This will be:

- the display format (if any) specified when the field was defined (in a FIELD or COMPUTE statement), or
- the display format (if any) specified in a previous OPTIONS statement's FORMAT parm (see page 500.) 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 559.

PC file note: display formats should not normally be used when creating PC files. Report 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 132.

'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. Report Writer automatically centers each part of the column heading for you.

For 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 dashes will have a column heading that says "NEW TELEPHONE" on a single line.

> **Note:** you may use the HDGSEP parm of the OPTION 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. For 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, Report 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, Report 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.

See page 127 for more information on column headings.

LEFT/CENTER/RIGHT

Specifies how the data should be justified within a column. The use of these parms is discussed on page 142.

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.

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 140.

EXAMPLE:

```
COLUMNS: REGION(NOREPEAT)  EMPL–NAME  SALES–DATE  CUSTOMER  AMOUNT
```

The above example specifies that repeating values of the REGION field should be blanked out.

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.

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. The use of the width parm is discussed on page 131.

COMPUTE Statement

PURPOSE

Defines a new field that can be **computed** using one or more existing fields. 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 may be used in any way that a field from an actual file may be used. That is, you may 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 be when displaying the field
- specify whether or not the field should be **accumulated**, and thus appear in the Grand Total line, etc.
- specify the **width** 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 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 30
- a lesson on using the COMPUTE statement in PC files begins on page 92
- use of the DIVTOTS parm in the COMPUTE statement is discussed beginning on page 187
- suggestions on writing COMPUTE statements for maximum CPU efficiency are given in Appendix I, "Speed-Up Tips" (page 603)

SYNTAX

COMPUTE STATEMENT SYNTAX

```
COMPUTE:     fieldname[( parms )]  =  computational–expression

    or

COMPUTE:     fieldname[( parms )] =
             WHEN(conditional–expr)    ASSIGN(computational–expr)
        [    WHEN(conditional–expr)    ASSIGN(computational–expr)           ]
        [    WHEN(conditional–expr)    ASSIGN(computational–expr)           ]
             ...
        [    ELSE                      ASSIGN(computational–expr)/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

Note: values are assigned to computed fields each time a new logical input record is assembled. For runs which do not use MULTI–type READ statements, that means each time a new primary input file record is read. (For a discussion of logical input records, see the READ statement Notes on page 520.)

There are two forms of the COMPUTE statement. A **simple** COMPUTE statement contains a *single* computational expression. Each time a new logical input record is assembled, the specified computation is performed and a value 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 logical input record is assembled, 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 current 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. See Note 1 on page 453 for more information on Conditional COMPUTE statements.

Note 2 on page 454 discusses the data type of computed fields.

A description of the **size** of character compute fields, and of the **number of decimal digits** in numeric and time compute fields appears under the "nnn" parm (page 450.)

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 388.

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 one or more 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 410. A lesson on writing computational expressions begins on page 30.

EXAMPLE:

See the examples beginning on page 451.

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. 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, Report 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 do want to see totals for a time field. Such 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 144.

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.

```
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)

Used in the conditional form of the COMPUTE statement. Specifies how to compute a value which may 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 410. A lesson on writing computational expressions begins on page 30.

For more details on how conditional COMPUTE statements work, see Note 1 on page 453.

> **Note:** no space is allowed between the word ASSIGN and the parenthesis that follows it.

EXAMPLE:

See the examples beginning on page 451.

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" beginning on page 550.

The display–format specified in the COMPUTE statement tells Report 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. Any display format specified here can, however, always be overridden by using a different display format parm directly in a COLUMNS or TITLE statement, etc.

If this parm is not specified, Report Writer uses a default display format when printing the field in a report. Default display formats are shown in the table on page 559.

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" whenever it is printed in the 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 Report Writer to compute the field's total by performing just such a group–wide division. The use of the DIVTOTS parm is discussed beginning on page 187.

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)
  ```

- Neither the numerator nor the denominator may be literal values. Each must be either a field or an expression. That is, 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.

EXAMPLE:

See the example on page 453.

ELSE

Used in the conditional form of the COMPUTE statement. 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 has. If present, the ELSE parm and its associated ASSIGN/RETAIN parm must be the last items in the COMPUTE statement.

For more details on how conditional COMPUTE statements work, see Note 1 on page 453.

EXAMPLE:

See the examples beginning on page 451.

'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. Report Writer automatically centers each part of the column heading for you.

> **Note:** you may use the HDGSEP parm of the OPTION 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, Report Writer uses 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 a column heading parm directly in the COLUMNS statement.

See page 127 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 during the computation. The final result, as well as each intermediate result obtained during the computation, is rounded to this precision. If this parm is omitted, Report Writer chooses a default number of decimal places to keep, based on the operands and operations involved in the computational expression(s). The maximum number of digits (including decimal digits) that Report Writer maintains for numeric fields is 31.

EXAMPLES:

```
COMPUTE: AVERAGE-SALES(0) = YEARLY-SALES / 4
```

The above example specifies that the AVERAGE–SALES field should not contain any decimal digits. If the "0" parm had not been specified, some decimal digits would have been retained in the result.

```
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. If we had specified 1 decimal digit in the COMPUTE statement, the computed result would have been less precise, since each intermediate result would have been rounded down to only 1 decimal digit.

```
COMPUTE: DURATION(1) = END-TIME - START-TIME
```

The above example specifies that the DURATION field 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 451.

RETAIN

Used in the conditional form of the COMPUTE statement. 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 current value (rather than be assigned a default value.) For more details on how conditional COMPUTE statements work, see Note 1 on page 453. For a speed–up tip relating to the RETAIN parm, see page 607.

WHEN(conditional–expression)

Used in the conditional form of the COMPUTE statement. 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 subsequent ASSIGN parm is assigned to the computed field.

The syntax for conditional expressions is shown on page 399. A lesson on writing conditional expressions appears on page 24. For more details on how conditional COMPUTE statements work, see Note 1 on page 453.

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 451.

EXAMPLES

See page 413 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 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 = 'AZ')  ASSIGN('ARIZONA')
                      WHEN(STATE = 'NV')  ASSIGN('NEVADA')
                      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."

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."

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 all total lines. The DIVTOTS parm tells Report 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, Report 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.

NOTES

Note 1. Conditional COMPUTE Statements

The value assigned to the result field is determined by evaluating 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. 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 (and none of the WHEN expressions was true) the compute field will retain the value it had for the previous logical 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:

FIELD TYPE	DEFAULT VALUE
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

Note 2. Data Type of the COMPUTE Field

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.

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 Report Writer Copy Library is discussed beginning on page 301
- the MVS JCL aspects of the copy library are discussed beginning on page 366
- the VSE JCL aspects of the copy library are discussed beginning on page 376
- the use of the COPY statement in conjunction with Cobol and Assembler record layouts is discussed on page 325

COPY

SYNTAX

COPY STATEMENT SYNTAX

```
COPY:        copyname/DDNAME(ddname)
        [    LIST(YES/NO)                 ]
        [    NOTALIAS                     ]
        [    PDSDDN(ddname)               ]
        [    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

Identifies a member to be copied from a PDS (MVS) or from a Librarian sublibrary (VSE). If present, copyname 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 (VSE only).
  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 in member names.

For MVS, the member will be copied from a PDS identified by a DD statement in the JCL. Report Writer will use the DD statement whose DDNAME is:

- the one named in the PDSDDN parm of the COPY statement, if any, or
- "COBLIB", if within the scope of a COBOL statement, or
- "ASMLIB", if within the scope of an ASM statement, or
- "SWCOPY" otherwise

For VSE, the member will be copied from a Librarian sublibrary. Report Writer will use the sublibrary whose name is:

- the one named in the SUBLIB parm of the COPY statement, if any, or

- the one named in an OPTIONS statement COBLIB parm, if within the scope of a COBOL statement and if such a COBLIB parm was found, or
- the one named in an OPTIONS statement ASMLIB parm, if within the scope of an ASM statement and if such an ASMLIB parm was found, or
- the one named in an OPTIONS statement SUBLIB parm

(Under VSE, if a sublibrary has not been named in any of the preceding ways, an error message will print and no copy will be performed.)

For VSE only, 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 within the scope of a COBOL statement, or
- "A", if within the scope of an ASM statement, or
- the member type named in an OPTIONS statement MEMTYPE parm, if any, or
- "SPECTWTR" otherwise

Under both MVS and VSE, you may use an **alias name** (rather than the actual member name) under certain circumstances. Alias names may be up to 70 characters long and must conform to the Report Writer naming conventions for file names. Aliases for library members are assigned in a special member in the standard Report Writer Copy Library. Under MVS, this is the member named SWALIAS in the PDS pointed to by the SWCOPY DD. Under VSE, this is the member named SWALIAS.SPECTWTR in the sublibrary named in the SUBLIB parm of an OPTION statement. The use of aliases is discussed beginning on page 308.

Alias checking is *not* performed (and therefore an alias may *not* be used) in each of the following cases:

- When the PDSDDN, SUBLIB or NOTALIAS parm is used in the COPY statement.
- When a member type is specified in the copyname parm.
- When the COPY statement appears within the scope of a COBOL or ASM statement.

EXAMPLES:

```
COPY:  SALES
```

The above example copies the member named SALES from a library. The earlier discussion explains how the library to use is determined.

```
COPY:  SALES–FILE
```

The above example also specifies that a member from a PDS or Sublibary should be copied. Since "SALES–FILE" itself is not valid as a member name, that name must be defined as an alias in the SWALIAS member of the copy library. As shown in Appendix F, "Sample File Definitions," "SALES–FILE" is an alias for the member name "SALES". Therefore, the above statement would cause the control statements in the copy library member named SALES to be copied.

DDNAME(ddname)

MVS 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 Report Writer does not access directly (such as PANVALET or MVS's LIBRARIAN.) Add a job step ahead of Report Writer to copy the desired proprietary data to a temporary sequential dataset. Then have Report 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/<u>NO</u>)

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)

MVS 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. This parm is valid only in conjunction with the copyname parm. 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: be sure 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, and the member type is TEST.

FIELD Statement

PURPOSE

Defines an input field to Report 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 Report Writer Copy Library.

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 275.

FIELD

SYNTAX

FIELD STATEMENT SYNTAX

```
FIELD:      fieldname
        [   ACCUM/NOACCUM                                        ]
        [   BIT(n)                                               ]
        [   COLUMN(nnnnn/expr/*)/DISP(nnnnn/expr/*)              ]
        [   DECIMALS(nn/0)                                       ]
        [   DESCRIPTION('text')                                  ]
        [   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 388.

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, Report 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 do 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 144.

EXAMPLES:

```
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.

```
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 Report 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 289.

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.

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)
```

> **Note:** when reading variable–length records, Report 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. See the KEEPRDW option (in the FILE, INPUT, READ and OPTIONS statements) 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. For 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:

```
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 position" 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 292.

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.

EXAMPLES:

```
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.

```
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 1 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`) .

DESCRIPTION('text')

You may specify a short free format description of the field in this parm. This information will be printed along with other information about the field in data dictionary listings.

EXAMPLE:

```
FIELD:  HIRE–DATE  COLUMN(34) TYPE(YYMMDD)
        DESC('DATE EMPLOYEE WAS FIRST HIRED')
```

When the HIRE–DATE field is listed data dictionary reports, the description shown above will be included.

DXPARM('text')

This parm is valid only for fields whose TYPE is a data exit (NUMEXIT, for example.) Anytime a user data exit program is called by Report Writer, the text specified in this parm is 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 297.

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 (MVS) or phase (VSE) that Report Writer will call in order to obtain the field's value. The use of this parm is discussed in the section beginning on page 297.

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 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 Report 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. Report 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 297.

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 Report 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. Report 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 297.

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 550) contains the complete list of 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 559.

The FORMAT parm that you specify in the FIELD statement tells Report 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 still be overridden by using a different display format parms directly in a COLUMNS 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 (LOTUS, 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.

EXAMPLES:

```
FIELD: SALARY      COLUMN(56) LENGTH(4) TYPE(PACKED) 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 form.

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 OPTION 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, Report 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 127 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 539) 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. Page 279 discusses how to compute a numeric field's length based on how many digits it has.

EXAMPLES:

```
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)

Some records contain fields that do not always begin at a fixed column within the record. In such cases there is usually another field in the record that tells the "offset" to the variably located field. The OFFSET parm is used to define such fields. The OFFSET parm can contain any numeric expression. Often it simply contains the name of another field which contains the appropriate offset value. Report Writer computes the value of the OFFSET parm anew for each input record. It adds that value to the contents of the COLUMN or DISP parm. This sum is then used as the starting byte of the field. (If no COLUMN or DISP parm is used, the "current location" value is added to the OFFSET value to determine the field's starting byte.) For additional information about the OFFSET parm, see the section beginning on page 295.

The OFFSET parm specified in one FIELD statement remains in effect for *all subsequent FIELD statements* until a new OFFSET parm is encountered. Use OFFSET(0) in a FIELD statement if you later want to define fields without any OFFSET value.

Note: the "current location" value is reset to column 1 (displacement 0) each time a FIELD statement with an OFFSET parm is encountered.

EXAMPLE:

```
FIELD: ADDR–OFFSET  DISP(26)    TYPE(HALFWORD)
...
FIELD: ADDR–LINE–1  LENGTH(30)  OFFSET(ADDR–OFFSET)
FIELD: ADDR–LINE–2  LENGTH(30)
...
```

In this example, the input record contains a halfword value named ADDR–OFFSET at displacement 26. This value is the offset within the record to an "address section" of the record. The address section consists of two 30–byte address lines. ADDR–LINE–1 is defined as a 30–byte character field. Since it is defined with an OFFSET parm, the field's location within the record is determined by adding the value of the ADDR–OFFSET field to the value of the COLUMN parm. Since no COLUMN (or DISP) parm was specified, the "current location" value is assumed. However, the "current location" is zero for this field, since the FIELD statement contains an OFFSET parm. Thus the field is simply located at the displacement contained in the ADDR–OFFSET field.

ADDR–LINE–2 is another 30–byte field. Again, no COLUMN or DISP parm is present, so the current location (now equal to displacement 30) is added to the contents of the ADDR-OFFSET field to derive the starting displacement of this field.

The example above used a single field as the OFFSET value. You are also allowed to use numeric expressions 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)
...
FIELD: LAST–FIELD  OFFSET(75 + (NUM–SLOTS * 12))  LENGTH(10)
```

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. The use of this parm is discussed in the section beginning on page 289.

EXAMPLES:

```
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.

```
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. The use of this parm is discussed in the section beginning on page 289.

EXAMPLE:

See the example above under the OFFTEXT parm.

TYPE(datatype/<u>CHAR</u>)

Specifies what type of data the field contains. There are five general categories of data that Report 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 539) contains the complete list of data types that Report 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 the default data type will be BIT.

EXAMPLES:

```
FIELD:  SALARY      TYPE(PACKED)    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 PACKED data. (Packed data is called COMP–3 in COBOL, 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 an input file to Report 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 Report 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 Report Writer Copy Library.

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 269
- using a file that is processed by a user I/O Exit is discussed in Appendix K, “I/O Exits“ (page 620)

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)                                       (MVS only)   ]
        [    DESCRIPTION('text')                                               ]
        [    EXITPARM('text')                                                  ]
        [    IOEXIT('program'[,'parm'] [,TRACE])                               ]
        [    KEEPRDW                                                           ]
        [    LRECL(nnnnn/1000)                                    (MVS only)   ]
        [    TYPE(SEQ/VSAM/DB2/EXIT)                              (MVS only)   ]
```

Standard Spelling	Alternate Spellings
DDNAME	DDN
DESCRIPTION	DESC
EXITPARM	PARM
FILE	FIL

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 388.

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.

type This parm is required. It tells Report Writer what kind of file is being defined. It must be one of the following values:

DASD	a SAM file residing on DASD (disk.) Use DASD (rather than VSAM) for VSAM–managed SAM files.
TAPE	a SAM file residing on a magnetic tape
VSAM	a VSAM file (ESDS or KSDS)
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. Report 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.) This parm is not required for EXIT type files. However, if a DLBL/TLBL is specified for an EXIT file, its value is passed to the exit program.

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 RDW which each variable–length record begins with.

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 273 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 Report 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 338.

EXAMPLE:

```
FILE: PROJECT  DB2NAME('DSN8230.PROJ')
```

The above example defines a file that will be referred to in Report Writer control statements as PROJECT. It refers to the DB2 table (or view) named DSN8230.PROJ.

DDNAME(ddname)

MVS only. The DDNAME parm specifies the name of a DD statement present in the JCL. This DD statement in the JCL will identify the actual data set to be read. Report Writer uses the DDNAME to open an input file and read from it. The DDNAME parm can also be specified in the INPUT or READ statements.

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 is passed to the I/O Exit program if 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.

DESCRIPTION('text')

You may specify a short free-format description of the file in this parm. This information will be printed along with other information about the file in data dictionary listings.

EXAMPLE:

```
FILE: EMPL–FILE TYPE(VSAM) DDNAME(EMPLDD)
      DESC('EMPLOYEE MASTER FILE –– NEW VERSION')
```

When the EMPL–FILE file is listed in data dictionary reports, the description shown above will be included.

EXITPARM('text')

This parm specifies any information that should be passed to user data exit programs. (Most installations will not use data exits, and will not need this parm.) Anytime a data exit program is called by Report 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 297.

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 the data exit program each time it is called.

IOEXIT('program' [,'parm'] [,TRACE])

EXIT files only. This parm provides the information necessary for Report Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix K, "I/O Exits" (page 620.)

MVS 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 (MVS) or phase (VSE) that Report 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 Report 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, Report 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 269 (MVS) and page 273 (VSE.)

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/<u>1000</u>)

MVS only. Specifies the length of the largest record that might be found in the file. If this parm is not specified, Report Writer assumes a default LRECL of 1000.

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.

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: SALES–FILE  LRECL(4000)
```

The above example defines a file named SALES–FILE. The LRECL parm specifies that records as large as 4000 bytes may be encountered in the file. Report Writer will reserve a 4000 byte I/O area for reading records from this file.

TYPE(SEQ/VSAM/DB2/EXIT)

MVS only. Specifies the type of access method to use when reading this file. If not specified, SEQ is assumed. Valid types are:

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.)
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 Report 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 Report 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 pages. In addition, the following parts of the manual relate to the FOOTNOTE statement:

- using the FOOTNOTE statement is discussed beginning on page 180

SYNTAX

FOOTNOTE STATEMENT SYNTAX

```
FOOTNOTE: print–expression [/ print–expression] [/ print–expression]
```

Note: the syntax for the print-expressions is shown on page 477.

Standard Spelling	Alternate Spellings
FOOTNOTE	FOOT

Note: the syntax of the FOOTNOTE statement is identical to that of the TITLE statement.

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 174.

You may also use empty FOOTNOTE statements. An empty FOOTNOTE statement results in one blank footnote line.

PRINT–EXPRESSION SYNTAX (IN FOOTNOTE STATEMENT)

Each **print–expression** consists of one or more **items**, optionally separated by numeric **spacing factors**:

```
FOOTNOTE:      [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[(  [   BIZ                         ]
             [   display–format              ]
             [   LEFT/CENTER/RIGHT           ]
             [   width                       ]   )]

'literal'[(      width                           )]
```

Standard Spelling	Alternate Spellings
CENTER	CJ
FOOTNOTE	FOOT
LEFT	LJ
RIGHT	RJ

fieldname

Specifies that the footnote line should contain the contents of this field. The field's data will be taken from the *last detail record* before the footnote line.

The field must be available to Report Writer at the time the FOOTNOTE 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" 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, it is formatted with this picture: PIC'ZZZ9' (4 digits). You can override this format by using a numeric display format parm. This fieldname can also be abbreviated as #PAGE.

EXAMPLE:

```
FOOTNOTE: #TODAY  /  'ABC COMPANY' /  'PAGE'  #PAGENUM
```

The above example contains three print expressions. It will produce a footnote 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 footnote is left–justified; the second part is centered; the third part is right–justified.

'literal'

Specifies that the footnote 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 should appear between two items in a footnote line. A spacing factor of zero is allowed. (It results in two items appearing in the footnote with no blank spaces between them.) If no spacing factor is given, the default is to leave one blank space between items.

EXAMPLE:

```
FOOTNOTE: #TODAY  /  'ABC COMPANY'  /  'PAGE' 6 #PAGENUM
```

The above example specifies that 6 blank spaces should be left between the literal text "PAGE" and the contents of the #PAGENUM field. The footnote would now look like this:

```
12/31/99                              ABC COMPANY                           PAGE      nnnn
```

BIZ

This “blank if zero” parm specifies that blanks should appear in the footnote 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:

```
FOOTNOTE: 'EMPLOYEES HIRED ON' HIRE–DATE(BIZ)
```

The above example causes the HIRE–DATE field in the footnote to be left blank whenever it contains a zero date.

display–format

Specifies how the contents of a field should be formatted in the footnote line. A complete list of display formats is found in Appendix B, "Display Formats" (page 550.) If this parm is not specified, Report 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 (see page 559)

EXAMPLE:

```
    FOOTNOTE: #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 to be spelled out in the date. 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 footnote 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 footnote line.

EXAMPLE:

```
    FOOTNOTE: #TODAY(LONG1,CENTER)
```

The above example specifies a footnote 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 footnote 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 under 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 footnote. The CENTER parm is needed to cause these shorter dates to be *centered within* the 18–byte area in the footnote line. The footnote line produced by the above statement would now look like this:

```
                                DECEMBER 31, 1999
```

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:

```
    FOOTNOTE: 'ABC COMPANY'  /  #TODAY(LONG1,RIGHT)
```

The footnote line produced by the above statement would look like this:

```
ABC COMPANY                                                      DECEMBER 31, 1999
```

width

This is a numeric parm that specifies the number of characters to reserve for an item in the footnote line. Use this parm if the default width is larger or smaller than you desire.

EXAMPLE:

```
FOOTNOTE: 'PAGE' #PAGENUM(9)
```

The above example specifies that 9 characters (not digits) should be reserved to display the #PAGENUM field in the footnote line. The resulting footnote would look like this:

```
PAGE n,nnn,nnn
```

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.

> **Note:** 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 PC file

LEARNING MORE

The complete syntax of the INCLUDEIF statement is shown on the following pages. 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 24
- a lesson on using the INCLUDEIF statement with PC files begins on page 86
- the syntax of conditional expressions is covered beginning on page 399
- suggestions on writing INCLUDEIF statements for maximum CPU efficiency are given in Appendix I, "Speed-Up Tips" (page 603)

INCLUDEIF

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 399.

EXAMPLES

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.)

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 contained the name JONES. 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, Report 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 Report 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, Report 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.

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 Report 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 Report Writer Copy Library (typically used 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 17
- information on using the INPUT statement with DB2 tables begins on page 338
- reading a file that is processed by a user I/O Exit is discussed in Appendix K, “I/O Exits“ (page 620)

INPUT

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)                                     (MVS only)   ]
    [    EXITPARM('text')                                                ]
    [    IOEXIT('program' [,'parm'] [TRACE})                             ]
    [    KEEPRDW                                                         ]
    [    KEYRANGE('begin' ['end'])                                       ]
    [    LIST(YES/NO)                                                    ]
    [    LRECL(nnnnn)                                       (MVS only)   ]
    [    ORDERBY(fieldname [ASC/DESC] [,] ... )             (DB2 only)   ]
    [    RECNAME(name/filename)                                          ]
    [    SHOWFLDS(YES/NO)                                                ]
    [    TYPE(SEQ/VSAM/DB2/EXIT)                            (MVS only)   ]
    [    WHERE(search–condition)                            (DB2 only)   ]
```

Standard Spelling	**Alternate Spellings**
DDNAME	DDN
EXITPARM	PARM
INPUT	INP
NO	N
TYPE	TYP
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. Each record that passes the conditions in the INCLUDEIF statement (if any) 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 301.

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 470.) For examples of using this parm, see page 273.

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(nnn)

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, Report 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(nnn)

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, Report 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, Report 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 affect your run.

Specifying CLEAR(ZEROS) causes Report 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. Report 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 301.

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 was defined in an earlier FILE statement. (In that case, the earlier FILE statement must have specified 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 Report 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 Report Writer file name of PROJECT. That is, other Report Writer control statements that refer to this input file will refer to PROJECT (rather than to DSN8230.PROJ.)

DDNAME(ddname)

MVS 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 FIELD 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')

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. (Most installations will not use exits, and will not need this parm.) Anytime a user data exit is called by Report 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 297.

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 Report Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix K, "I/O Exits" (page 620.)

MVS 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 (MVS) or phase (VSE) that Report 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 Report 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 needs in order to process the file. This parm can be up to 255 bytes in length.

TRACE This parm is optional. When specified, Report 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:

```
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 269 (MVS) and page 273 (VSE.)

EXAMPLE:

```
INPUT:  EMPL–FILE  KEEPRDW
```

The above example specifies that the RDW should be kept when reading records from the EMPL–FILE. Thus, assuming that EMPL–FILE is actually 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 value. 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.

EXAMPLES:

```
INPUT:  EMPL–FILE  KEYRANGE('03')
```

The above example specifies that only records whose keys begin with "03" should be read from the EMPL-FILE.

```
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 along with the other control statements 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 any records copied from the copy library should be listed in the control listing.

LRECL(nnnnn)

MVS 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.

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.

ORDERBY(fieldname [<u>ASC</u>/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 Report Writer in an "arbitrary" order. This is not normally of any consequence, as Report 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 Report 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 beginning on page 344.

EXAMPLE:

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       ORDERBY(DEPTNO, PROJNAME)
```

The above example would cause DB2 to pass the rows from the project table to Report Writer in department number order, with "ties" being passed in project name order.

RECNAME(name/<u>filename</u>)

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 388. The use of the RECNAME parm is discussed beginning on page 232.

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:

```
COLUMNS:  EMP.DATE
```

SHOWFLDS(YES/NO)

Specifies whether Report 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 Report 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.

TYPE(SEQ/VSAM/DB2/EXIT)

MVS 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 475.

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 Report Writer. (Report 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 Report 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 342. The precise syntax rules for the WHERE parm are given beginning on page 350.

EXAMPLE:

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')
       WHERE(DEPTNO = 'D21')
```

In the example above, the WHERE parm causes DB2 to return to Report 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 Report 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. Report Writer opens this file for sequential input at the beginning of the report process. Each record in the file is then read sequentially. (Or, if the KEYRANGE parm was used, then each record within the specified range is read sequentially.) As each record is read, the conditions specified in the INCLUDEIF statement (if any) are evaluated, using the data from that record (and any auxiliary input file records related to it.) Based on these conditions, the record will either be included in the run, or will not be included. If the record is to be included, a sort record is built using data from the record. The sort record is then passed to the sort routine, 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. The report (or PC file) is then written from the information in these sorted records.

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.

The OPTIONS statements should 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 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.

SYNTAX

OPTIONS STATEMENT SYNTAX

```
OPTIONS:
    [   ASCIITABLE('text')                                               ]
    [   ASMLIB('library.sublibrary')                          (VSE only] ]
    [   AUTOSORT]                                                        ]
    [   CENTURY(nn/0)                                                    ]
    [   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')                                              ]
    [   FORMAT(display-format [,display-format] ... )                    ]
    [   HEADINGSEP('char'/'|')                                           ]
    [   HGCOLHDG                                                         ]
    [   KEEPRDW                                                          ]
    [   LEFTMARGIN(nnn/0)                                                ]
    [   MAXINCLUDE(nnnnn)                                                ]
    [   MAXINPUT(nnnnn)                                                  ]
    [   MAXINVSHOW(nnnnn)                                                ]
    [   MAXPAGES(nnnnn)                                                  ]
    [   MAXPRINT(nnnnn)                                                  ]
    [   MEMTYPE('type'/'SPECTWTR')                            (VSE only) ]
    [   MISSOFFSET                                                       ]
    [   MULTICOLHDG                                                      ]
    [   NOCC                                                             ]
    [   NOCHECK                                                          ]
    [   NOCLEARIO                                                        ]
    [   NOCOLHDGS                                                        ]
    [   NOGRANDTOTAL                                                     ]
    [   NOMAXMSG                                                         ]
    [   NOOVERPRINT                                                      ]
    [   NOSYSINLIMIT                                                     ]
    [   NOTITLES                                                         ]
    [   OUTATTR(type[,'dlbl/tlbl'][,SYSnnn][,recsize][,blksize])  (VSE only) ]
    [   OUTLRECL(nnnnn)                                       (MVS only) ]
    [   OUTTYPE(SEQ/VSAM)                                     (MVS only) ]
    [   PAGELENGTH(nnn/60)                                               ]
    [   PC/MAINFRAME/OUTPUT/ACCESS/COREL/CSV/DBASE3/DBASE4/EXCEL
        /FOXPRO/HARVARD/LOTUS/MS-WORKS/PARADOX/QUATTRO/RBASE             ]
    [   PRTSETUP('text')                                                 ]
    [   PRTSHEET('text')                                                 ]
    [   QCHAR('char'/'"')                                                ]
    [   SINGLE/DOUBLE/TRIPLE                                             ]
    [   SKIPBLANKDET                                                     ]
    [   SKIPZERODET                                                      ]
    [   SORTNAME('program'/'SORT')                                       ]
    [   SORTSIZE(nnnn/256)                                               ]
    [   SORTWORKNUM(n/0)                                      (VSE only) ]
    [   SPLITDETAIL                                                      ]
```

(Continued on next page)

OPTIONS STATEMENT SYNTAX (CONTINUED)

```
[   STCKADJ(nn)                                          ]
[   SUBLIB('library.sublibrary')              (VSE only) ]
[   SUMMARY                                              ]
[   TIMEDELIM('char'/':')                                ]
[   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
NOCOLHDGS	NOCOLHDG
NOGRANDTOTAL	NOGRANDTOTALS, NOGRANDTOT, NOGRANDTOTS, NOGRAND
NOOVERPRINT	NOOVERPRT
NOTITLES	NOTITLE
OPTIONS	OPTION, OPTS, OPT
PAGELENGTH	PAGELGTH, PAGELEN, PAGEL
SINGLE	SINGLESPACE
SKIPBLANKDET	SKIPBLKDET
SPLITDETAIL	SPLITDET
SUBLIB	SUBLIBRARY
TRIPLE	TRIPLESPACE
ZERODIVBYZERO	ZERODIVZERO, ZERODIVZ
ZEROINVDATA	ZEROINV
ZEROOVERFLOW	ZEROOVER

ASCIITABLE('text')

Use this option to specify your own translation table to be used by 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 Report Writer what value to return for each of the 256 possible byte values it could encounter when performing the #ASCII built-in function on some operand. If this option is not specified, Report 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 Report 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 Report 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/0)

Specifies the century cutoff year. This option tells Report Writer which century a 2–digit year belongs to. You can use this option to process dates into the 21st century, even if they only contain 2 digits for the year. Any year *below* the specified value is considered to be in the 21st century. All years greater than or equal to the specified value are considered to be in the 20th century. The default value of 0 causes all 2–digit years to be treated as 20th century dates (since no year is less than zero).

Note: this option does not affect the way dates with 4–digit years are processed.

EXAMPLE:

```
OPTION: CENTURY(5)
```

The above example states that any dates with a year less than 5 are in the 21st century. Thus, the date 8/31/04 would mean August 31, 2004. However, 8/31/05 would mean August 31, 1905.

OPTIONS

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 Report Writer's default when creating many type of PC files. This option also suppresses titles, footnotes and all page break logic.

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: 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:

```
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 Report Writer recognizes date literals in the control statements. Date literals must always be written using slashes as delimiters.

DB2PLAN('plan'/'<u>SPECTnnn</u>')

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 Report Writer's DB2 Option. (Plan names are assigned when a DB2 "bind" is performed. A DB2 bind is required as a part of the procedure for installing Report Writer with the DB2 Option.) Report Writer assumes that you use a plan name of "SPECTnnn", where nnn is the Report Writer version number. (Thus, for release 2.7.1 of Report Writer, a plan name of SPECT271 is assumed.) If you used a different plan name to bind Report Writer in your shop, you must tell Report Writer your plan name via the DB2PLAN option. Enclose the plan name in quotation marks or apostrophes. For example, if you bind Report Writer with a plan name of XYZ12345, you would need to use a statement like the following:

```
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 Report 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 are 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 reports to a smaller, more convenient size. Remove the option when your are ready for the final run.

EXAMPLE:

```
OPTIONS: DETAIL(10)
INPUT:   EMPL–FILE
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 Report Writer what value to return for each of the 256 possible byte values it could encounter when performing the #EBCDIC built-in function on some operand. If this option is not specified, Report Writer uses a default EBCDIC translation table.

EXAMPLE:

See the example under the ASCIITABLE option which has the same syntax.

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 valid display formats is found in Appendix B, "Display Formats" on page 550.) 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 "delimited ASCII" output 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 (LOTUS, 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. For 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, specify numeric, date and time display formats *after* the CHARACTER or HEX format in the FORMAT parm. For 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 different lines. The default heading separator character is the vertical bar (|).

Tip: The vertical bar is the "Shift 1" key on most mainframe terminals. Some PC keyboards that emulate mainframe terminals do not have a key that shows the straight vertical bar. (The "pipeline" character is not the same as a vertical bar.) On many of these keyboards, the right–hand square bracket key (]) is used to send a vertical bar to the mainframe.

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.

KEEPRDW

When reading non–VSAM input files with variable length records, Report Writer considers column 1 of the input record to be the first byte *after* the RDW (record descriptor word.) This option tells Report 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 considered 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, INPUT or READ statements.

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/<u>0</u>)

Specifies a number of blank spaces to use as a left margin when printing the report. By default, there is no left margin.

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 MAXINPUT option, when the records required for your report are not the first records in the input file. (See also the related MAXINPUT option.)

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/<u>10</u>)

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 invalid field hex displays.

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'/<u>'SPECTWTR'</u>)

VSE only. Specifies the default member type to use when reading members from the Report 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.

EXAMPLE:

```
OPTIONS: MEMTYPE('SW')
```

The above statement tells Report 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 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). This suppresses the `***F***` indicator in reports.

MULTICOLHDG

By default, when more than one COLUMNS statement is used Report Writer does not automatically produce column headings. (The TITLE statement is often used in such situations to manually create column headings.) If you want Report Writer to automatically provide column headings for you in a report that has multiple COLUMNS statements, specify:

```
OPTIONS: MULTICOLHDG
```

Report 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 easier way to produce column headings. Of course, the first COLUMNS statement may contain override column headings as usual. Those override column headings will then be used in the report. Any default or explicit column headings in the 2nd and later COLUMNS statements are ignored.

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 MAINFRAME) also suppresses the carriage control character.

NOCHECK

Only relevant if the MAXPRINT or MAXPAGES option is used. Tells Report Writer that the NOCHECK option is in effect for your shop's sort program. This means Report Writer can safely quit the sort early when the MAXINPUT or MAXINCLUDE limit has been reached. Otherwise, in order to prevent a SORT ABEND, Report Writer must continue to process the remainder of the sort file (flushing the records), which takes a little more processing time.

NOCLEARIO

For some input files, Report Writer clears (sets to hex zeros) the I/O area where records are read before performing each read. (See under the FILE statement's CLEAR parm on page 477.) The NOCLEARIO option specifies that such clearing should not be performed for any files used in the run. When such clearing is not necessary, suppressing it may improve performance.

NOCOLHDGS

Specifies that Report Writer should not create column headings for the report or PC file. Report Writer also defaults to the NOCOLHDG option for all reports that use more than one COLUMNS statement.

NOGRANDTOTAL
Specifies that Grand Totals are not wanted for this report.

NOMAXMSG
Only relevant if the MAXPRINT or MAXPAGES option is used. Tells Report 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. 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.

NOSYSINLIMIT
By default Report 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, Report Writer will use 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.

OUTATTR(type [,'dlbl/tlbl'] [,SYSnnn] [,recsize] [,blksize])
VSE only. This parm describes the attributes to use for Report Writer's output. The section beginning on page 374 discusses the use of this parm.

type This parm is required. It tells Report Writer what kind of device to write the output to. It must be one of the following values:

PRT PRINTER	a printer–type device (including POWER print queues)
DASD	a SAM file on a DASD device (disk). (Use this 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 Report 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:** for report output, the first byte in each record is used as a "carriage control character." So in the example above, only 132 bytes would be 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. Report Writer output is always written as fixed length records (and fixed length blocks, if blocked.)

OUTLRECL(nnnnn)

MVS only. Specifies the LRECL to be used for the output records written by Report Writer. This parm is mainly intended for use when writing to a VSAM output file. The LRECL chosen by Report 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), if any, 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

OUTTYPE(<u>SEQ</u>/VSAM)

MVS only. Specifies the type of I/O to be used by Report Writer when writing output records. If OUTTYPE(VSAM) is specified, the dataset named in the SWOUTPUT DD statement must be an existing, ESDS VSAM dataset. If Report Writer's output will be written to a SYSOUT DD or to a non–VSAM file, OUTTYPE(SEQ) (the default) must be used.

PAGELENGTH(nnn/<u>60</u>)

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/MAINFRAME/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 78.

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 parm 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, Report Writer will send this string to the printer to cause it to skip to a new page.

> **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 Report 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'/<u>'"'</u>)

Specifies the "quotation character" to use in conjunction with the QCHAR, Q–MM–DD–YY and Q–HH–MM–SS 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 in the QCHAR, Q–MM–DD–YY and Q–HH–MM–SS display formats. 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"`.

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. 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 Report Writer to skip (suppress) any detail report line (or PC file record) that is all blank. For the purposes of this option, "detail lines" means: the lines printed for each individual 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 154.

> **Note:** only the first 256 bytes of each line are examined when checking for blank detail lines.

SKIPZERODET

This option causes Report Writer to skip (suppress) 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)
- 00/00/0000 (zero dates)
- 00:00:00 (zero times)

For the purposes of this option, "detail lines" means: the lines printed for each individual 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 154.

> **Note:** only the first 256 bytes of each line are examined when checking for zero detail lines.

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 Report Writer should use the program named SORT2 to perform any necessary sorts.

SORTSIZE(nnnn/256)

This parm specifies the size 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, Report 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 Report Writer is 8191 (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.)

EXAMPLE:

```
OPTIONS: SORTSIZE(64)
```

The above statement tells Report Writer to pass the sort program a size parm of 64K (if the sort program is used.)

SORTWORKNUM(n/0)

VSE only. This parm specifies how many, if any, external work files should be used to perform Report Writer VSE's internal sort. 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 381.) Use this parm to tell Report 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, Report 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 Report 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. IBM's STCK machine instruction stores its date–time stamps in GMT. Report Writer normally converts STCKDATE and STCKTIME values from GMT to local time. The number of hours to add or subtract 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)
```

SUBLIB('library.sublibrary')

VSE only. Specifies the name of the VSE sublibrary to use as the Report Writer copy library.

EXAMPLE:

```
OPTIONS: SUBLIB('LIB.SPECTWTR')
```

The above statement causes the Librarian dataset named LIB.SPECTWTR to be used as the Report 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 with 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 2 blank lines to 0 blank lines. This prevents the summary lines in the report from being 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 use 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 Report Writer recognizes **time literals** in the control statements. Time literals must always be written using colons as delimiters.

ZERODIVBYZERO

Tells Report Writer to assign a value of zero to COMPUTE fields whenever a division by zero error occurs. This suppresses the `***Z***` error indicator in reports.

ZEROINVDATA

Tells Report 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 Report Writer to assign a value of zero to COMPUTE fields whenever 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 Report 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 Report Writer.

FEATURES

Use the READ statement to:

- specify the **name** of an auxiliary input file for a report
- 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 Report 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 65
- a lesson on using the READ statement in PC files begins on page 113
- advanced techniques involving the READ statement are discussed beginning on page 228
- the use of the READ statement with DB2 tables is discussed beginning on page 345

- suggestions on writing READ statements for maximum run–time efficiency are given in Appendix I, "Speed-Up Tips" (page 603)
- reading a file that is processed by a user I/O Exit is discussed in Appendix K, “I/O Exits“ (page 620)

SYNTAX

READ STATEMENT SYNTAX

```
READ:        filename
        [    ATTR(VSAM/EXIT [,'dlbl'] ,recsize)          (VSE only)   ]
        [    BUFND(nnn)                                  (VSAM only)  ]
        [    BUFNI(nnn)                                  (VSAM only)  ]
        [    CLEAR(SPACES/ZEROS/NO)                                   ]
        [    COPY(YES/NO)                                             ]
        [    DB2NAME('[qualifier.]name')                 (DB2 only)   ]
        [    DDNAME(ddname)                              (MVS only)   ]
        [    EXITPARM('text')                                         ]
        [    GENERIC                                                  ]
        [    IOEXIT('program' [,'parm'] [TRACE])                      ]
        [    KGE                                                      ]
        [    LIST(YES/NO)                                             ]
        [    LRECL(nnnnn)                                (MVS only)   ]
        [    MULTI                                                    ]
        [    ORDERBY(fieldname [ASC/DESC] [,] ... )      (DB2 only)   ]
        [    READKEY(fieldname)                                       ]
        [    RECNAME(name/filename)                                   ]
        [    SHOWFLDS(YES/NO)                                         ]
        [    TYPE(VSAM/DB2/EXIT)                         (MVS only)   ]
        [    WHERE(search–condition)                     (DB2 only)   ]
```

Standard Spelling	Alternate Spellings
DDNAME	DDN
EXITPARM	PARM
GENERIC	GEN
NO	N
READKEY	KEY
RECNAME	NAME
TYPE	TYP
YES	Y

The filename parm is *required.* In addition, either a READKEY parm (for VSAM files) or a WHERE parm (for DB2 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 VSAM 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 301.

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 VSAM 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 273.

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(nnn)

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, Report 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 3 data control intervals when processing the EMPL–FILE.

BUFNI(nnn)

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, Report 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 3 data control intervals and 6 index control intervals when processing the EMPL–FILE.

CLEAR(SPACES/ZEROS/NO)

When processing certain types of input files, Report 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 affect your run. Specifying CLEAR(ZEROS) causes Report 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. The NOCLEARIO parm in the OPTIONS statement can be used to prevent clearing of *all files* in a run.

EXAMPLE:

```
READ: EMPL–FILE  READKEY(EMPL–NUM)  CLEAR(NO)
```

The above statement names the PAYROLL–FILE as the input file for a run. Report 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 301.

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. (In that case, the earlier FILE statement must have specified 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 Report 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 Report Writer file name of EMPLOYEE. That is, other Report Writer control statements that refer to this input file will refer to EMPLOYEE (rather than to DSN8230.EMP.)

DDNAME(ddname)

MVS 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 FIELD 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')

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. (Most installations will not use exits, and will not need this parm.) Anytime a user data exit is called by Report 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 297.

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 234.

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 2 bytes long, while the defined key length for the EMPL–FILE file is 3 bytes. Thus, when performing the above read, the record read will be the first one where the first 2 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 Report Writer to process an EXIT type input file. More information on I/O Exits can be found in Appendix K, "I/O Exits" (page 620.)

> **MVS 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 (MVS) or phase (VSE) that Report 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 Report 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 needs in order to process the file. This parm can be up to 255 bytes in length.

TRACE This parm is optional. When specified, Report 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 234.

> **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 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 along with the other control statements 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 any records copied from the copy library should be listed in the control listing.

LRECL(nnnnn)

MVS only. Specifies the length of the largest record that might be found in the file. If this parm is omitted, the LRECL value (if any) will be taken from FILE statement that defined the file. If no LRECL parm is specified in either the FILE or the READ statement, a default LRECL of 1000 is assumed.

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(s) should be read. If MULTI is not specified, only the first record which passes the WHERE parm condition(s) will be read.

The use of the MULTI parm is discussed in the section beginning on page 235.

> **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 2 bytes long, while the defined key length for the EMPL–FILE file is 3 bytes. Thus, when performing the above read, all records will be read where the first 2 bytes of their key equals the contents of SHORT–KEY.

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 Report Writer, one by one. If MULTI is not specified, Report 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 Report 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, Report 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 to read depends on two other parms which may be present in the READ statement:

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, Report 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 key (or partial key) matches the key (or 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, Report Writer will read *all records* whose key (or partial) key matches the READKEY value. If MULTI is not specified, only the first record with a matching key (or 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 key (or partial key) matches the key (or partial key) in the READKEY value. If no record matches the READKEY value, then the record with the next greater key (or 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 be 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 70, as well as below, for an example of computing a read key.)

This field must be available at the time the READ statement is processed. Therefore, the READKEY field 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.

EXAMPLES:

```
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.

```
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, "Sample File Definitions") 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/<u>filename</u>)

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 additional 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 388. The use of the RECNAME parm is discussed beginning on page 232.

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/<u>NO</u>)

Specifies whether Report 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 Report 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)

MVS 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 475.

> **Note:** only VSAM, DB2 and EXIT type files may 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, Report 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 Report 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 345. Its syntax is discussed in the section beginning on page 350.

EXAMPLE:

```
INPUT: PROJECT
       DB2NAME('DSN8230.PROJ')

READ:  EMPLOYEE
       DB2NAME('DSN8230.EMP')
       WHERE(EMPNO = RESPEMP)
```

Here's how Report Writer processes the above statements. The primary input to the report is the project DB2 table. So, Report Writer will retrieve all rows from that DB2 table. After it fetches each row from the project table, Report 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, Report Writer would fetch *all such rows.* When MULTI is not specified, Report 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*, 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 Report Writer processes multiple input files.

Program Flow With No READ Statements

To understand how auxiliary input files are processed, let's first notice how Report Writer produces a report when *no* auxiliary input files are used. In such a case, Report Writer repeats the following steps over and over.

Step 1)	read a record from the primary input file
Step 2)	evaluate the INCLUDEIF statement using the data from this input record
Step 3)	if the record passes the INCLUDEIF tests, pass the record to Report Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
Step 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.

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, Report Writer now assembles **"logical input records."** A logical input record is a group of records consisting of one record from each input file. The manner in which these logical records are assembled is different depending on whether any READ statement uses the MULTI parm.

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 Report 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.

Step 1)	assemble a "logical input record" consisting of one record from each of the input files
Step 2)	evaluate the INCLUDEIF statement using the data from this logical input record
Step 3)	if the logical input record passes the INCLUDEIF tests, pass the logical input record to Report Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
Step 4)	if the logical input record does not pass the INCLUDEIF tests, discard the logical input record

As mentioned, the specific way that Report 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 Report 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, Report 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 Report 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:

Step 1a)	read a record from the primary input file
Step 1b)	create one logical input record by also reading a single record from each auxiliary input file
Step 2)	evaluate the INCLUDEIF statement using the data from this logical input record
Step 3)	if the logical input record passes the INCLUDEIF tests, pass the logical input record to Report Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
Step 4)	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 Report Writer does *logically*. During actual processing, there may be cases where it is not necessary for Report 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, Report 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, Report 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 with the MULTI parm are used, the process is similar to that just described. But now the number of record combinations that Report Writer must assemble into logical records increases exponentially. For each primary input file record, Report 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:

Step 1a)	read a record from the primary input file
Step 1b)	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)
Step 2)	for each logical input record, evaluate the INCLUDEIF statement using the data from that logical input record
Step 3)	if the logical input record passes the INCLUDEIF tests, pass the logical input record to Report Writer's output phase (where it will be sorted and formatted into the desired report or PC file)
Step 4)	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 was 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, Report 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:

FIELD TYPE	DEFAULT VALUE
Character	Blanks
Numeric	Zero
Date	Zeros (00/00/0000)
Time	Zeros (00:00:00)
Bit	OFF

SORT Statement

PURPOSE

This statement specifies how Report Writer should sort the 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 48
- a lesson on using the SORT statement in PC files begins on page 100
- the use of the SORT statement to request control breaks is discussed beginning on page 182

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 Report 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 Report 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 spaces and/or a comma.

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.

<u>ASC</u>/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 displayed 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, a 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 displayed 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, 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 displayed 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, 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. Any 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" on page 183.

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 displayed 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, 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 displayed 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, 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.

EXAMPLES:

```
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.

```
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 Report Writer Determines Sort Order

All data processed by Report Writer falls into one of five general categories of data. The following table shows how each type of data is 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. **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.
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.

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 38
- advanced examples of using the TITLE statement are shown beginning on page 165
- using TITLE statements to create column headings is discussed in "How to Produce Multi–Line Reports" on page 147

TITLE

SYNTAX

TITLE STATEMENT SYNTAX

```
TITLE:  print–expression [/ print–expression] [/ print–expression]
```

Note: the syntax for the print-expressions is shown on page 533.

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 174.

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.

PRINT–EXPRESSION SYNTAX (IN TITLE STATEMENT)

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[(      [   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.

The field must be available to Report 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" 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, it formatted with this picture: PIC'ZZZ9' (4 digits). You can override this format by using a numeric display format parm. This fieldname 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–justified; the second part is centered; the third part is right–justified.

'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 6 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
```

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 2 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 550.) If this parm is not specified, Report Writer uses the display format from:

- the FIELD or COMPUTE statement that defined the field
- an OPTIONS statement FORMAT parm
- the default display format (see page 559)

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 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.

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:

```
                               DECEMBER 31, 1999
```

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                                                  DECEMBER 31, 1999
```

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
```

(This page left blank intentionally.)

Appendices

Appendices Table of Contents

Appendices Table of Contents (Continued)

Appendix A. Data Types

There are five general categories of data that Report 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 how a particular field's data is stored within an input record. A field's data type is defined to Report Writer with the TYPE parm in its FIELD statement.

The following charts show the data types that Report Writer supports for each category of data. These 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	Report 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

DATA TYPES FOR NUMERIC FIELDS			
DATA TYPE	DESCRIPTION	PROGRAMMING LANGUAGE EQUIVALENTS	LENGTH ALLOWED (See Note 1)

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	Report 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

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 (including 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'  ½/96' are Jan. 2, 1996.`	8
MM–DD–YYYY	`MM/DD/YYYY` date in character format (including 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'  ½/1996' 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 (including slashes or other delimiters.) (2) Example: `C'12/31/1996' and C'12.31.1996' are Dec. 31, 1996.` Example: `C'1/2/1996  ' and C'  ½/1996' are Jan. 2, 1996.`	10
YYMMDD	`YYMMDD` date in character format. Example: `C'961231' is Dec. 31, 1996.`	6
YYYYMMDD	`YYYYMMDD` date in character format. Example: `C'19961231' is Dec. 31, 1996.`	8

DATA TYPES FOR DATE FIELDS		
DATA TYPE	DESCRIPTION (See Note 1)	LENGTH
YYYY–DD–MM	YYYY/DD/MM date in character format (including slashes or other delimiters.) (2) 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
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

DATA TYPES FOR DATE FIELDS		
DATA TYPE	**DESCRIPTION** (See Note 1)	**LENGTH**
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
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	Report 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.) Report Writer automatically converts the STCK value from GMT to local time. For more details, see the STCKADJ parm in the OPTIONS statement.	8
ABSDATE	Report 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	Report 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 (including 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
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–HHMM	`HHMM` time in packed format. Decimal digits are *not* allowed. Example: `X'01234C' is 12:34`	3	3
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)

Data Types for Time Fields			
Data Type	Description	Default Length	Length Allowed (See Note 1)
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.)` 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)
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)` Example: `X'04B5' is 02:00:30.0` (6)	N/A (5)	1–8 (3)

DATA TYPES FOR TIME FIELDS			
DATA TYPE	DESCRIPTION	DEFAULT LENGTH	LENGTH ALLOWED (See Note 1)
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)
STCKTIME	Report 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.) Report Writer automatically converts the STCK value from GMT to local time. For more details, see the STCKADJ parm in the OPTIONS statement. STCKTIME fields always have 6 decimal digits.	8	8
ABSTIME	Report 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

Data Types for Time Fields			
Data Type	**Description**	**Default Length**	**Length Allowed** (See Note 1)
TIMEEXIT	Report 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 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	Report 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 statement to indicate how data should be formatted in a report or output file. When no display format is specified, Report Writer formats data using a default display format. To override Report Writer's default, specify one of the display formats found in the following pages. For example:

```
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 display format in the FIELD statement defines a default format to use for that field whenever it appears in a report or output file.

Here is another way display formats can be used:

```
COLUMNS:  HIRE–DATE(DD–MM–YYYY)
```

The above COLUMNS statement tells Report Writer to format the HIRE–DATE field in "DD/MM/YYYY" format, *for the current run only*.

Display formats can be specified in the following statements:

- the FIELD statement
- the COMPUTE statement
- the COLUMNS statement
- the BREAK statement
- the TITLE statement
- the FOOTNOTE statement
- the OPTIONS statement (FORMAT option)

For more information on these uses, see under the appropriate statement's description in Chapter 9, "Control Statement Syntax."

The display formats that can be used 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.

A table showing the *default* display formats for each type of data appears on page 559.

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. (Use the QCHAR parm of the OPTIONS statement to specify a character other than the quotation mark to use as the delimiter with this display format.)	`"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 8 character 0s and/or 1s to show the individual bits within the data.	`11000001`

Numeric Display Formats

DISPLAY FORMATS FOR NUMERIC FIELDS		
DISPLAY FORMAT	DESCRIPTION	EXAMPLE
	(the following formats are suggested for use 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 150.	`******`
		`********`
		`****`

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 393.	`(800) 555–1212`
		`123–45–6789`

DISPLAY FORMATS FOR NUMERIC FIELDS		
DISPLAY FORMAT	DESCRIPTION	EXAMPLE
(the following formats are intended mainly for use in building output records)		
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'`
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'`
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'`
HALFWORD **HALF**	Same as BINARY, but with an implied width of 2 bytes.	`X'04D2'`
FULLWORD **FULL**	Same as BINARY, but with an implied width of 4 bytes.	`X'0001E240'`

Date Display Formats

Display Formats for Date Fields		
Display Format	**Description**	**Example**
	(the following are suggested for use 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
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

Display Formats for Date Fields		
Display Format	**Description**	**Example**
LONG3	`DD MMMMMMMMMMM YY`	`31 DECEMBER 96`
(the following are intended mainly for use in building output records)		
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		
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 to specify a delimiter other than the slash (/).
(2) Use the QCHAR parm in the OPTIONS statement to specify a delimiter other than the quotation mark.

Time Display Formats

DISPLAY FORMATS FOR TIME FIELDS		
DISPLAY FORMAT	DESCRIPTION	EXAMPLE
(the following are suggested for use 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)	`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)	`1:31 AM`
		`1.31 PM`

HHMM	`HHMM`	`1331`
TPICTURE'...' **TPICT'...'** **TPIC'...'** **TP'...'**	User defined "time picture." (Time pictures are discussed on page 398.) 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`
(the following are intended mainly for use in building output records)		
Q–HH–MM–SS	`"HH:MM:SS"` time in quotation marks. This format is useful for formatting time fields for use in PC files. Use the QCHAR parm in the OPTIONS statement to specify a delimiter other than the quotation mark. (1)	`"13:30:45"`

DISPLAY FORMATS FOR TIME FIELDS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Q–HH–MM	"HH:MM" time in quotation marks. Use the QCHAR parm in the OPTIONS statement to specify a delimiter other than the quotation mark. (1)	"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 to specify a delimiter other than the colon (:).

Default Display Formats

The following table shows Report Writer's standard default display format for each type of data.

Note: the default display formats are changed by certain options, including the FORMAT option, the PC file options 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 290.	`FIELDNAME`
			`NOT FIELDNAME`

Appendix C. Built–In Fields

Report 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 Report Writer built–in fields. Following the table, each field is discussed in more detail.

REPORT 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 Report Writer is currently executing.
#TIME	Time of day, including AM or PM ("12:45 PM")
#TIME24	Time of day 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.

REPORT WRITER BUILT–IN FIELDS	
FIELD NAME	DESCRIPTION
	Time Built–In Fields
#HHMMSS	The system time.

Character Built–In Fields

#DAYNAME

Allowed in any control statement. A 9–byte field containing the name of the day of the week on which the job began. The value of this built–in field does not change during execution. The use of this field is discussed on page 172.

Example: `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. The use of this field is discussed on page 206.

#JOBNAME

Allowed in any control statement. An 8–byte character field containing the name of the job that is executing Report Writer.

#TIME

Allowed in any control statement. An 8–byte character field containing the system time at which the job began. The value of this built–in field does not change during execution. The time is in 12–hour format, including either AM or PM. The use of this field is discussed on page 172.

Example: `12:31 PM`

#TIME24

Allowed in any control statement. A 5–byte character field containing the system time at which the job began. The value of this built–in field does not change during execution. The time is in 24 hour format. The use of this field is discussed on page 172.

Example: `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 206.

#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 206.

#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 210. For a discussion of "control group levels", see page 211.

#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 172.

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 MVS, #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.)

Example: `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 default date display format that is in effect (usually MM–DD–YY.) The use of this field is discussed on page 172.

Example: `12/01/99`

Time Built–In Fields

#HHMMSS

Allowed in any control statement. Contains the system time at which the job began. The value of this built–in field does not change during execution. By default, it is formatted using the default time display format that is in effect (normally HH-MM-SS.)

Example: `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 bit.)

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 the syntax for each function. 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
- a field name
- a computational expression (which may itself involve other built–in functions)

Separate the arguments with blanks and/or a comma.

The following table lists the Report Writer built–in functions. Following the table, each function is discussed in more detail.

REPORT WRITER BUILT–IN FUNCTIONS

FUNCTION	**DESCRIPTION**	**PAGE**
	Functions that Return a Character Value	
#AND	returns the result of ANDing two character strings	page 569
#ASCII	returns the ASCII equivalent of an EBCDIC string	page 569
#COMPRESS	concatenates multiple fields and compresses out extra blanks	page 569
#DAY	returns the day of the week for a given date	page 570
#EBCDIC	returns the EBCDIC equivalent of an ASCII string	page 570
#FORMAT	converts a numeric, date or time value to a character value	page 570
#LCASE	returns the lower–case value of a character string	page 571
#LEFT	returns the leftmost *n* characters of a character string	page 571
#MONTH	returns the month name pertaining to a given date	page 571
#OR	returns the result of ORing two character strings	page 571
#PARSE	returns one individual word parsed out of a character string	page 572

REPORT WRITER BUILT–IN FUNCTIONS		
FUNCTION	DESCRIPTION	PAGE
#RIGHT	returns the rightmost *n* characters of a character string	page 572
#SUBSTR	returns a substring from a character string	page 572
#TRANSLATE	translates one set of characters within a character string to another set of characters	page 573
#UCASE	returns the upper–case value of a character string	page 573
#XOR	returns the result of XORing two character strings	page 574
#YEAR	returns the 4–byte year pertaining to a given date	page 573
Functions that Return a Numeric Value		
#ABS	returns the absolute value of a number	page 575
#DAYNUM	returns the day of the month (1–31) for a given date	page 575
#INDEX	returns the starting column of a substring	page 575
#INT	returns the integer portion of a number	page 575
#MAKENUM	converts a character, date or time value to a numeric value	page 575
#MAX	returns the greater of two or more values	page 576
#MIN	returns the smaller of two or more values	page 577
#MOD	returns the remainder left after division ("modulus")	page 577
#MONTHNUM	returns the month number (1–12) for a given date	page 577
#NUMWORDS	returns the number of words within a character string	page 578
#ROUND	returns the rounded value of a number	page 578
#YEARNUM	returns the 4–digit year for a given date	page 578
Functions that Return a Date Value		
#MAKEDATE	converts a character or numeric value to a date	page 579
Functions that Return a Time Value		

REPORT WRITER BUILT–IN FUNCTIONS		
FUNCTION	DESCRIPTION	PAGE
#MAKETIME	converts a character or numeric value to a time	page 580
Functions that Return a Bit Value		
#OFF	returns the "off" bit value (0)	page 581
#ON	returns the "on" bit value (1)	page 581

Functions that Return a Character Value

#AND(char1,char2)

Performs the logical AND operation on the two character arguments and returns the result. 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.

Examples:

```
COMPUTE: A = #AND(X'01FF',X'035E')
```
would result in `A=X'015E'`

Here is an example of using the #AND built–in function to test individual bits within a status flag. 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.

```
COMPUTE: TEMP = #AND(STATUS, X'A0')
COMPUTE: BOTH–BITS–ARE–ON = WHEN(TEMP = X'A0') ASSIGN(#ON)
INCLUDEIF: BOTH–BITS–ARE–ON
```

#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')` would result in `A=X'313233'`

Note: to specify your own EBCDIC-to-ASCII translation table, use the ASCIITABLE option in the OPTIONS statement (page 496.) Otherwise, Report Writer uses a default translation table.

#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 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.

Examples:

```
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             "`

#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')` would result in `A=X'F1F2F3'`

Note: to specify your own ASCII-to-EBCDIC translation table, use the EBCDICTABLE option in the OPTIONS statement (page 500.) Otherwise, Report Writer uses a default translation table.

#FORMAT(fieldname [,display–format] [,width] [,BIZ] [,LEFT/CENTER/RIGHT])

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. 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 COLUMNS statement syntax on page 437.

Examples:

`COMPUTE: A = #FORMAT(#TODAY)`
might result in `A='03/31/95'`

`COMPUTE: A = #FORMAT(#TODAY, LONG1, CENTER)`
might result in `A='  MARCH 31, 1995  '`

`COMPUTE: A = #FORMAT(SALES–DATE, BIZ)`
might result in `A='03/31/95'` (when SALES–DATE is not all zeros), or
`A='        '` (when SALES–DATE is all zeros)

`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)`
might result in `A='      92,125.89'` (when TOTAL–SALES is not zero)
`A='               '` (when TOTAL–SALES is zero)

#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 characters remain unchanged. 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)` would result 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.

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. 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')`
would result 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 = 123`:

```
COMPUTE: PACKED   = #FORMAT(AMOUNT,PACKED,2)
COMPUTE: PACKED–F = #OR(PACKED,X'000F')
```

would result 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. Individual words are 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.

Examples: (Assume that `DESC = "THIS IS A DESCRIPTION"`)

```
COMPUTE: A = #PARSE(DESC,1)   would result in   A="THIS           ".

COMPUTE: A = #PARSE(DESC,2)   would result in   A="IS             ".

COMPUTE: A = #PARSE(DESC,–1)  would result in   A="DESCRIPTION    ".

COMPUTE: A = #PARSE(DESC,5)   would result in   A="               ".
```

Note: to parse a text using a delimiter other than blanks, use the #TRANSLATE built-in function to first translate the desired delimiter character into a blank. For example, to parse a text using a dot as the delimiter, use:

```
COMPUTE: A = #PARSE(#TRANSLATE(DESC,"."," "),1)
```

Note: use the NUMWORDS built–in function to count the number of words in a character string.

#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.

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.

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* re–evaluated for additional translation. The size of the value returned by this function is the size of the char1 argument.

> **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 Report 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.

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)

```
COMPUTE: A = #TRANSLATE(DESC,"TA","XY")
```

would result in `A="XHIS IS Y DESCRIPXION"`.

#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 characters remain unchanged. The size of the value returned by this function is the size of the character argument.

> **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.

Example: (Assume that `NAME = "Smith     "`)

```
COMPUTE: SORT–NAME = #UCASE(NAME)
```

would result in `SORT–NAME = "SMITH     "`

#YEAR[(date)]

Returns the year portion of the date argument as a 4–byte character field. If no date argument is present, the system date is used.

Example:

```
COMPUTE: A = #YEAR(HIRE–DATE)
```

might result in `A="1995"`

#XOR(char1,char2)

Performs the logical XOR operation on the two character arguments and returns the result. 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')
```

would result in `A=X'0022'`

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. If no date argument is present, the system date is used.

Example:

```
COMPUTE: A = #DAYNUM(3/31/1995)
```
results in A=31

#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 decimal digits, if any, are simply truncated, regardless of the sign of the argument.

Examples:

`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

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 twentieth or twenty-first century. (You can use the #MAKEDATE function to convert a numeric day in century back into a date.)

Examples:

```
COMPUTE: A = #MAKENUM(12/31/2000)       results in A=36890
COMPUTE: A = #MAKENUM(1/1/2001)         results in A=36891
```

Example of computing the number of days between two dates:

```
COMPUTE: NUM–DAYS = #MAKENUM(END–DATE) – #MAKENUM(START–DATE)
```

If `END–DATE = 4/2/1997` and `START–DATE = 3/28/1997`, 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).

```
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, just add this statement:

```
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:

```
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:

```
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:

```
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:

```
COMPUTE: FIRST–DATE = #MAKEDATE(#MIN(#MAKENUM(DATE1), #MAKENUM(DATE2)))
```

#MOD(num1,num2)

Returns the remainder left when the first argument is divided by the second argument.

Examples:

`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(3/31/1995)` 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 572).

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)

```
COMPUTE: A = #NUMWORDS(DESC)  would result 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:

```
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. Note: num2 must be a literal integer. The *number* of decimal digits returned by this function is the same as the number of decimal digits in the num1 argument.

Examples:

```
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
COMPUTE: A = #ROUND(–12345.678, 2)     results in A=–12345.680
```

#YEARNUM[(date)]

Returns the numeric value of the year portion of the date (including the century). If no date argument is present, the system date is used.

Example:

```
COMPUTE: A = #YEARNUM(3/31/95)     results in A=1995
```

Functions that Return a Date Value

#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, Report Writer assigns the century based on the CENTURY Option in effect, if any.

Example:

```
COMPUTE: A = #MAKEDATE('19950331')
```
results in A=3/31/1995

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 in century. (Use this function to change the results of the #MAKENUM(date) function back into a date.)

Example:

```
COMPUTE: A = #MAKEDATE(36890)
```
results in A=12/31/2000

Example of adding 7 days to a date:

```
COMPUTE: NUM–DATE =        #MAKENUM(SALES–DATE)
COMPUTE: NUM–DATE–PLUS–7 = NUM–DATE + 7
COMPUTE: DATE–PLUS–7 =     #MAKEDATE(NUM–DATE–PLUS–7)
```

Functions that Return a Time Value

#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 containing the time 13:59:59.

For **numeric arguments,** the argument is treated as being 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 the #MAKENUM function back into a time.)

Example:

```
COMPUTE: A = #MAKETIME(3600)
```
results in A containing the time 01:00:00.

Example of adding 5 minutes to a time:

```
COMPUTE: NUM–TIME =        #MAKENUM(SALES–TIME)
COMPUTE: NUM–TIME–PLUS–5 = NUM–TIME + (5 * 60)
COMPUTE: TIME–PLUS–5 =     #MAKETIME(NUM–TIME–PLUS–5)
```

Functions that Return a Bit Value

#OFF

Always returns an "off" value.

Examples:

```
COMPUTE: A = #OFF
```
results in A="off"

```
COMPUTE: SALES–AWARD = WHEN(TOTAL–SALES > 50000) ASSIGN(#ON)
                       ELSE                      ASSIGN(#OFF)
```

results in SALES–AWARD being "on" (or true) if sales are greater than 50,000; otherwise results in SALES–AWARD being "off" (or false.)

#ON

Always returns an "on" value.

Examples:

```
COMPUTE: A = #ON
```
results in A="on"

```
COMPUTE: SALES–AWARD = WHEN(TOTAL–SALES > 50000) ASSIGN(#ON)
                       ELSE                      ASSIGN(#OFF)
```

results in SALES–AWARD being "on" (or true) if sales are greater than 50,000; otherwise results in SALES–AWARD being "off" (or false.)

Appendix E. Error Indicators

Sometimes an error prevents Report 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 possible error codes:

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. (See page 232.)
****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. Use the MISSOFFSET option to ignore this error condition.
****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. Use the ZEROINVDATA (or just ZEROINV) option to ignore this error condition.
****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 the section beginning on page 131.) 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. Report 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.)

Use the ZEROOVERFLOW (or just ZEROOVER) option to ignore this error condition.

****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)
```

Use the ZERODIVZERO (or just ZERODIVZ) option to ignore this error condition.

Suppressing Error Indicators

In some cases you may not be concerned with certain error conditions. In that case, you can suppress the asterisk error indicators by using one or more of the following options.

`OPTION: ZEROINVDATA`	—	treat fields containing invalid data as if they contained zeros instead. This suppresses the `***I***` indicator. May also be abbreviated ZEROINV.
`OPTION: ZEROOVERFLOW`	—	assign a value of zero to COMPUTE fields that have overflow errors. This suppresses the `***V***` indicator. May also be abbreviated ZEROOVER.
`OPTION: 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.

`OPTION: 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 Report 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 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:  SALES–AMOUNT(PIC'ZZZ,ZZZ')
```

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.

Testing for Invalid Data

You may want to detect when certain fields contain invalid data and use different processing for such fields. Here are two methods you might use to detect invalid data in a field.

To detect a specific invalid value in a field, just compare the field to the appropriate hexadecimal value (such as hex zeros or hex F's, perhaps.) Here is an example of detecting and excluding records that have hex F's in the AMOUNT field:

```
INCLUDEIF:  AMOUNT ¬= X'FFFFFFFFFFFF'
```

Note that when using hexadecimal literals (as above) you need to know the exact length of the field in the input file (6 bytes in this example.) For comparisons involving explicit literals, Report Writer compares the raw input file data — no data conversion is attempted. The hexadecimal literal should be the same length as the field. Otherwise, Report Writer pads the shorter operand with blanks, which is not usually what you want.

You can use a second method to detect any kind of invalid data in a field. This method is useful if you do not know in advance what invalid value a field may have. This method utilizes the fact that Report Writer always evaluates as *false* any conditional expression that attempts to process an invalid operand. In the following statement, we set a bit field name GOOD–AMOUNT to be true only if the AMOUNT field contains a valid numeric value:

```
COMPUTE: GOOD–AMOUNT = WHEN(AMOUNT = AMOUNT) ASSIGN(#ON)
```

As long as AMOUNT contains any valid value, the test `AMOUNT = AMOUNT` will always be true and GOOD–AMOUNT will be assigned a bit value of ON (or true). If AMOUNT contains any invalid value, Report Writer will evaluate the WHEN parm expressions as false and GOOD-AMOUNT will be assigned the OFF value (false.) You could then use this bit field in other statements as desired. Here are two statements that use the GOOD–AMOUNT bit field:

```
INCLUDEIF: GOOD–AMOUNT

COMPUTE: TAX–PERCENT = WHEN(GOOD–AMOUNT)  ASSIGN(TAX  / AMOUNT)
                       ELSE               ASSIGN(0.08)
```

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 unformatted contents of each file is also shown.

The SWALIAS member of the copy library contained the alias entries shown in the box below.

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

```
*****************************************************************
*                                                               *
*  REPORT 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)
```

Notes:

- 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)
```

CONTENTS OF SALES–FILE (UNFORMATTED)

```
JOHNSON   037041SOUTH010138060903509503121025OOACE ELECTRICAL 21355598710079952
BAKER     044045WEST 01370008220360950326120909JACKS CAFE     21455511240102978
MORRISON  042036EAST 00443502660360950329153022STAR MARKET    40855576540599907
MORRISON  042045EAST 00296501780360950330190541A1 PHOTOGRAPHY 40855577860600919
SIMPSON   041039EAST 00149900900360950401081757EUROPEAN DELI  40855565430150916
JOHNSON   039036NORTH02344514070370950401170247VILLA HOTEL    41555576300929926
JOHNSON   039044NORTH00099800600370950405143310MARYS ANTIQUES 41555512560000997
BAKER     044037WEST 01357508150360950412143112JACKS CAFE     21455511240231916
THOMAS    045037WEST 00099800600360950414154138YOGURT CITY    21455517895421997
JONES     036042NORTH00102500620370950415075832EZ GROCERY     41555548720810977
JONES     036039NORTH01217607310370950415080159TOY TOWN       41555515001200907
JONES     036039NORTH00102500620370950415135241TOY TOWN       41555515000523977
JOHNSON   037042SOUTH05000030000350950416114833ACME BUILDING  21355521211025976
SIMPSON   041042EAST 00238701430360950430153021J & S LUMBER   40855523212451916
```

DEFINITION STATEMENTS FOR EMPL–FILE

```
***********************************************************
*                                                         *
*  REPORT 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')
```

Note:

- 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)
```

CONTENTS OF EMPL–FILE (UNFORMATTED)

```
036JONES          JERRY          8001312MA012098765007842509890995601105115608698071333425125 MAIN S
TREET     SAN FRANCISCO  CA940124155557653
037JOHNSON        THOMAS         7506211MA912040334012886999242156015213502119970102411878400 LINDA
 VISTA    SCOTTSDALE     AZ900126025556654
039JOHNSON        LINDA          7911252FA004779981010475023551459034172201020100082311131212 LINCOLN
 DRIVE    SANTA ROSA     CA954124155556785
040MACDONALD      RICHARD        8207042M 889790013000602560980054850006871300599250072610525 FOOTHI
LL DRIVE  PLEASANTON     CA945684155559887
041SIMPSON        TIMOTHY        8212013MA11205045600160872388012875805109030099812013291589876 WEST
 53 STREETARCADIA        CA910068185551887
042MORRISON       MICHAEL        7911303MA90012055601549805499250141926112212801009189185098 SOUTH L
AKESIDE DRGLENDALE       CA912028185554748
043CHRISTOPHERSON MELISSA        8108151FA41509076100654766531138072216549010805007092590161752 TIMB
ERIDGE RD PHOENIX        AZ905026025554556
044BAKER          VIVIAN         8206044FA878190156014792125892133610249990224001332178944667 CRESTH
AVEN BLVD WALNUT CREEK   CA945984155551209
045THOMAS         MARTIN         8206044MA77683822101186019349148890718045051425012130092577812 S. H
UNTINGTON CONCORD        CA945194155551152
```

DEFINITION STATEMENTS FOR PRODUCT–FILE

```
****************************************************************
*                                                              *
*  REPORT 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)
```

Note:

- 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)
```

CONTENTS OF PRODUCT–FILE (UNFORMATTED)

```
NEWP907INKPADS
NEWP916RED PENS
NEWP919GREEN PENS
OLDP926DESK CALENDARS
NEWP952PENCILS (NO. 1)
OLDP976CHAIRS
OLDP977PAPER CLIPS
NEWP978HOLE PUNCHERS
OLDP997MAILING LABELS
```

DEFINITION STATEMENTS FOR STATE–FILE

```
************************************************************
*                                                          *
*  REPORT 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)
```

Note:

- 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)
```

CONTENTS OF STATE–FILE (UNFORMATTED)

```
AZARIZONA
CACALIFORNIA
OROREGON
WAWASHINGTON
```

Appendix G. Sample Data Exit Program

Report 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 Chapter 5, "How to Define Your Input Files" (page 297.)

Data exit programs can written in Cobol, PL/1 or Assembler. A sample data exit program written in Assembly language appears on the following pages. This sample program performs 5 simple functions in order to illustrate data exit calls for each of the five types of data. A parm is passed to the exit program each time it is called. That parm tells the exit program which function is desired. The functions performed by this sample program are:

CALLING PARM	**FUNCTION PERFORMED**
T	Returns a time value. This sample program simply returns the constant time 12:34:56.
N	Returns a numeric value. This program simply increments a counter and returns its value.
B	Returns a bit value. This program returns the value of the low-order bit of the record field passed to it.
D	Returns a date value. This program returns the constant date 12/31/1996.
R	Returns a character value. This program "reverses" the characters in the record field passed to it and returns that reversed value.

Use this sample program as a model for writing your own data exit programs. (A copy of this program is contained in the sample Copy Library found in your installation tape.)

Note the $DX DSECT located near the end of the program. That DSECT shows the parm list that Report Writer passes to all data exit programs. Specifically, when a data exit program is called by Report Writer, register 1 will point to a fullword. That fullword will contain the address of the $DX DSECT parm list.

Figure 108 (page 595) 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 of our sample exit program.

When processing a report request, Report 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 Report Writer.

> **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 name 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 choose whichever of these approaches you prefer.

Sample Data Exit Program Written in Assembly Language — 1 of 3

```
SWDEXIT  TITLE '- SAMPLE REPORT 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
*
***********************************************************************
* 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
```

Sample Data Exit Program Written in Assembly Language — 2 of 3

```
***********************************************************************
* 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 — 3 of 3

```
**********************************************************************
* 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 REPORT 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
```

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
```

Notes:

- This report uses 5 fields that are created by the data exit program named SWDEXIT

Figure 108 A report that uses a data exit program

Appendix H. How to Import PC Files

This Appendix shows the steps used to import Report Writer's PC files into the following PC programs:

- Lotus 1–2–3 for Windows (page 597)
- Lotus 1–2–3 DOS versions (page 597)
- Excel (page 597)
- Quattro Pro (page 598)
- Paradox for Windows (page 598)
- Paradox DOS versions (page 599)
- Microsoft Works (page 599)
- Corel Chart (page 599)
- PowerPoint (page 600)
- Harvard Graphics (page 600)
- dBASE III and IV (page 601)
- R:BASE (page 601)
- word processing programs (page 602)

While we aren't able to include every version of every PC program in this Appendix, we have included a representative sample. If the PC program you want to use is not listed in one of the following sections, we suggest:

- see if the steps described for Lotus or Excel also work with your PC program. Most newer PC software has similar Menu structures and functions related to the opening of input files.
- check your PC program's online Help (or printed documentation.) Search for such keywords as: IMPORT, TEXT FILE, ASCII, DELIMITED FILE, FILE FORMATS.
- consider a two–step approach. If your PC program cannot import comma delimited files, it may still be able to import other types of files (such as Lotus or Excel spreadsheets). For example, Microsoft's PowerPoint does not import comma delimited files. However, it *will* import Excel spreadsheet files and Excel charts. In such a case, use Report Writer to create a PC file for Excel and first import it into Excel. Then save the Excel spreadsheet. You can then import the Excel spreadsheet file into your desired program. An example of this appears on page 600.

Tip: when downloading your PC files, it is often helpful to give them a name ending with `.CSV` or `.TXT` on the PC. That helps some PC programs recognize your PC file as a comma delimited import file.

Importing a PC file into Lotus 1–2–3 for Windows

Use the following statement to create a PC file for Lotus 1–2–3:

```
OPTIONS: LOTUS
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Lotus 1–2–3 Release 5 for Windows. From an empty spreadsheet do the following:

```
choose FILE
choose OPEN
for File Type, select "TEXT (txt;prn;csv;dat;out)"
for File Name, enter the name of your PC file (for example, C:SALES.CSV)
click on OK
```

Importing a PC file into Lotus 1–2–3 (DOS Versions)

Use the following statement to create a PC file for Lotus 1–2–3:

```
OPTIONS: LOTUS
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Lotus 1–2–3 (DOS version 3.0) From an empty spreadsheet do the following:

```
press the / key (to bring up the menu)
choose FILE
choose IMPORT
choose NUMBERS
enter the filename (for example, C:SALES.CSV)
```

Importing a PC File into Excel

Use the following statement to create a PC file for Excel:

```
OPTIONS: EXCEL
```

PC files for Excel differ from most other PC files in that "tab" character is used as the column delimiter (rather than a comma.) This also means that quotation marks are not required for character fields.

Once you have created your PC file and downloaded it to your PC, here is how to import it into Excel Version 5.0. From an empty spreadsheet, do the following:

```
choose FILE
choose OPEN
under "List Files of Type", select Text Files (*.prn;*.txt;*.csv)
under "File Name", enter the name of your PC File (for example, C:SALES.TXT)
click on OK
(at this point, the Text Import Wizard screen appears)
under "Original Data Type", select Delimited
under "Start Input at Row", select 1
under "File Origin", select Windows (ANSI)
click on FINISH
```

Alternative Method: the method described above creates what Excel calls a "text" or "tab delimited" file. Excel will also import files formatted as "Comma Separated Values" (CSV). To produce a CSV file, use the following statement:

```
OPTIONS: CSV
```

When downloading this file to your PC, you may want to name it with a .CSV extension. (That tells Excel that the file is formatted as a CSV file.) Then to import the PC file, follow these steps:

```
choose FILE
choose OPEN
under List Files of Type, select Text Files (*.prn;*.txt;*.csv)
under File Name, enter the name of your PC File (for example, C:SALES.CSV)
click on OK
```

Importing a PC File into Quattro Pro

Use the following statement to create a PC file for Quattro Pro:

```
OPTIONS: QUATTRO
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Quattro Pro Version 5.0 for DOS. From within Quattro Pro do the following:

```
press / key (to bring up the menu)
choose Tools
choose Import
choose Comma and "" Delimited Files
enter the filename (for example, C:SALES.CSV)
press ENTER
```

Importing a PC File into Paradox for Windows

Use the following statement to create a PC file for Paradox:

```
OPTIONS: PARADOX
```

Once you have created your PC file and downloaded it to your PC, you will use Paradox to convert the PC file into a Paradox table. Then you can open the Paradox table in the normal way. Here is how to convert the PC file to a table under Paradox for Windows Version 1.0. From within Paradox, do the following:

```
choose FILE
choose UTILITIES
choose IMPORT
under Type, select <Delimited Text>
under File Name, enter the name of your PC file (for example, C:SALES.CSV)
click on OK
(a "Delimited ASCII Import" dialog box will appear at this point)
leave the File Name entry alone
enter a New Table Name
click on OK
```

After creating the new table, open it in the normal manner:

```
choose FILE
choose OPEN
choose TABLE
under File Name, enter the path and name of the new table you just created
click on OK
```

Importing a PC File into Paradox (DOS Versions)

Use the following statement to create a PC file for Paradox:

```
OPTIONS: PARADOX
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Paradox DOS Version 3.5. From within Paradox, do the following:

```
choose Tools
choose ExportImport
choose ASCII
choose Delimited
enter the filename (for example, C:SALES.CSV)
enter a new table name
```

Importing a PC File into Microsoft Works

Use the following statement to create a PC file for Microsoft Works:

```
OPTIONS: MS–WORKS
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Microsoft Works Version 2.0a. From within Microsoft Works do the following:

```
click on SPREADSHEET (at the Startup dialogue box)
choose FILE
choose OPEN EXISTING FILE
under List Files of Type, select "TEXT (*.TXT)"
under File Name, enter the name of the PC file (for example, C:SALES.CSV)
click on OK
click on SPREADSHEET (at the Open File As dialogue box)
```

Importing a PC File into Corel Chart

Use the following statement to create a PC file for Corel Chart:

```
OPTIONS: COREL
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into Corel Chart Version 3.0. From within Corel Chart, do the following:

```
choose FILE
choose NEW
choose a chart type, for example BAR
(an empty spreadsheet grid will appear)
```

```
choose FILE
choose IMPORT DATA
under List Files of Type, select "CSV Data (*.CSV)"
under File Name enter the name of your PC file (for example, C:SALES.CSV)
click on OK
```

Importing a PC File into PowerPoint

PowerPoint does not import comma delimited values directly. However it does import Excel spreadsheets.

Use the following statement to create a PC file for Excel:

```
OPTIONS: EXCEL
```

Once you have created your PC file and downloaded it to your PC, import it into Excel (as described on page 597.) Then save the Excel spreadsheet and exit Excel.

Here is how to import the data from your new Excel spreadsheet into a slide in PowerPoint Version 4.0. From within a PowerPoint presentation, do the following:

```
choose INSERT NEW SLIDE
choose GRAPH as the autolayout type for your slide
double click where it says "Double Click to Add Graph"
(a Presentation Database grid will appear)
select the upper left grid box with the cursor (to import data starting there)
choose EDIT (from the main menu)
choose IMPORT DATA
under List Files of Type, select "Microsoft Excel Files (*.xl*)"
under File Name enter the name of your saved Excel file (for example, C:SALES.XLS)
click on OK
(at this point you can edit the imported data in the database grid, perhaps deleting any
unneeded columns or rows)
choose CLOSE (from the Presentation Database dialogue box's pulldown menu)
(PowerPoint will format the data into a chart, which you can then massage as desired.)
```

Alternative Method:
You can also create and save your desired chart while you are still within Excel. Then import that Excel Chart into your PowerPoint slide. Follow the same steps as above, except that you will now choose `IMPORT CHART` (rather than `IMPORT DATA`) and will then list and name a Microsoft Excel Chart (*.xlc) file.

Importing Files into Harvard Graphics

Use the following statement to create a PC file for Harvard Graphics:

```
OPTIONS: HARVARD
```

Once you have created your PC file and downloaded it to your PC, there are a number of ways that it can be imported into Harvard Graphics (depending on the kind of chart you want). As an example, to import it as a bar chart into Harvard Graphics DOS Version 2.12, do the following:

```
choose Create new chart
choose Bar/line
press <ESC> (from the "X Data Type" menu, to use the defaults)
press <ESC> (to return to the menu screen)
choose Import/Export
choose Import delimited ASCII
(select the correct Directory and Filename)
(on the "ASCII Delimiters" menu, select the following:)
    Quote character: "
    End of field delimiter: ,
    end of record delimiter: #13#10
choose YES (to the prompt "Import first record as series legends:")
```

Note: the column headings for Harvard Graphics files are a little different than for other PC files. For Harvard Graphics, Report Writer always creates a *single line* of column headings. (This is because Harvard Graphics only accepts a single line of column headings when importing files.) This means that column headings which normally are split onto multiple lines will be run together into a single column heading line. A single space will appear where the line breaks would otherwise have occurred. If this results in column headings that are too long or that look odd, you may wish to specify override column headings (in your COLUMNS statement) when creating files for Harvard Graphics.

Importing a PC File into dBASE IV

Use the following statement to create a PC file for dBASE IV:

```
OPTIONS: DBASE4
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into a dBASE IV structure. First create a structure that corresponds field–for–field to the Report Writer output file. Then enter the following "dot" commands:

```
USE structure
APPEND FROM filename TYPE DELIMITED
```

Importing a PC File into R:BASE

Use the following statement to create a PC file for R:BASE:

```
OPTIONS: RBASE
```

Once you have created your PC file and downloaded it to your PC, here is how to import it into R:BASE. From within a fresh R:BASE session, do the following:

```
choose Tools
choose Import/export
choose ASCII delimited...
enter the filename (for example, C:SALES.CSV)
choose YES (to the question "Importing Date Values?")
check MM/DD/(YY)YY (by hitting <ENTER>)
press <F2>)
choose NEW (to the question "Choose database:")
enter a new database name
choose NEW (to the question "Choose table:")
enter new table name
choose (,) (to the prompt "Enter the column separator character:")
press <F2> (to the prompt "Press F2 to continue")
fill in the correct column names on the screen, if desired
```

```
press <ESC> (to return to the menu)
choose Data
choose Load
```

Importing Files into Word Processing Programs

Use the following statement to create a PC file for most word processing programs:

```
OPTIONS: NOCC
```

Notice that for word processor programs we do *not* use any PC program option. Word processors can normally import a report "just as it is." That is, columns need not be "delimited". And the data need not be specially formatted or enclosed in quotes. And you normally *do* want to include report titles, column headings, and Grand Totals.

The NOCC option simply tells Report Writer to omit the "carriage control" character from the beginning of each output record. The carriage control characters are not needed since the output records will be going to a *file*, rather than to a *printer*.

After you have downloaded the PC file to your PC, import it into your word processing program as an "ASCII file." The exact way to do this varies between different word processing programs. Check the program's online Help (or printed documentation). Search for such words as "Importing", "ASCII files", "text files" or "DOS files."

For example, to import such a PC file into WordPerfect 5.1, you would do the following:

```
press TEXT IN/OUT (CTL–F5)
choose DOS text (1)
choose Retrieve (CR/LF to [HRt]) (2)
enter the filename (for example, C:SALES.TXT)
```

When importing reports into word processing programs, there are several things to keep in mind:

- The report should not be too **wide**. If the report is wider than the width of a page, the word processing program will probably break each line into multiple lines, ruining the appearance of the report. Using a width parm (in the COLUMNS statement) to create smaller columns may help reduce the width of your report (see page 131.)
- Use a font with a **small point size** (in the word processing program) to allow wider reports to fit on a page
- If your report is still too wide, try using a **"landscape" page layout**
- Use a **non–proportional (monospaced) font** to display the report. Otherwise the report may be skewed as the word processing program uses a slightly different width for each character. Some fonts that are usually non–proportional are: COURIER, LINE PRINTER, and MONOSPACED.
- Disable justification. If the word processor program attempts to justify the report lines, they will probably become skewed.

Appendix I. Speed–Up Tips

Because Report 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 Report 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 Report Writer run at its fastest.

The INCLUDEIF statement simply consists of a conditional expression. Report 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 Report Writer may not always need to perform every test in a conditional expression. By writing your conditional expressions so that Report 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. We could write it as:

```
INCLUDEIF: SEX = 'F'  AND  NAME = 'JOSEPHSON'
```

If we write the statement as above, Report 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. Report 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, Report 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, Report 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, Report 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 Report Writer would have to go 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. For 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 were in department 4 than were 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, Report 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 Report 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, Report 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, Report Writer will not have to read the auxiliary record 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 608.

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 section in this Appendix on speed–up tips for the INCLUDEIF statement (page 603.)

The second consideration when writing conditional COMPUTE statements is the order of the WHEN parms themselves. Remember that when evaluating a conditional COMPUTE statement, Report 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 Report 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, Report 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. Report 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 Report 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), Report 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 every record in the entire input file. 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 very 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 Report 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 Report 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 amount of I/O that Report Writer has to perform.

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 Report Writer's VSAM I/O performance with.

When you write Report Writer a 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 Report 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.

Report 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, Report Writer chooses a default number of data and index buffers 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 Report 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, Report 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 Report Writer process it. The first SALES–FILE record might be for EMPL-NUM 036, for example. Report 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 Report Writer step to perform the pre–sort, if you like. This is explained on page 263.

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 Report 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, Report 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, Report 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 Report 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 490.

INCLUDEIF Statement Order

If you have not done so, please read the speed–up tips for the INCLUDEIF statement (page 603.) Pay particular attention to the subsection titled Fields from Auxiliary Input Files (page 605.) Writing your INCLUDEIF statement so as to eliminate unnecessary reads to auxiliary input files can greatly reduce the amount of slow VSAM I/O that must be performed.

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 Report 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) Report Writer would need to perform the read to the VSAM file.

Clearing I/O Areas

When processing certain types of files, Report 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 may result in somewhat 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. Report 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.

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.

OPTION	DESCRIPTION
MAXINPUT(nnnnn)	Tells Report Writer to *read* only the specified number of records from the input file. After reading that many records, Report Writer acts as if it has hit EOF (end of file) on the input file and produces the final report or PC file. Example: `OPTIONS: MAXINPUT(500)`
MAXINCLUDE(nnnnn)	Tells Report Writer to *include* only the specified number of records in the run. This option is different 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 Report Writer to read as many records as necessary until it finds the specified number of records that can be included in the report.

Example: `OPTIONS: MAXINCLUDE(500)`

MAXPAGES(nnnnn)
MAXPRINT(nnnnn)

Tells Report 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.

Example: `OPTIONS: MAXPAGES(500)`

If you use either of these options, also see the NOCHECK option (page 503).

DETAIL(nnnnn)

Tells Report 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.

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 Report Writer from having to perform any 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 later record layout modification affects 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 Report 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. Report Writer automatically performs the conversion needed to make the literals compatible with the data field in each case.

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 Report Writer— it's ease of use. Thus, we don't recommend using this technique in routine cases.

Appendix J. Year 2000 Information

Report Writer version 2.7.1 (and later versions) are Year 2000 Ready. We use the following definition of "Year 2000 Ready" provided by IBM:

> “A product is ‘Year 2000 Ready’ if, when used in accordance with its associated documentation, it is capable of correctly processing, providing and/or receiving date data within and between the twentieth and twenty-first centuries, provided that all products (for example, hardware, software and firmware) used with the product properly exchange accurate date data with it.”

Here are some specific points regarding Report Writer's handling of dates:

- Report Writer's internal system run date includes the correct century as provided to it by the operating system (MVS or VSE). Of course, for this century to be correct after 1999, Report Writer must be running on a version of the operating system that is itself Year 2000 Ready.

- All date fields read from input files are stored internally with 4-digit years. For input date fields that do not contain an explicit century (for example, YYMMDD or YYDDD dates), Report Writer assigns a century for you. If you have not specified a century cutoff year (with the CENTURY Option) all YY input file dates are stored internally as 19YY. If you have specified a century cutoff year, Report Writer stores all dates before your cutoff year as 20YY and all other dates as 19YY.

- All date literals used in Report Writer control statements may be written in either MM/DD/YYYY or MM/DD/YY format. All date literals are stored internally with 4-digit years. When the MM/DD/YY format is used for a date literal, Report Writer assigns a century for you in the same manner as described above (for input file dates.)

  Note: date literals may also be written in DD/MM/YYYY and DD/MM/YY formats if the DDMMYYLIT Option is specified.

- Date comparisons and date computations performed by Report Writer yield the correct result whether the dates are from the 20th century, the 21st century, or any combination of the two.

- By default, all dates that appear in Report Writer reports are formatted in MM/DD/YY format, regardless of their century and regardless of how the date was stored in the input file. However, you can easily change this default and display your dates in any of over 40 different date formats. Any of the date display formats in Appendix B can be used to display any date field, regardless of how that date field was stored in the input file. You can also change the *default* date display format by using the FORMAT Option.

- By default, dates in most Report Writer PC Files appear in MM/DD/YY format. If you want MM/DD/YYYY dates in a PC File, use the FORMAT Option (*afte*r your PC File Option) to specify a different default display format. For example:

  ```
  OPTIONS: LOTUS FORMAT(MM-DD-YYYY)
  ```

How to Prepare for the Year 2000 and Beyond

Like most shops, in the years leading up to 2000 your shop probably is/was engaged in a systematic effort to ensure that all existing jobs continue to work in the year 2000 and beyond. Here are some points that may help you in evaluating your Report Writer jobs.

Q. We are converting some files in our shop by expanding the old 6-byte YYMMDD date fields to 8-byte YYYYMMDD fields. How does this affect our Report Writer jobs?

A. As with any other record layout change, you need to change the Report Writer file definition for the file in question. Change the FIELD statements for the affected date fields to specify the correct new data type. In this example, change the TYPE(YYMMDD) parm to TYPE(YYYYMMDD).

Q. To avoid expanding the size of our records, we are changing our date fields over to a special in-house "compressed" date format that includes century information. Can we use these special date fields with Report Writer?

A. Yes. However, if your date format is not one of those listed in Appendix A, "Data Types", you will need to convert your in-house date field into a standard date value that Report Writer recognizes. How you do this will depend on your particular in-house date format.

For example, some shops have chosen to use the 2-byte character YY portion of their old date fields to hold a 2-byte binary YYYY value (while leaving the MMDD portions of the field in character format.) One way to convert this kind of date is as follows:

```
FIELD:  YYYY  COLUMN(1) LENGTH(2) TYPE(BINARY)
FIELD:  MMDD                      TYPE(CHAR)
COMPUTE: MY-DATE = #MAKEDATE(#FORMAT(YYYY,PIC'9999') + MMDD)
```

The #MAKEDATE built-in function in the COMPUTE statement above takes a character string in YYYYMMDD format and converts it into a true date value. The #FORMAT built-in function was used to convert the 2-byte binary YYYY value into a 4-byte character string.

Other shops are storing dates as a binary or packed number of "days since xx/xx/xx" (where xx/xx/xx is some fixed date.) For example, if your dates are stored as a 2-byte binary "days since 1/1/1950", you could use these statements to convert that field into a standard Report Writer date field:

```
FIELD:   NEW-DATE  COLUMN(1) LENGTH(2) TYPE(BINARY)
COMPUTE: MY-DATE = #MAKEDATE(NEW-DATE + 18262)
```

In the above example, 18,262 is added to the "days since 1/1/1950" value to get the number of days since 1/1/1900, which is what Report Writer's #MAKEDATE built-in function requires for numeric parms.

Note: the #FORMAT and #MAKEDATE built-in functions used in the above examples are explained in Appendix D, "Built-In Functions".

Another way to convert your special date fields into standard date fields is to write a data exit program that Report Writer can call to perform the data conversion. Data exits are discussed in Chapter 5, "How to Define Your Input Files".

Q. Rather than make any changes to our files, we are using a "sliding century" (or "windowing") concept to allow our YY date fields to work past the Year 2000. All dates with years less than 80 will be considered to be 20YY dates. Dates with years equal to or greater than 80 will be considered to be 19YY dates. Can Report Writer accommodate such a scheme?

A. Yes. Just use Report Writer's CENTURY Option. For example, in the particular case you described, you would add this statement near the beginning of your other Report Writer control statements:

```
OPTION: CENTURY(80)
```

That option tells Report Writer that YY dates less than 80 are 20YY and all other dates are 19YY. Note that when the CENTURY Option is used, it is applied to all YY dates encountered in the run. That includes YY dates from all of your input files, as well as any MM/DD/YY date literals found in your control statements.

Q. Some of the YYMMDD dates in my file use a "sliding century" and others do not. What can I do?

A. Since the CENTURY Option applies to all YY dates in a run, you would not use it in this case. However, you can apply your own sliding century logic to individual fields by using COMPUTE statements. For example, assume that you have a YYMMDD field whose cutoff year is 50. You could handle it 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 < '50') ASSIGN(#MAKEDATE('20' + YYMMDD))
                    ELSE                 ASSIGN(#MAKEDATE('19' + YYMMDD))
```

Q. We use Report Writer to create a PC File that we download to use in a Lotus spreadsheet. The dates in that PC File only have 2-digit years. How we can get 4-digit years in our PC File?

A. Use the FORMAT option to specify a different default date display. For example:

```
OPTIONS: LOTUS 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: LOTUS FORMAT(Q-MM-DD-YYYY)
```

Note: be sure that the FORMAT option *follows* the PC format option (LOTUS in the above examples). Otherwise, the PC format option will reset the default date display format.

When using the Q-MM-DD-YYYY format, it is possible that the records in your PC File may now need to be longer than before (since each date field is now 2 bytes longer.) Verify that the record length specified in your execution JCL is still large enough to contain all of your output fields. You can quickly determine this by running a test job and looking for "truncation" warning messages in the control listing. If you get truncation warning messages, increase the record length in your execution JCL (see pages 362 and 374.)

Appendix K. I/O Exits

Report 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
- files containing a number of "segments" (or array elements) that you wish to "normalize". That is, your exit can return more than one logical record to Report Writer for each physical record present in the file.

Report 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 Report Writer. That is, they can successfully use the KEYRANGE, MULTI, KGE, GENERIC and DDNAME/DLBL parms in the normal way within their INPUT or READ statements. To the end-users, your exit-type files will look just like any other file.

Report Writer also passes your exit program an optional, user-defined parm text 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 MVS, you can specify this with a LRECL parm (as shown above) or omit it and use Report 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 Report 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 Report 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 record. 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), Report Writer loads two copies of the exit program -- one for each input record.

When Is the I/O Exit Called?

The I/O Exit for an input is called each time Report Writer needs to obtain a record from that input. In other words, the exit is called at the same times that Report Writer would, for a nonexit-type input, issue its own I/O request. In addition, Report 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 Report Writer calls it to obtain a record.

Following is a more detailed explanation of when Report Writer reads records from different kinds of inputs.

For **the primary input** (named in the INPUT statement), Report 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 Report Writer. For primary input files, Report 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,** Report Writer calls the I/O Exit the first time it needs a field from a new auxiliary input record. When subsequent fields from the same input record are needed, Report Writer will not call the I/O Exit again, since the record is already available for it to use. For non-MULTI inputs, Report Writer calls the I/O Exit a maximum of one time per primary input file record. (Report 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, Report 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 Report 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 Report 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 Report 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 Report 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, Report Writer stops processing that input and does not call that I/O Exit any more.

What Does Report 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 628.) The contents of the $IX DSECT will have been set correctly by Report Writer, as described below. Register 13 points to an 18-fullword save area within Report Writer which the I/O Exit should use to save Report Writer's registers. Register 14 contains the return address within Report Writer. Register 15 contains the entry point address of the I/O Exit.

Report Writer always runs in 24-bit addressing mode. Therefore, the I/O Exit program will be called in 24-bit address mode and must return to Report Writer in the same mode.

Note the $IX DSECT located near the end of the sample program. That DSECT shows the complete parm list that Report Writer passes to all I/O Exit programs. Following is a description of each item in the $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 Report 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.
FRST	read the first 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 MULTI-type 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 for any exit-type file used in a MULTI-type READ statement.

CLOS perform any close-type processing that may be required. Report 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 Report Writer.

$IXRECNM This 70-byte character fields 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 fields 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 specified in the IOEXIT parm. 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 Report 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. 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, 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.

$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.

> **Note:** 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 Report Writer does not perform any validity-checking on the readkey's length (since Report 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 Report Writer. Report Writer will print your error message for the user and stop processing the file.

$IXPRMLN This halfword contains the length of the parm text (from the IOEXIT parm) that appears at the address contained in $IXPRMAD, if any.

$IXRECLN This halfword contains the length of the I/O area reserved for the exit program at the address contained in $IXRECAD.

$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 Report Writer does not perform any sort of validity-checking on the length of the beginning keyrange value (since Report 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 Report Writer does not perform any sort of validity-checking on the length of the ending keyrange value (since Report Writer knows nothing about your file's structure.)

$IXRETCD This halfword must be set by the I/O Exit program before it returns to Report Writer after each call. The following list shows the valid values for $IXRETCD. If $IXRETCD contains any other value upon return to Report 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 Report Writer sees a return code of 12 for an input file, it does not attempt any further processing of that input.

$IXDDN	For MVS, this 8-byte character fields contains the value of the DDNAME parm, if any, being used for the input file. For VSE, this field contains the DLBL/TLBL value (from the ATTR parm), if any, being used for the input 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 record.
$IXGEN	This 1-byte character field contains a Y if the user specified the GENERIC parm in the READ statement for this input record.
$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 record.
$IXUSER	This 50-byte, doubleword aligned area is available for the exit program to use any way it wishes. The area is initialized by Report Writer to hex zeros before the first call. Thereafter, Report 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. Report Writer initializes this field to all spaces. Upon return to Report Writer, if the first byte of this field is non-blank, Report Writer prints the contents of this field as a Warning-level message in the control listing and blanks the field out again.
$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 $IXFUNC
- the return code, which is initialized to -1 by Report Writer before each call.
- the error message area ($IXERR) is reset to blanks each time it is used.

What Does the I/O Exit Pass Back to Report Writer?

Before returning to Report 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 (MVS) 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).

Caution: if your exit program fails to ever indicate EOF (via return code 4), Report 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 Report 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. A copy of this program is contained in the sample Copy Library found in your installation tape.

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 reduces 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 and use the excess portion of the I/O area (beyond your record) to hold debug values. The entire I/O area is printed in the TRACE output.

Sample I/O Exit Program Written in Assembly Language — 1 of 7

```
IOEXITA  TITLE '- SAMPLE REPORT 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 REPORT 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 REPORT 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
*
         CLC   $IXFUNC,=CL4'KEY '  DOES CALLER WANT A KEYED READ?
         BE    DOKEY               YES - DO KEYED IO LOGIC
*
```

Sample I/O Exit Program Written in Assembly Language — 2 of 7

```
         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
*
         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
*
*
```

Sample I/O Exit Program Written in Assembly Language — 3 of 7

```
**********************************************************************
*  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   $IXERR+26(4),FEEDBACK USER MUST VIEW THIS IN HEX
         B     RETERROR            RETURN WITH ERROR RETCODE
*
*
**********************************************************************
*  DO READ-FIRST OF EMPLOYEE FILE                                    *
**********************************************************************
DOFIRST  EQU   *
         CLI   MULOPEN,C'Y'        HAVE WE OPENED THE MULTIACB YET?
         BE    MULISOPN            B IF YES - DON'T OPEN IT AGAIN
*
**********************************************************************
*  DO ONE-TIME STUFF ON FIRST CALL.  OPEN ACB AND MODIFY THE RPL.    *
**********************************************************************
         MVI   MULOPEN,C'Y'        REMEMBER FILE HAS BEEN OPENED
         MVC   MULNAME,$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
*
```

Sample I/O Exit Program Written in Assembly Language — 4 of 7

```
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
*
*
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
*
*
```

Sample I/O Exit Program Written in Assembly Language — 5 of 7

```
**********************************************************************
*  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
*
*
***********************************************************************
*   CLOSE ONE OF REPORT 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
*
*
```

Sample I/O Exit Program Written in Assembly Language — 6 of 7

```
*
*
***********************************************************************
*   RETURN TO REPORT 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 REPORT WRITER
*
*
OURSAVE  DC    18F'0'              OUR SAVE AREA
*
FEEDBACK DS    F                   HOLDS FEEDBACK INFO FROM RPL
*
*
***********************************************************************
*   DATA READ FOR SEQUENTIAL FILE I/O                                 *
***********************************************************************
SEQOPEN  DC    C'N'                FLAG - WHETHER SEQ ACB IS OPEN YET
SEQNAME  DC    CL70' '             REPORT 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' '             REPORT 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' '             REPORT 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
```

Sample I/O Exit Program Written in Assembly Language — 7 of 7

```
*********************************************************************
*                                                                   *
*        $IX --  PARM DSECT FOR CALLING USER I/O EXIT (FOR INPUT)      *
*                                                                   *
*********************************************************************
$IXDSECT DSECT ,                   IO EXIT (INPUT) PARM DSECT
$IX      DS    OD
*
$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.                 *
*    "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"  -- REPORT 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    OF                  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. REPORT 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    OD                  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
```

Updates to This Manual

To Keep Your Manual

Current, Please File All

Updates Behind This Page.

(This page left blank intentionally.)

Index