z/Writer

Quick-Code Tool for z/OS

Native Reference Manual

Our Customers Talk About Our Products...

"Everything we hoped it would be. In fact, I can honestly say it exceeded our initial expectations."

"I can create reports in a few minutes. I don't know what I'd do without it."

"Thanks again for the information. We were able to download the evaluation copy, and I have found it super easy to use. To me, it is exactly the right tool for us to use."

"It is easy to use, flexible and has all the features needed to produce virtually any type of report."

"The syntax is straightforward, the documentation is good, the support is good."

"Since I only used the product today and saw how easy it was to generate these reports, it would save me a lot of time from writing assembler routines to do this. Hopefully I can get this point across to management."

"A really great product for an old bit-flipper that finds himself in a management job ;-)"

"I REALLY like your product and I am trying my best to prove how valuable that this product could be for us at such a low cost. For the couple of times that I have used the product in testing to produce certain results, I was able to get the information I needed without a large effort and time on my part. That is what I liked. I didn't have to create a new assembler program to do this work."

... and Our Customer Service

"You are truly a breath of fresh air in an industry that has level1 level2 level3 support. I have been twiddling bits in the mainframe for 40 years and the last 10 have been scarce to find support like you have provided. Being able to debug without the data is a truly scarce talent."

"Thanks for getting back to me so quickly! ... By the way, you have a terrific product."

"As I mentioned before, I really appreciate all of your help, and wish that all of the vendors with whom I work were half as pleasant and helpful as you have been. It has been a pleasure."

"Thanks [for checking] but it is all working perfectly. Really pleasantly surprised."

"Wow!! Excellent turn around time! Thank you!"

"That's perfect! I knew it could be done but I just couldn't find it. Thanks so much for your help. By the way, I love this product. I'm just getting my feet wet with it but I can see a lot of potential for our shop."

6th Edition. z/Writer Release 2.0 (175)

Pacific Systems Group, LLC
501 Fourth St. #790
Lake Oswego OR 97034

1-503-675-5982
pacsys.com

Table of Contents

Who Is This Manual For?

This manual is intended for users of our z/Writer product without any special filter. We call that “native z/Writer.”

We also have several sister z/Writer products that do include special built-in filters, The filters allow z/Writer to execute certain CA programs without any conversion at all. (Currently, there are versions that can emulate DYL-280, Easytrieve, Quikjob and CA-Earl.) These specialized products each have their own manuals.

This manual is intended specifically for those users who are not replacing an existing quick-code product from CA.

Chapter 1. What is z/Writer?

z/Writer is a powerful quick-code tool for z/OS mainframes: Its “simplified COBOL-like” syntax makes it an ideal tool for writing:

- one-time queries
- quick-and-dirty reports or analyses.
- attractive new production-quality reports.
- file manipulations, for scheduled production runs or one-time updates or conversions.
- exporting mainframe data as a CSV file to work with in PC programs.

Your programmers will be up and running with z/Writer in no time at all. z/Writer also accepts existing COBOL or Assembler record definitions so you can get started using it right away.

z/Writer is a powerful quick-code tool for z/OS mainframes. Its “simplified COBOL-like” syntax makes it an ideal tool for quickly writing new reports, modifying files and much more.

z/Writer Features

Some of z/Writer's major features include:

- control statements use an easy, free format, COBOL–like syntax that's quickly learned
- user–friendly field names can be up to 70 characters long (unlike some report writers that restrict you to cryptic 8–byte names). This also allows full compatibility with existing COBOL, PL/1 and Assembler data names.
- you can easily work with any number of flat files and VSAM files
- access DB2 tables, with our available DB2 Option
- access IMS databases, with our available IMS Option
- export function formats data as comma delimited file for use on PCs
- create and read back temporary work files, without any JCL changes
- use your existing COBOL or Assembler record layouts instead of creating a data dictionary. Or, use z/Writer's simple data dictionary for added functionality.
- produces efficient internal machine code that is easy on your CPU
- produces multiple reports (or output files) in a single file pass
- macro expansion feature to help in supplying run-time parms
- floating point data support
- report lines are not limited to only 132 characters. z/Writer can format a report as wide as your laser printer supports.
- any number of control breaks allowed

- ability to print full–page forms
- full control of carriage control when printing reports
- allows complete control over formatting of numeric fields, including handling of special cases like telephone numbers, social security numbers, etc.
- has special formatting options for international users
- allows complete control over report titles, column headings
- includes thorough, clear documentation
- includes an option to validity–check numeric data before processing it, to prevent S0C7 abends
- ability to display file data in hexadecimal format, for analyzing invalid data
- full program trace facility, to help when developing new program
- translates fields from EBCDIC to ASCII and vice verse
- supports full "boolean logic" (the use of AND, OR and NOT) in conditional expressions
- ability to scan free format fields, to see if a certain text appears anywhere within the field
- comparisons and computations are allowed among *all* numeric fields, (even if some are packed, some are binary, and others are zoned, etc.)
- supports all types of mainframe data widely in use, including packed, BCD, signed and unsigned binary and hexadecimal floating point
- full mathematical calculations are supported, including the use of many built–in functions
- supports a full range of functions to manipulate string data, including powerful parsing features
- "compress" formatting features lets you, for example, compress separate city, state and ZIP fields into a normal formatted line format
- handles complicated record layouts, including variably–located fields, fields located by pointer or pointer expressions, etc.
- supports records that contain arrays, nested to any level
- allows an unlimited number of input files
- built–in fields provide the system date, time, jobname, etc.
- can halt input processing when a user-defined condition is met, to eliminate unnecessary I/O

Chapter 2. How to Make Reports with z/Writer

This chapter gives you a quick introduction to writing z/Writer programs

z/Writer has two programming modes. The one you choose will depend on what you need z/Writer to do.

- **Auto-Cycle Mode.** In this mode, z/Writer automatically handles the primary file I/O, report formatting, and, optionally, control breaks and subtotals and grand totals.
- **Standard Mode.** In this mode, you have complete control on what the program does and how it does it.

We will look at a simple example of each run mode next. In the following chapter, we will examine more complex examples of each mode.

z/Writer's Auto-Cycle Mode

Use auto-cycle mode if you need to print a relatively simple report from an input file. In this mode, z/Writer reads sequentially through your whole primary input file, record by record. You can choose which records to include in your report, and you can print whichever fields you want from them. z/Writer then formats the raw data into an attractive report for you automatically.

In this mode, you do not need to code any READ statements for your input file. Nor do you need to code a test for EOF on that file. The statements that you code in the report program are simply executed, from the beginning, for each input record that z/Writer reads. In your program code, you will take care of any data manipulation or computation logic that may be needed. And you will normally then print a line to a report (using a PRINT statement). Or you could export it as a comma-delimited file (using EXPORT and PRINT statements.) Or perhaps you will want to format a new mainframe record and write that record to an output file with a WRITE statement.

Even though this feature is designed for "basic" reports, you can still do all of the following in an auto-cycle report:

- sort the input file
- read through the (sorted) input file until it reaches EOF
- include only selected records from the file in the report and the report totals
- print whatever fields you want in the report
- break and print subtotals (to any level), as well print grand totals

So this actually covers quite a few reports.

The auto-cycle execution mode is automatically invoked whenever a report has a primary input file, but the program code does not have any READ statements for it.

The **primary input file** is the first input (or update) type file defined in your program (for a given phase). If you program has multiple phases (see page 14), each phase can run as a separate auto-cycle report, if you so choose.

This Job:

```
//ZWRITER JOB
//ZW EXEC  PGM=ZWRITER
//STEPLIB  DD DSN=PROD.ZW200.LOADLIB,DISP=SHR
//SALES    DD DSN=PROD.SALEFILE,DISP=SHR       INPUT FILE
//ZWOUT001 DD SYSOUT=*   ZWRITER REPORT
//SYSPRINT DD SYSOUT=*   ZWRITER CONTROL LISTING
//SYSOUT   DD SYSOUT=*   SORT MESSAGES
//SYSIN DD *             PROGRAM STATEMENTS
FILE SALES
*
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX

TITLE 'SALES REPORT'

//
```

Produce this Report:

```
09/20/12            SALES REPORT            PAGE    1

       EMPL    EMPL    SALES  SALES
REGION NUM     NAME     DATE   TIME   AMOUNT    TAX

SOUTH  037  JOHNSON    950312 102500    101.38   6.09
WEST   044  BAKER      950326 120909    137.00   8.22
EAST   042  MORRISSOHN 950329 153022     44.35   2.66
EAST   042  MORRISSOHN 950330 190541     29.65   1.78
EAST   041  SIMPSON    950401 081757     14.99   0.90
NORTH  039  JOHNSON    950401 170247    234.45  14.07
NORTH  039  JOHNSON    950405 143310      9.98   0.60
WEST   044  BAKER      950412 143112    135.75   8.15
WEST   045  THOMAS     950414 154138      9.98   0.60
NORTH  036  JONES      950415 075832     10.25   0.62
NORTH  036  JONES      950415 080159    121.76   7.31
NORTH  036  JONES      950415 135241     10.25   0.62
SOUTH  037  JOHNSON    950416 114833    500.00  30.00
EAST   041  SIMPSON    950430 153021     23.87   1.43

GRAND TOTAL                          1,383.66  83.05
```

Figure 1. A Simple z/Writer Auto-Cycle Report, with JCL

For auto-cycle mode, the essential elements that you should code are:

- one FILE statement and multiple FIELD statements to define your primary input file
- a PRINT statement to specify how your report body should look

- a TITLE statement to specify the title for the report

Of course, you will also use any other statements needed to perform any special logic or computations required by your report.

> **Note:** you will learn the syntax and usage details of each z/Writer statement in Chapter 5. z/Writer Control Statements (page 25).

Figure 1 shows a very simple auto-cycle report. You can see that z/Writer printed one report line for each input record. It also completed the report title and printed column headings and grand totals automatically.

z/Writer's Standard Mode

Now let's look at a simple standard mode report.

Use z/Writer's **standard mode** if you want complete control of the program flow. With this mode, you can do anything a COBOL program could do, but using much simpler code.

In standard mode, z/Writer does not perform any automatic I/O or totaling. (It will, however, complete your report title and prepare column headings for you, unless you request otherwise.) In this mode, you must execute a READ statement each time you want to read a record from the primary input file.

As long as your code includes at least one READ statement for the primary (first) input file, z/Writer will execute your program in standard mode. That is, it passes control to the first executable statement in your program. From then on, your program controls the flow of execution (using GOTOs, DOWHILE loops, etc.) The program will end when the flow reaches the end of the executable statements, or when a STOP statement is encountered.

Figure 2 shows a simple example of a standard mode program. This report is similar to the report in Figure 1 (page 10). However, in this example we coded the READ statements and checked for EOF ourself. Also notice that there are no grand totals in this report. z/Writer only performs automatic grand totals in auto-cycle reports. In standard mode reports, your code must handle the accumulation and printing of any totals (or other statistics) that you want.

Order of Statements In Your Program

Whichever execution mode you prefer, there are a few rules regarding the location of certain statements. In general, non-executable statements should come at the beginning or the end of the program (depending on the statement.) They should not be mixed in among the executable statements.

You should generally put your program statements in the following order.

1. First put any **OPTION statements** you may need. These statements specify special options that apply to all phases in the execution (if there are more than one.)

2. Then put the **REPORT statement** for your first (or only) report. This statement specifies report options that apply only to the primary report in the current phase. If you have other REPORT statements (for additional reports in the same phase), you may want to put them here as well. (However, that is not a requirement.)

3. Next come all statements that define inputs, outputs and working storage areas. This means all **FILE, TABLE** and/or **WORKAREA statements**, along with the **FIELD statements** that follow those statements.

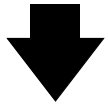
These Control Statements:

```
FILE SALES
*
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

READ SALES

DOWHILE (#EOF = 'N')
   PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX
   READ SALES
ENDDO

TITLE 'SALES REPORT'
```



Produce this Report:

```
09/20/12            SALES REPORT            PAGE    1

       EMPL    EMPL    SALES  SALES
REGION NUM     NAME     DATE   TIME   AMOUNT   TAX

SOUTH  037  JOHNSON    950312 102500    101.38   6.09
WEST   044  BAKER      950326 120909    137.00   8.22
EAST   042  MORRISSOHN 950329 153022     44.35   2.66
EAST   042  MORRISSOHN 950330 190541     29.65   1.78
EAST   041  SIMPSON    950401 081757     14.99   0.90
NORTH  039  JOHNSON    950401 170247    234.45  14.07
NORTH  039  JOHNSON    950405 143310      9.98   0.60
WEST   044  BAKER      950412 143112    135.75   8.15
WEST   045  THOMAS     950414 154138      9.98   0.60
NORTH  036  JONES      950415 075832     10.25   0.62
NORTH  036  JONES      950415 080159    121.76   7.31
NORTH  036  JONES      950415 135241     10.25   0.62
SOUTH  037  JOHNSON    950416 114833    500.00  30.00
EAST   041  SIMPSON    950430 153021     23.87   1.43
```

Figure 2. A Simple Standard Mode Program

For DB2 runs, also put your **CURSOR** statements in this are. And for IMS runs, also put the **IMSFILE** and **SEGMENT** statements here.

4. Next comes all of the **actual executable code** for your specific program. For auto-cycle reports, this is the code that will be executed for each input record read. For standard mode runs, this is the code that will be executed one time -- from the first statement until the last program statement (or until encountering a STOP statement.) This is where all executable statements (such as IF, PRINT, PERFORM, MOVE, GOTO, and so on) will go.

Within the executable user code, be careful where you place the non-executable BREAK and TITLE statements, in relation to the first PRINT statement. Follow these rules:

- All **BREAK statements** must *precede* the first PRINT statement for a report.
- All **TITLE statements** must be coded *after* the first PRINT statement for a report.

Note: these requirements ensure that the titles and break total lines are aligned correctly with the body of your report.

The program in Figure 3 (page 17) illustrates this order of statements (as do the other examples.)

Report Column Headings

By default, z/Writer prints column headings for any report (auto-cycle or standard mode) that uses at least one PRINT statement. The column headings are printed at the top of each page of the report, right after all of the title lines. Column headings are printed for each field listed in the *first* PRINT (or PRINTMODEL) statement in the program.

The column heading printed for each column is taken (in this order of preference) from:

- override column heading text specified directly in the first PRINT statement
- column heading text specified in the HEADING parm of the FIELD statement
- the fieldname itself, broken apart at each dash or underscore

Suppressing Column Headings

You can suppress column headings from a report by specifying the NOCOLHDG report parm, like this:

```
REPORT NOCOLHDGS
```

You might want to suppress the automatic column headings if you plan to build your own column heading lines using TITLE statements.

Printing Multiple Reports

z/Writer allows you to print a (theoretically) unlimited number of reports during your program execution. This has the advantage of only reading an input file once to create multiple reports.

Notice that *only three* statements directly control the appearance of a report:

- REPORT statement (which is optional for the primary report in a program). This non-executable statement describes overall characteristics of the report (such as page size and whether column headings are wanted, for example.)
- PRINT statement. This executable statement writes one line to the report output. It tells which fields to print in the body of the report.
- TITLE statement (optional for all reports). This non-executable statement defines how the title(s) for a report will look. z/Writer decides when the titles are actually written, but the TITLE statement determines the contents and layout of the report titles.

So, if you want to print more than one report, you will just make an adjustment to each of these three statements. Specifically, you will just include a report-name, in parentheses, as the first parm in each of these 3 statements. Here are the steps to write a new report from your program:

- add a new REPORT statement that specifies a 1- to 8-byte "report name" that you want to use for it. The report name must be enclosed in parentheses and must be the first parm in the statement. After the report name, you can add any other REPORT statement options that you want for the new report. For example, to define a new "error" report named ERRRPT, you could code:

```
REPORT (ERRRPT) NOCOLHDGS
```

- to write lines to the new report, use a PRINT statement that has the new report name, in parentheses, as its first parm. Example:

```
PRINT (ERRRPT) EMPL_NAME STATUS
```

- optionally, you can add one or more TITLE statements for the new report. The TITLE statement will also begin with the new report name, in parentheses, as its first parm. Example:

```
TITLE (ERRRPT) 'EMPLOYEES WITH INVALID STATUS CODES'
```

- add a DD to your execution JCL for the new report. The DDNAME is the same as the report name. Example:

```
//ERRRPT DD SYSOUT=*
```

The "report name" (ERRRPT in this case) is what links the REPORT, PRINT and TITLE statements for a given report to each other. And it links the report to its output DD in your JCL.

You may specify as many different report names in your REPORT, PRINT and TITLE statements as you like. Just be sure to add a DD statement for each report to your JCL.

By the way, all PRINT, TITLE and REPORT statements that do *not* begin with a report name parm (in parentheses) apply to the first, or "primary" report for the phase. The primary report does not have a name. It uses the DDNAME ZWOUT001 (for the first phase.)

Note: certain auto-cycle features, such as automatic control break handling via the BREAK statement, are supported only for the primary report in each phase.

Note: if multiple phases are used, the report names you choose must be unique across all phases.

Runs with Multiple Phases

Within a single execution of the z/Writer program, you may code multiple "phases". A new phase is almost like starting z/Writer again in a new jobstep. When you start a new phase, *nothing* is retained from any earlier phase. In the new phase, you will again define each file you that wish to use, any needed working storage fields, etc., and then you will code your execution logic for that phase. (However, z/Writer does offer the REUSE statement (page 135) that makes it easy to quickly define again any file that was defined in an earlier phase.)

To add additional phases (after the first one), use the NEWPHASE statement (page 97). This statement ends the previous phase and begins a fresh, new phase.

One use of multiple phases is to perform file processing logic that can not be done in a single "step". For example, you may need an initial phase to read a file and write "summary" records to a z/Writer TEMP file. The next phase might then read this "summary" TEMP file and perform further processing on it. Of course, you could do this in 2 separate JCL jobsteps. But if you prefer to keep the jobstream simple, you can use phases to perform both steps in a single execution of z/Writer.

The **“primary report”** output for each phase is written to its own DD. The first phase’s report is written to the ZWOUT001 DD. The second phase’s report is written to the ZWOUT002 DD, and so on.

In addition to a primary report, each phase is also allowed to produce any number of additional reports. See "Printing Multiple Reports" (page 13).

Chapter 3. Sample z/Writer Reports

Programmers often pick up new techniques best by browsing examples. With that in mind, we will look at a variety of z/Writer programs in this chapter.

Auto-Cycle Report with Control Breaks

We saw earlier in Figure 1 (page 10) that auto-cycle reports automatically get grand totals. (You can suppress them, if you want, with a NOGRAND parm on a REPORT statement.)

Auto-cycle reports can also include any number of additional control breaks, with automatic subtotals at each break. To add a control break, you should:

- add a PRESORT parm to the FILE statement for the primary input file (unless the input file is already sorted in the proper order)
- add one or more BREAK statements to specify which sort field(s) to break on. If you have multiple BREAK statements, put the BREAK statement for the high-order break field first. (That is the break field that appeared earliest in the SORT parm.) All BREAK statement should appear before your first PRINT statement.

Figure 3 shows an example of an auto-cycle report with two levels of control breaks. z/Writer maintained and printed subtotals for each level of break (plus the grand total) automatically.

Note: we used a "(+6)" parm for REGION in the PRINT statement in this example. That parm shifted the REGION column right by 6 bytes (from its default location of column 1.) We did that in order to leave room for the word "TOTAL" on the total lines, without overlapping the region value. Without this extra space, the word TOTAL would have been printed on a separate line, above the line containing the region name and the actual numeric totals. The "+6" parm just helps keep the report a little easier to read, and somewhat shorter.

Note: the BREAK statement is only available for auto-cycle reports.

Auto-Cycle Report with Record Selection

Our examples so far have printed a report line for *every* record in the input file. z/Writer also provides a method for selecting only certain records to include in an auto-cycle report.

Of course, you can always use an IF statement to prevent the PRINT statement from being executed under certain conditions. But if that is all that you do, the record would *still* be included in the grand totals (and in any subtotals for control breaks.)

Use the EXCLUDE and/or INCLUDE statements to tell z/Writer whether to include a record's values in the totals it maintains. EXCLUDE means do not include the record in the totals. INCLUDE means include the record in the totals. (Of course, including the record is also the default, when neither an INCLUDE nor EXCLUDE statement is executed.)

Both INCLUDE and EXCLUDE also cause the remainder of the program code (from that statement to the end of the program) to be skipped over for the current input record.

These Control Statements:

```
FILE SALES
     PRESORT(REGION EMPL_NAME)
*
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

BREAK REGION
BREAK EMPL_NAME

PRINT REGION(+6) EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX

TITLE 'SALES REPORT'
TITLE 'TOTALED BY REGION AND EMPLOYEE'
```

Produce this Report:

```
09/20/12              SALES REPORT               PAGE    1
              TOTALED BY REGION AND EMPLOYEE

              EMPL    EMPL    SALES  SALES
      REGION  NUM     NAME    DATE   TIME   AMOUNT   TAX

      EAST    042   MORRISSOHN 950329 153022     44.35   2.66
      EAST    042   MORRISSOHN 950330 190541     29.65   1.78
TOTAL EAST          MORRISSOHN                   74.00   4.44

      EAST    041   SIMPSON    950401 081757     14.99   0.90
      EAST    041   SIMPSON    950430 153021     23.87   1.43
TOTAL EAST          SIMPSON                      38.86   2.33

TOTAL EAST                                      112.86   6.77

      NORTH   039   JOHNSON    950401 170247    234.45  14.07
      NORTH   039   JOHNSON    950405 143310      9.98   0.60
TOTAL NORTH         JOHNSON                     244.43  14.67

      NORTH   036   JONES      950415 075832     10.25   0.62
      NORTH   036   JONES      950415 080159    121.76   7.31
      NORTH   036   JONES      950415 135241     10.25   0.62
TOTAL NORTH         JONES                       142.26   8.55

TOTAL NORTH                                     386.69  23.22

(additional lines not shown)

      WEST    045   THOMAS     950414 154138      9.98   0.60
TOTAL WEST          THOMAS                       9.98   0.60

TOTAL WEST                                      282.73  16.97
```

Figure 3. An Auto-Cycle Report with Two Levels of Control Breaks

Auto-Cycle Report with Record Selection

Figure 4 uses the EXCLUDE statement to exclude records with sales amounts under $100. The EXCLUDE statement causes the PRINT statement to be skipped. It also excludes those records from all report totals.

Figure 8 (page 24) shows another example of using the INCLUDE and EXCLUDE statements.

Note: the INCLUDE and EXCLUDE statements are only available for auto-cycle reports.

These Control Statements:

```
FILE SALES
*
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

IF AMOUNT < 100
    EXCLUDE
ENDIF

PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX

TITLE 'SALES REPORT'
TITLE 'FOR SALES OVER $100'
```

Produce this Report:

```
09/20/12            SALES REPORT            PAGE    1
                  FOR SALES OVER $100

       EMPL    EMPL    SALES  SALES
REGION NUM     NAME     DATE   TIME   AMOUNT    TAX

SOUTH  037  JOHNSON    950312 102500    101.38   6.09
WEST   044  BAKER      950326 120909    137.00   8.22
NORTH  039  JOHNSON    950401 170247    234.45  14.07
WEST   044  BAKER      950412 143112    135.75   8.15
NORTH  036  JONES      950415 080159    121.76   7.31
SOUTH  037  JOHNSON    950416 114833    500.00  30.00

GRAND TOTAL                           1,230.34  73.84
```

Figure 4. An Auto-Cycle Report with Record Selection

A Sequential File Conversion Program

It is easy to perform file conversions with z/Writer. Just define the input and output files, add code to prepare your output record (from the input record), and then use a WRITE statement to write each record to the output file.

Figure 5 shows an example of an auto-cycle file conversion program. It writes out a new file that is just like the old file, except that the amount field is increased by 10. It also prints a log report showing the changes made.

Making a Comma-Delimited Export File

z/Writer automates the process of making export files from your mainframe data. These are comma delimited files that can easily be imported into PC programs. This opens up the possibility of working with mainframe data in PC spreadsheets, databases, graphing programs, etc.

Use an EXPORT statement (page 63) to indicate that you will be creating an export file output in your run. (The EXPORT statement is similar to the REPORT statement, which defines a report output.) The EXPORT statement assigns a name to your export file. The name will also be the DDNAME of the output DD in your JCL. Optionally, you can also specify certain options for your export file on this statement.

To write data to your export file, just use a standard PRINT statement, specifying the export file as the "report name." The fields (or literals) specified in the PRINT statement will be formatted into an output record in comma delimited format:

- character (and date) data will be enclosed in parentheses
- numbers will be formatted without commas
- the fields will be separated with commas.

Also note these differences between an export file and a report:

- export files do not have titles
- the column headings for export files only appear once, at the beginning of the export file. (You can also specify NOCOLHDGS in your EXPORT statement if you do not want any column headings.)

Figure 6 shows an example of making an export file.

Keyed Reads to a VSAM file

In this example, we read sequentially through a flat file (SALES). We let auto-cycle handle that part for us. Then, for each record automatically read from that SALES file, we perform a direct read to the EMPL file. This is a keyed (KSDS) VSAM file.

The key to the EMPL file is a 3-byte employee number. The SALES file also has an employee number field. So, we simply use the employee number from the primary input file (SALES) as they key to read a record from the secondary file (EMPL).

The PRINT statement can now include fields from both the SALES record and the EMPL record. So we printed mostly data from the SALES file in our report. But we did include the Department Number and Total Accounts data from the EMPL file.

These Control Statements:

```
FILE SALES
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

FILE NEWSALES OUTPUT
FLD NEW_EMPL_NAME       10
FLD NEW_EMPL_NUM        3
FLD NEW_BACKUP_EMPL_NUM 3
FLD NEW_REGION          5
FLD NEW_AMOUNT          N6.2
FLD NEW_TAX             N4.2
FLD NEW_COMMISSION_RATE N4.3
FLD NEW_SALES_DATE      6
FLD NEW_SALES_TIME      6

MOVE SALES TO NEWSALES              /* COPY WHOLE RECORD         */
NEW_AMOUNT = AMOUNT +10             /* MODIFY AMOUNT IN NEWSALES */

WRITE NEWSALES                      /* WRITE NEW RECORD          */

PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME
      AMOUNT NEW_AMOUNT TAX          /* PRINT A LINE IN LOG      */

TITLE 'FILE CONVERSION LOG'
```

Produce an Output File and this Log Report:

```
09/20/12              FILE CONVERSION LOG             PAGE    1

       EMPL    EMPL    SALES  SALES               NEW
REGION NUM      NAME    DATE   TIME   AMOUNT    AMOUNT    TAX

SOUTH  037  JOHNSON    950312 102500    101.38    111.38   6.09
WEST   044  BAKER      950326 120909    137.00    147.00   8.22
EAST   042  MORRISSOHN 950329 153022     44.35     54.35   2.66
EAST   042  MORRISSOHN 950330 190541     29.65     39.65   1.78
EAST   041  SIMPSON    950401 081757     14.99     24.99   0.90
NORTH  039  JOHNSON    950401 170247    234.45    244.45  14.07
NORTH  039  JOHNSON    950405 143310      9.98     19.98   0.60
WEST   044  BAKER      950412 143112    135.75    145.75   8.15
WEST   045  THOMAS     950414 154138      9.98     19.98   0.60
NORTH  036  JONES      950415 075832     10.25     20.25   0.62
NORTH  036  JONES      950415 080159    121.76    131.76   7.31
NORTH  036  JONES      950415 135241     10.25     20.25   0.62
SOUTH  037  JOHNSON    950416 114833    500.00    510.00  30.00
EAST   041  SIMPSON    950430 153021     23.87     33.87   1.43

GRAND TOTAL                          1,383.66  1,523.66  83.05
```

Figure 5. A Sequential File Conversion Program

These Control Statements:

```
FILE SALES
FLD EMPL_NAME       10
FLD EMPL_NUM        3
FLD BACKUP_EMPL_NUM 3
FLD REGION          5
FLD AMOUNT          N6.2
FLD TAX             N4.2
FLD COMMISSION_RATE N4.3
FLD SALES_DATE      6
FLD SALES_TIME      6

EXPORT

PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX
```

Produce This Comma Delimited Export File:

```
" ","EMPL","EMPL","SALES","SALES"," "," "
"REGION","NUM","NAME","DATE","TIME","AMOUNT","TAX"
"SOUTH" ,"037" ,"JOHNSON   ","950312","102500",   101.38,   6.09
"WEST " ,"044" ,"BAKER     ","950326","120909",   137.00,   8.22
"EAST " ,"042" ,"MORRISSOHN","950329","153022",    44.35,   2.66
"EAST " ,"042" ,"MORRISSOHN","950330","190541",    29.65,   1.78
"EAST " ,"041" ,"SIMPSON   ","950401","081757",    14.99,   0.90
"NORTH" ,"039" ,"JOHNSON   ","950401","170247",   234.45,  14.07
"NORTH" ,"039" ,"JOHNSON   ","950405","143310",     9.98,   0.60
"WEST " ,"044" ,"BAKER     ","950412","143112",   135.75,   8.15
"WEST " ,"045" ,"THOMAS    ","950414","154138",     9.98,   0.60
"NORTH" ,"036" ,"JONES     ","130415","075832",    10.25,   0.62
"NORTH" ,"036" ,"JONES     ","130415","080159",   121.76,   7.31
"NORTH" ,"036" ,"JONES     ","130415","135241",    10.25,   0.62
"SOUTH" ,"037" ,"JOHNSON   ","950416","114833",   500.00,  30.00
"EAST " ,"041" ,"SIMPSON   ","950430","153021",    23.87,   1.43
"GRAND TOTAL"," "," "," "," ",  1383.66,  83.05
```

Figure 6. Making a Comma Delimited Export File

Note that if a field of the same name exists in both records (like EMPL_NUM) it must be qualified (in the READ and PRINT statements.) We wanted to use the EMPL_NUM field from the SALES file as the key value in our READ statement, so we qualified it thus:

```
SALES.EMPL-NUM
```

Figure 7 shows the resulting report.

These Control Statements:

```
COPY SALES

FILE EMPL TYPE(KSDS) V(150)
FLD EMPL_NUM   3
FLD LAST_NAME   15
FLD FIRST_NAME  15
FLD HIRE_DATE   N6
FLD DEPT_NUM    1
FLD SEX         1
FLD STATUS_BYTE 1
FLD SOCIAL_SEC_NUM   N9
FLD NUM_ACCOUNTS     N4
FLD TOTAL_SALES      N7.2
FLD SALES_QTR1       N7.2
FLD SALES_QTR2       N7.2
FLD SALES_QTR3       N7.2
FLD SALES_QTR4       N7.2
FLD ADDRESS         20
FLD CITY            15
FLD STATE           2
FLD ZIP             5
FLD TELEPHONE       10

READ  EMPL  KEY(SALES.EMPL_NUM)

PRINT REGION SALES.EMPL_NUM DEPT_NUM EMPL_NAME
      NUM_ACCOUNTS SALES_DATE SALES_TIME AMOUNT TAX

TITLE 'SALES FILE REPORT'
```

Produce This Report:

```
 04/05/14                    SALES FILE REPORT                    PAGE    1

                 DEPT              NUM    SALES  SALES    AMOUNT
 REGION EMPL NUM NUM  EMPL NAME  ACCOUNTS  DATE   TIME     PAID      TAX

______ ________ ____ __________ ________ ______ ______ __________ _______
 SOUTH  037      1    JOHNSON       128   950312 102500     101.38    6.09
 WEST   044      4    BAKER         147   950326 120909     137.00    8.22
 EAST   042      3    MORRISSOHN    154   950329 153022      44.35    2.66
 EAST   042      3    MORRISSOHN    154   950330 190541      29.65    1.78
 EAST   041      3    SIMPSON        16   950401 081757      14.99    0.90
 NORTH  039      2    JOHNSON       104   950401 170247     234.45   14.07
 NORTH  039      2    JOHNSON       104   950405 143310       9.98    0.60
 WEST   044      4    BAKER         147   950412 143112     135.75    8.15
 WEST   045      4    THOMAS        118   950414 154138       9.98    0.60
 NORTH  036      2    JONES          78   130415 075832      10.25    0.62
 NORTH  036      2    JONES          78   130415 080159     121.76    7.31
 NORTH  036      2    JONES          78   130415 135241      10.25    0.62
 SOUTH  037      1    JOHNSON       128   950416 114833     500.00   30.00
 EAST   041      3    SIMPSON        16   950430 153021      23.87    1.43
0GRAND TOTAL                                            1,383.66   83.05
```

Figure 7. A Report with Keyed Reads to a Second File

Chapter 4. Using Macros

z/Writer provides a macro facility that makes it easy to customize individual runs by simply changing the value in a MACRO statement at the beginning of your code.

When used, macros definitions should normally precede all other statements. The format of a macro definition is:

```
MACRO $name = value
```

All macro names must begin with a dollar sign ($). That allows z/Writer to distinguish between macro references from fieldnames or keywords.

Here is an example of a macro definition:

```
MACRO $BEGDATE = 01012018
```

As soon as z/Writer processes this statement, *all* occurrences of $BEGDATE in subsequent control statements will be replaced with the text 01012018. This even includes occurrences of the macro name found in quoted texts, in comments, and in subsequent MACRO statements.

Here is an example of using a MACRO statement to specific selection criteria in a run:

```
MACRO $BEGDATE = 01012013
...
IF RECDATE = '$BEGDATE'
       INCLUDE
ENDIF
```

The IF statement will be treated as if it was written this way:

```
IF RECDATE = '01012018'
       INCLUDE
ENDIF
```

Note that the control listing printed by z/Writer shows both the original versions of statements that reference macros, and the modified version. That lets you see exactly how z/Writer sees the final statement.

> **Note:** you only need to enclose the MACRO statement's "value" in ticks *if* the value contains embedded spaces or other delimiters. When the value is enclosed in ticks, the outer ticks are *not* considered a part of the value. However, you may use double ticks anywhere within the outer ticks if you want a tick to be included in your value. For example:

```
MACRO $BEGDATE = '''01012018'''
...
IF RECDATE = $BEGDATE
       INCLUDE
ENDIF
```

Now you do not need ticks in the source version of the IF statement, since in this case they are included in the macro value itself.

Figure 8 shows an example of using a macro in a report.

These Control Statements:

```
MACRO %SELREGION = NORTH

COPY SALES

IF REGION = '%SELREGION'
    PRINT REGION EMPL_NUM EMPL_NAME SALES_DATE SALES_TIME AMOUNT TAX
    INCLUDE
ENDIF
EXCLUDE

TITLE 'SALES FILE REPORT'
```

Produce This Report:

```
04/07/14             SALES FILE REPORT            PAGE    1

                             SALES  SALES    AMOUNT
REGION EMPL NUM EMPL NAME     DATE   TIME     PAID      TAX
______ ________ __________ ______ ______ _________ _______
NORTH  039      JOHNSON    950401 170247     234.45   14.07
NORTH  039      JOHNSON    950405 143310       9.98    0.60
NORTH  036      JONES      130415 075832      10.25    0.62
NORTH  036      JONES      130415 080159     121.76    7.31
NORTH  036      JONES      130415 135241      10.25    0.62

GRAND TOTAL                                  386.69   23.22
```

Figure 8. Using a Macro to Select Records for a Report

Chapter 5. z/Writer Control Statements

This chapter explains the syntax and usage details of each of z/Writer's control statements.

BREAK Statement

PURPOSE

This is a declaratory statement that specifies that a control break should occur in an auto-cycle report. The BREAK statement can also be used to customize the handling of the grand totals.

SYNTAX

BREAK STATEMENT SYNTAX

```
BREAK fieldname/#GRAND
       [ 'total line text'                ]
       [ BREAKCODE(label1 [THRU label2] )  ]
       [ NOTOTALS                         ]
       [ SPACE(n/PAGE [,N/PAGE] )         ]
```

Abbreviations Allowed:

```
BREAKCODE - BRKCODE
NOTOTALS - NOTOTAL
```

DISCUSSION

BREAK statements are only allowed when these two conditions are met:

- it is used for the primary report in a program phase (page 14).
- it is used in a report that uses auto-cycle logic (page 9).

If you want control breaks in reports that do not meet both of these requirements, you will need to code the break logic yourself.

The BREAK statement specifies that a control break should occur whenever the value of the named field changes. By default, the following actions are taken at a control break:

- a **total line** prints, showing the total value of all quantitative fields for that break
- a **single, extra blank line** prints

The **layout of the total line** is governed by the primary (first) PRINT statement in your program code. The total line at a control break shows:

- the control group's total value for each quantitative field listed in the primary PRINT statement. A "quantitative" field is any numeric field for which the "decimals" parm was specified. (That includes numeric fields for which *zero* decimals was explicitly specified, using a DEC(0) parm or a shorthand notation such as N7.0.)
- the value of the breaking control field itself (if it was listed in the primary PRINT statement.)
- the value of any more major control field(s) that also appeared in the primary PRINT statement.

Since the break total line is governed by the first PRINT or PRINTMODEL statement, be sure your program has at least one of those statements whenever you use a BREAK statement. Try to design your print line (in the PRINT statement) so that there will be **enough room at the beginning** of the line to print the total line text (e.g., "TOTALS FOR REGION") before the first numeric column that is totalled. (Otherwise, z/Writer will have to be split the total line into two lines.) You can either begin the print line with some character fields (that are not totalled) or use a numeric spacing factor with the first columns (to shift it to the right):

```
PRINT SALES_QTR1(+20) SALES_QTR2 SALES_QTR3 SALES_QTR4
```

Normally, only a **sort field** should be named in a BREAK statement. (A sort field is a field that appears in the PRESORT parm of the primary input file's FILE statement.) However, in some cases you may know that the primary input file will already be sorted externally. In such cases it would make sense to break on a field that the file is sorted on, even though there is no PRESORT parm on the FILE statement.

You may have **multiple** BREAK statements in a report. The BREAK statements should appear in major-to-minor order. (That is, the same order as in the PRESORT statement.)

The BREAK statement can also be used to customize the **grand totals**. Just specify #GRAND as the break field name on the statement.

The BREAK statement can also:

- specify control break spacing (whether to skip to a new page or print a number of blank lines at a control break)
- suppress the default printing of a total line at a control break
- customize the text used in the total line at a control break

BREAK Statement Location

As a declaratory (rather than executable) statement, the *exact* location of your BREAK statement is not critical. However, it should appear near the beginning of the z/Writer control statements (after your file definitions). It is *mandatory* that it appear before the first PRINT statement.

PARMS

The fieldname is required in a BREAK statement, and must be the *first* item after the statement name. 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. This field will normally have been specified as a sort field in the primary input file's FILE statement.

```
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 one blank line.

You may also specify #GRAND rather than an actual field name. Using #GRAND allows you to specify control break options for the grand totals "pseudo control break" (at the end of the whole report).

```
BREAK: #GRAND  'END OF RUN TOTALS'
```

The above statement specifies that the text on the grand total line should be "END OF RUN TOTALS."

'total line text'

Specifies the constant text to use at the beginning of each break total line. (When this parm is not present, the total line will begin with a default text.)

```
BREAK REGION 'TOTALS FOR THE ABOVE REGION'
```

The above example specifies that a control break should occur whenever the value of REGION changes. The total line at this break should begin with the text "TOTALS FOR THE ABOVE REGION."

BREAKCODE(label1 [THRU label2])

Specifies a paragraph, or a range of paragraphs, to be performed each time the control break occurs. The paragraph(s) will be performed *before* the default total line prints (if it prints).

```
BREAK: REGION  BREAKCODE(REG-100 THRU REG-999)
```

The above example specifies that whenever the REGION field changes value a control break should occur. At the break, paragraphs REG-100 through REG-999 (inclusive) will be performed. After that, the default total line and one blank line will print.

The built-in field named #TALLY is also useful within break code. It is a numeric field containing the number of records included in the control group just ended.

Note: you can use this parm (in a BREAK statement for #GRAND) to perform custom code at EOF on the primary input file. (The regular program code does not execute when the file gets to EOF.)

NOTOTALS

Suppresses the total line which normally prints at a control break.

```
BREAK: REGION  NOTOTAL
```

The above example specifies that a control break should occur whenever the REGION field changes value. However, no total line will print. z/Writer will just print the default blank line after the break.

SPACE(1 / n / PAGE [, n / PAGE])

Specifies the spacing desired at the control break. An "n" in this parm represents a number of blank lines to print. "PAGE" indicates that a page break is wanted.

When only one value ("n" or PAGE) is specified, that value determines the spacing to be performed *after* the control break processing is complete. That is, after the break code, if any, has been performed and after the total line, if any, has printed.

When two "n" and/or PAGE values are specified, the first one indicates the spacing desired *before* the control break. That is, before the break code, if any, is performed and the total line, if any, is printed. The second value indicates the spacing desired *after* the control break processing is complete.

When this parm is omitted, the default is to print 1 blank line after the control break processing.

```
BREAK REGION  SPACE(1, PAGE)
BREAK CITY    SPACE(2) NOTOTALS
```

The above example specifies that one blank line should print *before* the total line prints at REGION control breaks. And after the REGION total prints, the report should skip to a new page. When the CITY field changes value, two blank lines (and no total line) should print.

EXAMPLE

```
FILE EMPL TYPE(KSDS) V(150)
     PRESORT(STATE DEPT_NUM EMPL_NUM LAST_NAME)
FLD EMPL_NUM         N3
FLD LAST_NAME        15
FLD FIRST_NAME       15
FLD HIRE_DATE        N6
FLD DEPT_NUM          1
FLD SEX               1
FLD STATUS_BYTE       1
FLD SOCIAL_SEC_NUM   N9
FLD NUM_ACCOUNTS     N4
FLD TOTAL_SALES      N7.2
FLD SALES_QTR1       N7.2
FLD SALES_QTR2       N7.2
FLD SALES_QTR3       N7.2
FLD SALES_QTR4       N7.2
FLD ADDRESS          20
FLD CITY             15
FLD STATE             2
FLD ZIP               5
FLD TELEPHONE        10

****************** PROCEDURE *****************
BREAK  #GRAND   NOTOTALS
       BRKCODE(GRAND-TOTALS)

BREAK  STATE    NOTOTALS
       BRKCODE(STATE-TOTALS)

BREAK  DEPT_NUM NOTOTALS
       BRKCODE(DEPT-TOTALS)

PRINT STATE(+25) DEPT_NUM  EMPL_NUM  LAST_NAME
      SALES_QTR1  SALES_QTR2  SALES_QTR3  SALES_QTR4
```

```
TITLE 'SALES BROKEN DOWN BY DEPARTMENT AND STATE'
GOTO DONE

DEPT-TOTALS:
 PRINT 'DEPT' DEPT_NUM(LAST) 'TOTALS'
       SALES_QTR1(SUM @SALES_QTR1) SALES_QTR2(SUM)
       SALES_QTR3(SUM)             SALES_QTR4(SUM)

STATE-TOTALS:
 PRINT STATE(LAST) 'STATE TOTALS (' #TALLY(+0) 'RECORDS)'
       SALES_QTR1(SUM @SALES_QTR1) SALES_QTR2(SUM)
       SALES_QTR3(SUM)             SALES_QTR4(SUM)

GRAND-TOTALS:
 PRINT 'GRAND TOTALS (' #TALLY(+0) 'RECORDS)'
       SALES_QTR1(SUM @SALES_QTR1) SALES_QTR2(SUM)
       SALES_QTR3(SUM)             SALES_QTR4(SUM)


DONE:
```

CALL Statement

PURPOSE

Calls an external program. Optionally, you may pass one or more parms to the called program.

SYNTAX

CALL STATEMENT SYNTAX

```
CALL program [ USING fieldname/'literal' [, fieldname/'literal'] ...) ]
```

DISCUSSION

z/Writer uses the standard Assembler language **linkage conventions** when calling the program. Specifically, the following registers will have special settings:

- R1 holds the address of a standard parm list (if parms were specified on the CALL statement.)
- R13 holds the address of an 18 fullword save area that the called program should use to preserve z/Writer's registers.
- R14 holds the return address within z/Writer.
- R15 holds the entry point address of the called program.

Note: programs are called in 24-bit address mode, and must return in the same.

PARMS

program

Specifies the name of the program to call. This parm must be the 1-8 byte name (or alias) of a load module present in a program library accessible to the job step.

Note: you may need to add the library containing the called program to your JOBLIB or STEPLIB DD in order for the program to be found.

fieldname / 'literal'

Specifies one parm to be passed to the called program. If you name a field, its contents will be passed to the program as parm. If you specify a literal text (within quotes or apostrophes), that text will be passed to the program as a parm.

Note: when a field is used as a parm to the program, z/Writer passes the address of the field's data in the actual record area or workarea. The called program is allowed to modify the data there, but it should be careful not to accidentally modify memory before or after that area.

EXAMPLE

```
CALL DECRYPT USING ENCRYPT-SEGMENT 'TYPE1'
```

CASE Statement

PURPOSE

The CASE statement begins a "case-structure." The purpose of a case-structure is to conditionally execute (at most) one set of statements within the structure.

SYNTAX

CASE STRUCTURE SYNTAX

```
CASE fieldname

WHEN  [NOT] value/range [value/range ...]
other statements

WHEN  [NOT] value/range [value/range ...]
other statements
...

ELSE
other statements

ENDCASE
```

DISCUSSION

A case-structure functions just like an IF statement followed by a number of ELSEIF statements and an ELSE statement. If you are simply comparing one field to various values, you can use a case-structure instead of an if-structure. The advantage of the case-structure is that it is less verbose, and may be easier to read, modify and maintain.

Here is a description of the case-structure syntax.

The CASE statement itself must contain exactly one fieldname, of any type. This is the "test field." Immediately after the CASE statement, a WHEN statement must follow.

A WHEN statement contains one or more values (or range of values) which the test field (from the CASE statement) should be compared to. Therefore the values in the WHEN statements must be of a type (character or numeric) that is compatible with the test field.

If the test field equals any value on the WHEN statement, then that WHEN statement is true and all of the "other statements" immediately following it will be executed. After those statements are executed, control is passed to the statement immediately following the ENDCASE statement. (You are allowed to have zero "other statements" following a WHEN, in which case control just passes immediately to the first statement after the ENDCASE statement.)

If the CASE test field is not equal to any value on the WHEN statement, then the next WHEN statement (if any) is evaluated. This process continues until a "true" WHEN statement is found, or until there are no more WHEN statements.

If none of the WHEN statements are true, then one of the following occurs:

- if there is a final ELSE statement within the case-structure, the statements following it are executed.
- if there is no ELSE statement, control just passes to the statement after the ENDCASE statement (with none of the "other statements" being executed.)

The last element of a case-structure is always the required ENDCASE statement.

WHEN Statement Syntax

A WHEN statement can contain:

- a single value (either a fieldname or a literal)
- a range of values (specified using the keyword THRU), or
- a list of values and/or ranges

Note: when specifying a list of values and/or ranges, you may separate them with spaces or commas

Using NOT on WHEN Statements

You may also begin a WHEN statement with the keyword "NOT" (followed with one or more values and/or ranges):

```
WHEN NOT 2, 4, 6, 100 THRU 9999999999
```

When NOT is specified, the WHEN statement is passed if the test field *does not* equal any value on the statement, and does not fall within any range (inclusive) on the statement.

EXAMPLE

```
CASE AMOUNT
  WHEN TEST-AMOUNT
    PRINT 'EQUALS TEST AMOUNT'
  WHEN 400
    PRINT 'EQUALS 400'
  WHEN 500 THRU 599
    PRINT 'BETWEEN 500 AND 599 (INCLUSIVE)'
  WHEN 2, 4, 6
    PRINT 'EQUALS 2, 4 OR 6'
  WHEN 1, 3, 5, 700-799, 1000-1099, 9999
    PRINT '1, 3, 5, SEVEN HUNDRED SOMETHING, ONE THOUSAND SOMETHING OR 9999'
  ELSE
    PRINT 'DID NOT EQUAL ANY OF THE ABOVE'
ENDCASE
```

CLOSE Statement

PURPOSE

Closes a file or DB2 cursor (if it is open.)

SYNTAX

CLOSE STATEMENT SYNTAX

```
CLOSE   filename/cursorname
```

DISCUSSION

You do not normally need to code an explicit CLOSE statement. z/Writer closes all open files for you at the end of your program. (Or at the end of each phase's execution, when the program consists of multiple phases.)

However, this statement (along with the OPEN statement) is useful if you wish to read through a file or DB2 cursor more than one time within the same phase.

If you attempt to close a file that is not open, the CLOSE instruction is just ignored. No error occurs and no messages are printed.

PARMS

filename/cursorname

Specifies the name of the file or DB2 cursor to close. The file or cursor must have been previously defined using a FILE or CURSOR statement. This parm is required.

EXAMPLE

```
OPEN EMPL
READ EMPL
DOWHILE #EOF <> 'Y'
   PRINT EMPL_NUM EMPL_NAME
   READ EMPL
ENDDO
CLOSE EMPL

OPEN EMPL /* READ 2ND PASS */
...
```

COMPUTE Statement

PURPOSE

Use this statement to compute a value and move it to a field.

SYNTAX

COMPUTE STATEMENT SYNTAX

```
[ COMPUTE ] fieldname = computational-expression
```

Note: the statement name (“COMPUTE”) is optional. You may omit it if you prefer. Both of the following statements are valid and do the same thing:

```
COMPUTE TOTAL = AMOUNT + TAX
TOTAL = AMOUNT + TAX
```

DISCUSSION

Use the COMPUTE statement to perform calculations on one or more fields and literals to obtain a new value, to move to a target field.

In addition to math and character operations, z/Writer has many powerful built-in functions to help calculate your value. These easy-to-use functions can save you a lot of procedural code when performing complex tasks, such as:

- computing math functions (such as square roots, modulus, rounding, truncating and more)
- parsing words from text strings
- searching for or replacing text strings

Appendix B, "Built-In Functions" on page 184 contains a complete list of built-in functions that can be used in COMPUTE statements.

Note: to simply assign the contents of one field (or literal) to another field, you can use the more efficient MOVE statement. The MOVE statement is allowed even when the field’s are stored in different formats.

Syntax of Computational Expressions

Computational expressions are used to specify a value. A computational expression might be nothing more than a single field name (or literal value). Or, it might be dozens of lines long and involve many mathematical operations and built-in functions.

COMPUTATIONAL EXPRESSION SYNTAX

```
operand  [ operator operand ]  [ operator operand ]  ...
```

Parentheses can also be used, indicate the priority of operations. Nesting is allowed to any level.

Only the first operand is required. You may specify as many additional operator/operand pairs as you like. Computational expressions return either a character or numeric value. The type of expression (numeric or character) must be compatible with the target field to which it is being assigned.

Operands

An operand in a computational expression specifies a data value. An operand can be any of the following:

- a literal value
- a field from a file record area or a work area
- a built–in field (such as #TIME)
- a built–in function (see Appendix B, "Built-In Functions" on page 184 for a complete list)

Operators

An operator in a computational expression specifies an operation to perform on the operands surrounding it. The following table shows the operators that are supported, and the symbol to use for each

OPERATORS ALLOWED IN COMPUTATIONAL EXPRESSIONS	
CHARACTER OPERATORS	NUMERIC OPERATORS
+ (concatenation)	+ (addition)
	– (subtraction)
	* (multiplication)
	/ (division)
	** (raise to an integer power)

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 all other operators, blanks are not required around the symbol (but are allowed.)

Order of Operations

Operations within parentheses are performed first. If nested parentheses are encountered, the most deeply nested operations are performed first. Within the same level of nesting, the order of operations is as follows:

- powers are performed first
- multiplications and divisions are performed next
- additions and subtractions are performed last

Operations of equal priority are performed left to right.

Decimal Precision in Results

The final precision (number of decimal digits) of the result is, of course, determined by the definition of the target field (the field where the computed result is placed.) However, for complex expressions, z/Writer must often compute intermediate results while computing the final value. After divisions in particular, it is not always possible to retain all of the decimal digits of the result. z/Writer determines the number of decimal digits to retain in such intermediate results. It take into consideration the number of decimal digits defined for the operators in the expression, and for the target. However, to prevent overflow errors, z/Writer will not attempt to retain a very large number of decimal digits.

Note: if you are concerned about the precision of an intermediate calculation in a complex computation, we recommend breaking the whole thing down into multiple, simple computations. You can then control the precision of each intermediate result (via your definition of those target fields.)

Note: when multiplication and division are performed at the same level within an expression, we recommend putting the multiplications before the divisions. This usually results in the best precision for the calculation. For example:

```
PERCENT = AMOUNT*100/TOTAL  (recommended)
PERCENT = AMOUNT/TOTAL*100  (not recommended)
```

EXAMPLES

```
COMPUTE PERCENT-TAX = (TAX * 100) / (AMOUNT + TAX)
COMPUTE A=A+1
COMPUTE DOLLARS = #INT(AMOUNT) /* RETURN INTEGER PORTION */
COMPUTE ROUNDED-AMOUNT = #ROUND(AMOUNT,0) /* ROUND TO 0 DECIMALS */
COMPUTE FIRST-INITIAL = #SUBSTR(FIRST-NAME,1,1)
```

```
COMPUTE WHOLE-NAME = LAST-NAME + ', ' + FIRST-NAME
COMPUTE FORMATTED-NAME = #COMPRESS(FIRST-NAME, LAST-NAME)
COMPUTE NEW-DESC = #REPLACE(DESC,'THIS','THAT')
```

COPY Statement

PURPOSE

Causes control statements from a member of a PDS to be included in the program. Use this statement, for example, to copy file definitions or common processing routines from a copy library.

SYNTAX

COPY STATEMENT SYNTAX

```
COPY member
     [COBOL/ASM]
```

DISCUSSION

In addition to copying other z/Writer statements, if you are in the field definition portion of a program, you may also copy COBOL or Assembler field definitions. z/Writer internally converts them to z/Writer field definitions.

PARMS

member

Specifies library member to copy. This parm must be the 1-8 byte name of a member present in the PDS identified by the SYSCOPY DD statement in the execution JCL.

COBOL / ASM

Specifies that the member to be copied contains field definitions written in COBOL (or Assembler). These parms are only allowed on COPY statements that follow a FILE statement (and optionally one or more FIELD statements.)

Note: when copying COBOL members, if the first field definition is at the 01 level, then the COBOL copybook defines fields starting at the beginning of the whole file record area (even if earlier FIELD statements were also processed). If the COBOL member begins at a higher level (05, 10, etc.) then the field is defined at the current location within the file record area. That may be the beginning of the record area or, if there were preceding FIELD statements, after the last field defined.

Note: Initial values contained in the COBOL or ASM definitions are treated as comments.

EXAMPLES

```
COPY EMPLDEFS
COPY ACCT120C COBOL
COPY SALEREC ASM
```

CURSOR Statement

PURPOSE

Requires z/Writer's DB2 Option. Defines one DB2 input source to z/Writer. Before DB2 data can be accessed in a run, it must first be defined using the CURSOR statement. The CURSOR statement defines a DB2 "cursor," which is used to create and fetch rows from a DB2 result table. (The CURSOR statement is the DB2 equivalent of a FILE statement.)

SYNTAX

CURSOR STATEMENT SYNTAX

```
CURSOR cursorname
       QUERY( SQL fullselect )
       [ FLOATDEC(n/3 ]
       [ SHOWFLDS     ]
```

DISCUSSION

This statement by itself does not retrieve any DB2 data. This statement simply defines a DB2 input to z/Writer so that subsequent control statements can refer to that input. After a DB2 input has been defined using this statement, OPEN, FETCH and CLOSE statements may be used to perform I/O operations on the DB2 input.

You may have more than one CURSOR statement in a run. Their order is not important — except in one case. If you are producing a DB2 auto-cycle report (see page 9), be sure that the first CURSOR you define is the cursor that z/Writer should read automatically.

The CURSOR statement does several things:

1. The CURSOR statement defines a cursor name, which serves a similar purpose as a file name in the program code. The cursor name is the name that you will use in your OPEN, FETCH and CLOSE statements for this DB2 input.

2. The CURSOR statement always specifies a DB2 query, in the form of a SQL SELECT statement. The query defines a "result table" from DB2. The rows of this result table can be "fetched" later in the program, using the FETCH statement.

3. Like files, cursor also have fields associated with them. These fields can be referenced in the program code, just like fields defined for files, tables and workareas. However, you do not code FIELD statements for these fields yourself. z/Writer automatically defines one new field for each result column that appears in the result table. That is, one field for each column selected in the "select" clause of the SQL Select statement (in the QUERY parm). Learn more about using the CURSOR statement in the section titled "The CURSOR Statement - A Quick Look" on page 159.

In additional to the fields defined for the result columns, z/Writer also maintains certain built-in fields for each cursor. Some of these fields are "read-only." Read-only means that you can examine the contents of the field, but not modify it. The built-in fields available for a cursor are shown in the table below. Note that if your program has more than one cursor or file, you will need to qualify these built-in field names with the cursor name in order to uniquely reference them. (For example, PROJECT.#EOF).

Built-In Fields Available for DB2 Cursors

FIELDNAME	TYPE & LENGTH	READ ONLY	DESCRIPTION
#COUNT	4-byte binary	Yes	Number of successful fetches from the cursor.
#EOF	1-byte character	Yes	Indicates that the most recent operation attempted to fetch past the last row in the cursor ("end of file"). Y = the last FETCH raised the end of file condition N = the end of file condition has not been raised.
#RECSIZE	4-byte binary	No	This field contains a constant value, which is the sum of all the result columns (plus room for any null indicators).
#SQLCODE	2-byte binary	Yes	Indicates the SQLCODE from the more recent SQL operation performed (OPEN, FETCH, etc.) Check your SQL documentation for the meaning of the SQL codes.

PARMS

cursorname

An arbitrary name for the cursor being defined. All other control statements will use this name when referring to this DB2 input. The cursorname is required.

Example:

```
CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)
```

You may assign any name you like, within the naming conventions in the box below.

Requirements for Cursor Names

- cursor names may be from 1 to 70 characters long
- cursor names may contain alphanumeric characters, the "national" characters #, @, and $, underscores (_) and dashes (-).
- cursor names must not begin with a numeric character
- cursor names are not case sensitive
- cursor names must not be the name of a file, workarea or table
- cursor names must not be a statement name

Examples

```
SALES
EMPLNUMS
EMP#
ACCOUNTING-DEPT-PROJECT-TABLE
```

QUERY(SQL fullselect)

This parm is required. Within the parentheses, you should specify a valid SQL "fullselect". It will always begin with the word SELECT. It may contain FROM, WHERE, ORDER BY, GROUP BY, HAVING clauses. It may *not* include an INTO clause. (Fetches are always performed "into" the fields that z/Writer automatically defines for the cursor.)

FLOATDEC(n/3)

This optional parm specifies how many decimal digits to keep when converting floating point DB2 data to fixed decimal data.

Of course, by their nature floating point fields can hold varying number of decimal digits. All of those digits are preserved when the floating point data is received by z/Writer from DB2. They are also preserved when a floating point field is moved to another floating point field.

However for other processing (such as moving a floating point field to a decimal field, or when printing the value of a floating point field in a report), z/Writer first converts the floating point value to a fixed decimal format (in an internal, intermediate data area). The FLOATDEC parm tells z/Writer how many decimal digits to preserve in this intermediate field.

By default, z/Writer keeps 3 decimal digits for such floating point conversions. Use this parm to change this default for all floating point fields fetched from this DB2 input. (Or use the global FLOATDEC option in an OPTION statement to change this default for floating point fields from all input sources.)

Example:

```
CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ) FLOATDEC(5)
```

SHOWFLDS

This optional parm causes z/Writer to list all of the fields that it creates for your cursor. The list of fields (along with their z/Writer data type and length) appears in the control listing.

Example:

```
CURSOR PROJECT QUERY(SELECT * FROM DSN8111O.PROJ) SHOWFLDS
```

DATA Statement

PURPOSE

Denotes the end of the program statements (in the SYSIN stream) and the beginning of the data records for the CARD input file.

SYNTAX

DATA STATEMENT SYNTAX

```
DATA
```

DISCUSSION

If you have defined an input file as a CARD type file, use the DATA statement to tell z/Writer where the program ends and the data part of the SYSIN stream begins. Program parsing ends when the DATA statement is encountered. All records following the DATA statement are data records. They are read during program execution by using a READ statement for the CARD type input file.

PARMS

The DATA statement has no parms.

EXAMPLES

```
FILE STATUS-CODES TYPE(CARD)
CODE 2
DESC 10

READ STATUS-CODES
DOWHILE #STATUS = 'Y'
    PRINT 'CODE' CODE 'MEANS' DESC
    READ STATUS-CODES
ENDDO
DATA
FTFULLTIME
PTPARTTIME
```

DELREC Statement

PURPOSE

Deletes the current record from a VSAM file that has been read for "update" processing. (A record must have been successfully read from the file before using this statement.)

SYNTAX

DELREC STATEMENT SYNTAX

```
DELREC  filename
```

DISCUSSION

When you specify UPDATE on a FILE statement, the records read from that file are available for update processing. Each time a record is read from an update file, VSAM puts a lock on the record until you take one of the following actions:

- you update that record by executing a REWRITE statement for the same file (normally after changing the contents and/or length of the record)
- you delete that record by executing a DELREC statement for the same file
- you explicitly release the record by executing a RELEASE statement for the same file
- you READ a new record from the same file, which releases the lock on the current record (and puts a lock on the newly read record)

After the DELREC statement, you can check the result of the operation by examining the file's #STATUS built-in field. (If your phase has more than one file defined, you will need to qualify #STATUS by preceding it with a filename and a period.)

The table below shows the possible values of #STATUS after a DELREC.

#STATUS Built-In Field Values After a DELREC Statement

Y - record successfully deleted
N - record not deleted.

PARMS

filename

Specifies the name of the file whose current record should be deleted. The file must have been previously defined (in a FILE statement) as an UPDATE file. Also, a record must have been successfully read from that file, and not yet rewritten or released.

EXAMPLE

```
FILE EMPL UPDATE TYPE(KSDS)
...

READ EMPL         /* READ FIRST RECORD FOR UPDATE */

DOWHILE EMPL.#STATUS = 'Y'
  IF EMPL.EMPL-NUM = '444'
    DELREC EMPL    /* DELETE RECORD FROM FILE */
  ELSEIF EMPL.EMPL-NUM = '555'
    MOVE 'I' TO EMPL.STATUS   /* MARK REC 555 AS INACTIVE */
    MOVE 40 TO EMPL.#LENGTH   /* SHORTEN INACTIVE RECORD */
    REWRITE EMPL              /* REWRITE SHORTER 555 RECORD */
  ELSE
    RELEASE EMPL    /* LEAVE RECORD UNCHANGED ON FILE. */
  ENDIF
  ... /* OTHER PROCESSING */
  READ EMPL /* READ NEXT RECORD FOR UPDATE */
ENDDO
```

DELTABREC Statement

PURPOSE

Deletes the last record retrieved from a keyed table.

SYNTAX

DELTABREC STATEMENT SYNTAX

```
DELTAB  tablename

Abbreviations Allowed:
DELTABREC - DELTAB
```

DISCUSSION

This statement is only allowed with *keyed* tables. The last record retrieved from the table will be removed from the table. The record to be deleted may have been retrieved either sequentially or directly (with a key.) If no records have yet been retrieved from the table, this statement does not delete any record from the table.

This statement does not change the current contents of the record area of the table. That is, even though the record has been removed from the table itself, the copy that was moved to the record area for the earlier retrieve remains there. The #ENTRY built-in function for the table also remains unchanged.

However, z/Writer's internal "position" within the table is backed up to the record just before the record that was deleted. (Or to the beginning of the table, when you delete the first record in the table.) This allows you to resume performing sequential retrievals, if desired, from the same place in the table. That is, if you perform a sequential RETRIEVE after a DELTAB statement, you will get the first record past the deleted record.

Note: after the delete, the table remains a balanced binary tree, for efficient access.

After the DELTAB statement, you can check the result of the operation by examining the table's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the tablename, if the phase has definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a DELTAB.

#STATUS Built-In Field Values After a DELTAB Statement

Y - record successfully deleted
N - record not deleted.

PARMS

tablename

Specifies the name of the table whose last record retrieved should be deleted. The table must have been previously defined (in a TABLE statement) as a keyed table.

EXAMPLE

```
TABLE EMPL-TABLE
EMPNUM 5
INACTIVE 1
...

RETRIEVE EMPL-TABLE       /* RETRIEVE FIRST RECORD FROM TABLE */

DOWHILE EMPL-TABLE.#STATUS = 'Y'
   IF EMPL-TABLE.INACTIVE = 'Y'
      DELTAB EMPL-TABLE    /* DELETE INACTIVE RECORD FROM TABLE */
   RETRIEVE EMPL-TABLE /* RETRIEVE NEXT RECORD TO EXAMINE */
ENDDO
```

DOUNTIL Statement

PURPOSE

The DOUNTIL statement begins an "iterative-structure." The purpose of an iterative-structure is to execute a set of "other statements" some variable (non-zero) number of times, depending on a logical condition.

Unlike the similar DOWHILE statement, the "other statements" after a DOUNTIL statement are always executed at least one time.

SYNTAX

DOUNTIL STRUCTURE SYNTAX

```
DOUNTIL conditional-expression
   other statements
ENDDO
```

DISCUSSION

The DOUNTIL structure begins with a DOUNTIL statement and ends with an ENDDO statement. Within the structure any number (including zero) of other statements are allowed. Those statements may include additional DOUNTIL structures. Such nesting is permitted to any level.

This is how the DOUNTIL structure is processed:

- When control reaches the DOUNTIL statement, the DOUNTIL statement itself does nothing and the "other statements" after it are then executed.
- When the corresponding ENDDO statement is encountered, the conditional expression from the DOUNTIL statement is evaluated.
- If the condition is true, the structure's iterations are complete and control passes to the first statement after the ENDDO statement
- If the expression is false, another iteration is performed; control passes back up to the DOUNTIL statement and the statements following it are again executed.

The above process continues until the conditional expression from the DOUNTIL statement is found to be true. (This of course means that the program will loop "forever" if the condition is never true.)

EXAMPLE

```
READ SALES
DOUNTIL SALES.#STATUS <> 'Y'
  TOTAL-SALES = TOTAL-SALES + AMOUNT
  READ SALES
ENDDO

J = 1
DOUNTIL J > 9
    NAME[J] = ID[J
    J=J+1
ENDDO
```

DOWHILE Statement

PURPOSE

The DOWHILE statement begins an "iterative-structure." The purpose of an iterative-structure is to execute a set of "other statements" some variable number of times (possibly zero), depending on a logical condition.

Unlike the similar DOUNTIL statement, it is possible for the "other statements" after a DOWHILE statement to not be executed at all.

SYNTAX

DOWHILE STRUCTURE SYNTAX

```
DOWHILE conditional-expression
   other statements
ENDDO
```

DISCUSSION

The DOWHILE structure begins with a DOWHILE statement and ends with an ENDDO statement. Within the structure any number (including zero) of other statements are allowed. Those statements may include additional DOWHILE structures. Such nesting is permitted to any level.

This is how the DOWHILE structure is processed:

- Whenever control reaches the DOWHILE statement, its conditional expression is evaluated.
- If the expression is false, the structure's iterations (if any) are complete; control passes to the first statement after the corresponding ENDDO statement.
- If the DOWHILE statement's condition is true, the "other statements" after it are executed.
- When the ENDDO statement is reached, control is passed back up to the DOWHILE statement (where the conditional expression is reevaluated.)

The above process continues until the conditional expression from the DOWHILE statement is found to be false. (This of course means that the program will loop "forever" if the condition is never false.)

EXAMPLE

```
READ SALES
DOWHILE SALES.#STATUS = 'Y'
  TOTAL-SALES = TOTAL-SALES + AMOUNT
  READ SALES
ENDDO

J = 1
DOWHILE J <= 10
    NAME[J] = ID[J
    J=J+1
ENDDO
```

ELSE Statement

PURPOSE

The ELSE statement is used within "if-structures" and "case-structures." The purpose of both structures is to conditionally execute (at most) one set of statements within the structure. The ELSE statement introduces the optional, final default set of instructions to be executed when none of the structure's earlier clauses are executed.

SYNTAX

IF STRUCTURE SYNTAX

```
IF conditional-expression
other statements

[ ELSEIF conditional-expression
other statements  ]

[ ELSEIF conditional-expression
other statements  ]
...
[ ELSE
other statements ]

ENDIF
```

CASE STRUCTURE SYNTAX

```
CASE fieldname

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]
...

[ ELSE
other statements  ]

ENDCASE
```

DISCUSSION

A complete discussion of an “if-structure” can be found under the IF statement on page 86. A complete discussion of a “case-structure” can be found under the CASE statement on page 32.

ELSEIF Statement

PURPOSE

The ELSEIF statement is used within an “if-structure”. The purpose of an if-structure is to conditionally execute (at most) one set of statements within the structure. The ELSEIF statement introduces a new set of instructions and the condition under which they should be executed, assuming that none of the if-structure’s earlier clauses are true.

SYNTAX

IF STRUCTURE SYNTAX

```
IF conditional-expression
other statements

[ ELSEIF conditional-expression
other statements  ]

[ ELSEIF conditional-expression
other statements  ]
...
[ ELSE
other statements ]

ENDIF
```

DISCUSSION

A complete discussion can be found under the IF statement on page 86.

ENDCASE Statement

PURPOSE

The ENDCASE statement ends a “case-structure”. The purpose of a case-structure is to conditionally execute (at most) one set of statements within the structure.

SYNTAX

CASE STRUCTURE SYNTAX

```
CASE fieldname

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]
...

[ ELSE
other statements  ]

ENDCASE
```

DISCUSSION

A complete discussion can be found under the CASE statement on page 32.

ENDDO Statement

PURPOSE

The ENDDO statement ends an “iterative-structure” begun by an earlier DOUNTIL or DOWHILE statement. The purpose of an iterative-structure is to conditionally execute a set of statements a variable number of times, depending on a logical condition.

SYNTAX

DOUNTIL Structure Syntax

```
DOUNTIL conditional-expression
   other statements
ENDDO
```

DOWHILE Structure Syntax

```
DOWHILE conditional-expression
   other statements
ENDDO
```

DISCUSSION

A complete discussion can be found under the DOUNTIL statement (page 50) and the DOWHILE statement (page 52).

ENDIF Statement

PURPOSE

The ENDIF statement ends an “if-structure”. The purpose of an if-structure is to conditionally execute (at most) one set of statements within the structure.

SYNTAX

IF STRUCTURE SYNTAX

```
IF conditional-expression
other statements

[ ELSEIF conditional-expression
other statements  ]

[ ELSEIF conditional-expression
other statements  ]
...
[ ELSE
other statements ]

ENDIF
```

DISCUSSION

A complete discussion can be found under the IF statement on page 86.

ENDREDEFINE Statement

PURPOSE

Allowed only within record descriptions, this statement cancels the redefine in progress.

SYNTAX

ENDREDEFINE STATEMENT SYNTAX

```
ENDREDEFINE [fieldname]

Abbreviations Allowed:
ENDREDEFINE - ENDREDEF
```

DISCUSSION

The ENDREDEFINE statement terminates one or more redefines that are in progress and resets the current location in the record area. It is not required to end each REDEFINE with an ENDREDEFINE statement. But, if you do use this statement, then you must redefine *all* of the bytes in the field being redefined. The purpose of the ENDREDEFINE statement is to let you "cancel" a redefinition early, without having to redefine all of the bytes in the field being redefined.

After the ENDREDEFINE statement, the current location in the record area is set to the location immediately after the field named in the ENDREDEFINE statement. (Or to the location immediately after the last field named in a REDEFINE statement, if no name is used on the ENDREDEFINE statement.) Note that if the redefined field was an array, the current location is set to the end of the whole array, not just the end of the first element.

PARMS

[fieldname]

Specifies the name of the REDEFINE field being ended. Any nested REDEFINEs will be ended. When this parm is omitted, the most REDEFINE statement is ended. You can use this parm to end more than one REDEFINE statement at the same time.

EXAMPLE

```
FLD SALES-DATE       6
REDEFINE SALES-DATE
FLD SALES-DATE-YY    2
FLD SALES-DATE-MM    2
```

```
FLD SALES-DATE-DD    2 /* COMPLETE REDEFINE. ENDREDEF IS OPTIONAL */

FLD CUSTOMER         15

FLD TELEPHONE        N10
REDEFINE TELEPHONE
FLD AREA-CODE        N3
ENDREDEF                   /* PARTIAL REDEFINE. ENDREDEF IS NEEDED */
```

EXCLUDE Statement

PURPOSE

This statement (allowed only in auto cycle runs) “excludes” the current input record and terminates execution of the program for that record. “Excludes” means that the fields in the record *will not be* included in calculating the totals (printed at control breaks and at the end of the run.)

SYNTAX

EXCLUDE STATEMENT SYNTAX

```
EXCLUDE
```

DISCUSSION

This statement is allowed only in auto cycle runs (see page 9). When it is executed, the normal flow of the user program ends (for the current record). That is, any statements following the EXCLUDE statement are not executed for the current record. The auto cycle code reads the next input record and starts program execution again for it *without* including the fields from the current record in any totals that are being accumulated.

This statement is similar to the INCLUDE statement (see page 91). However, the INCLUDE statement *does* include the current record in the totals before it terminates execution of the program for that record.

There are no parms for the EXCLUDE statement.

EXPORT Statement

PURPOSE

This declarative statement can be used to specify that the primary "report" in this program phase should be formatted as an export file (that is, a comma delimited file). Or, this statement can specify the name of a *new export file* to be written in the current program phase (along with any override options to be used for it).

SYNTAX

EXPORT STATEMENT SYNTAX

```
EXPORT
       [ (reportname)                          ]
       [ COLSEP('xxx'/',')                     ]
       [ CURRCHAR('x'/'$')                     ]
       [ FORMAT(display-format ...)            ]
       [ NOCOLHDGS                             ]
       [ NOGRANDTOTALS                         ]
       [ QUOTECHAR('x'/'"')                    ]

Abbreviations Allowed:
NOCOLHDGS - NOCOLHDG
NOGRANDTOTALS - NOGRANDTOTAL, NOGRAND
```

DISCUSSION

Only a single EXPORT statement is allowed per export file, but it may contain as many options as you like.

The report name parm, if used, must be the first parm on the EXPORT statement. The report name must be enclosed in parentheses. It specifies the name of a new export file to be written in the current phase. (Program Phases are discussed on page 97.) After the report name, the other options may be specified in any order. All are optional.

Primary Report

If you want your primary report to be a comma delimited file, you must specify an EXPORT statement without a report name.

When the EXPORT statement does not begin with a report name (in parentheses), the statement applies to the *primary report* for the phase. The primary report is the one that all PRINT statements without a report name write to. (Output for the unnamed, primary report (or export file) goes to the ZWOUT001 DD for the first phase, ZWOUT002 in the second phase, and so on.)

Additional Outputs

When the EXPORT statement does begin with a report name (in parentheses), the statement defines a *new export file* for the current phase. You may have as many reports and/or export files in a single phase as you like.

To write to the new export file, use the same report name parm (again, in parentheses) as the first item in a PRINT statement. (TITLE statements are treated as comments for export files.)

The DDNAME used for the new export file will be the report name itself.

Note that automatic control break processing (using the BREAK statement) is only available for the primary report/export file in each phase.

EXPORT Statement Location

As a declaratory (rather than executable) statement, the *exact* location of your EXPORT statement is not critical. However, it is *mandatory* that it appear before the first PRINT statement for the export file.

Export File Record Length

You can choose the record length to use for your export file in either of two ways:

- specify DCB LRECL information in the output DD in your JCL
- write to an existing fixed length dataset

In either case, z/Writer will format your export file to that maximum length. If no DCB LRECL information is specified for a new output file, z/Writer picks a default length of 133.

Differences Between Reports and Export Files

Please note the following differences between regular reports and reports that are formatted as export files:

- TITLE statements are treated as comments for export files
- Column headings are written only one time, at the beginning of the export file. (Use the NOCOLHDGS parm in the EXPORT statement if you wish to suppress them.)
- PRINT statements for export files may not use the ADVBEFORE and ADVAFTER parms
- no carriage control character is written for export files.

PARMS

(reportname)

This parm, when present, must be the first item on the EXPORT statement.

When a report name is not present, the EXPORT statement specifies that the first (and often only) report in the current phase should be formatted as an export file. All PRINT statements that do not have a report name parm write to this unnamed export file.

Use the report name parm if you are writing to more than one output during a single phase. The reports and export files after the first one must have a name. You use this parm to specify the name you want

to use for the export file. (The naming rules are like those for file names (page 78). However, the report name must also be valid for use as a DDNAME.) The name may not be the name of a file, table or workarea in the program. Use this same report name (again in parentheses) later in the PRINT statements for this export file.

When using this parm, you must also supply a DD with this same report name in your execution JCL. The export file will be written to that output DD.

COLSEP('xxx'/',')

Specifies a separator text that should appear between each column of the export file. When not specified, the default is to put a comma between columns in the export file.

For example, the following statement would cause the export file to be "tab-delimited" rather than comma-delimited. (Microsoft Excel can import tab-delimited files.)

```
EXPORT (name) COLSEP(X'05')
```

CURRCHAR('x'/'$')

Specifies the character to use as the currency symbol in this export file. Items formatted with the CURRENCY display format will use this character. (Display formats are listed on page 117.) You may specify any 1-byte

FORMAT(display-format ...)

Specifies one or more default display formats to use in this export file. Normally, you should only specify a *date* display format with this parm. z/Writer chooses the character and numeric display format needed to make a comma delimited export file.

A complete list of the display formats available appears in the table on page 117.

Note that this parm only specifies the *default* display format(s) to use. Different display formats can still be specified for individual fields or columns in the report. (Do that with a FORMAT parm directly in the PRINT statement.)

NOCOLHDGS

Suppresses column headings in the export file. By default, export files get columns headings based on the fields in the first PRINT statement (for that output) found in the program. (See page 14 for a discussion of column headings.)

NOGRANDTOTALS

Suppresses grand totals from appearing at the end of an auto-cycle export file.

QUOTECHAR('x'/'"')

Specifies the character to use as the quote character for this export file. Character and date fields in the export file will be surrounded with this character. You may specify any 1-byte character or hex literal here. The default quote character is the double quotation mark.

EXAMPLES

```
FILE EMPL TYPE(KSDS)
...

EXPORT FORMAT(YYYY-MM-DD)
```

```
PRINT EMPL_NAME HIRE_DATE STATUS
EXPORT (PARTTIME)
PRINT (PARTTIME) EMPL_NAME STATUS
```

FETCH Statement

PURPOSE

Requires z/Writer's DB2 Option. Fetches one row from the DB2 result table created by a DB2 cursor. (The FETCH statement is the DB2 equivalent of a READ statement.)

SYNTAX

FETCH STATEMENT SYNTAX

```
FETCH   cursorname
```

DISCUSSION

The FETCH statement, together with an earlier CURSOR statement, is how DB2 data is accessed in a z/Writer program. The only parm, which is required, is the name of the cursor to fetch from. It must have been defined in an earlier CURSOR statement.

Before the first FETCH from a cursor, z/Writer first opens the cursor (unless it has already been opened with an explicit OPEN statement.)

Status of a Fetch Operation

After a FETCH statement, you can check the result of the operation by examining the cursor's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the cursorname, if the phase has definitions for multiple files, cursors and tables.) The table below shows the possible values of #STATUS after a FETCH.

#STATUS Built-In Field Values After a FETCH Statement

Y - file successfully read
N - no record read. For sequential reads, this normally indicates EOF; for keyed reads, this normally indicates that no record matching the key (or partial key) was found.

You can also test for "end of file" (no more rows) after a FETCH using the file's #EOF built-in field. The table below shows the possible values of #EOF after a FETCH.

#EOF Built-In Field Values After a FETCH Statement

Y - file has reached the end-of-file
N - file has not reached end-of-file

In addition, DB2 cursors also have a special #SQLCODE built-in field. This numeric field contains the status code returned by SQL for the previous DB2 operation. Check your SQL documentation to find the meaning of these codes.

PARMS

cursorname

Specifies the name of the DB2 cursor to fetch from. The cursor must have been previously defined using a CURSOR statement.

FIELD Statement

PURPOSE

Defines the properties of one field in the record area of a file or table, or in a work area. Before a field can be referred to in any other control statement, it must first be defined using a FIELD statement.

SYNTAX

FIELD STATEMENT SYNTAX

```
[FIELD] fieldname/FILLER
        [ n / tn / tn.n                                          ]
        [ LEN(n)                                                 ]
        [ TYPE(datatype/CHAR)                                    ]
        [ COL/DISP(n or +/-n or fieldname)                       ]
        [ CONTAINS(contains-type))                               ]
        [ DEC(n/0)                                               ]
        [ DIM(n [,n] ...)                                        ]
        [ FORMAT(display-format)                                 ]
        [ HEADING('line1|line2|...')                             ]
        [ INDEX(indexfld [,indexfld] ...)                        ]
        [ INIT(value)                                            ]
        [ LJ/CJ/RJ                                               ]
        [ REINIT(value)                                          ]

Abbreviations and Alternative Spellings Allowed:
FIELD - FLD
FORMAT - FMT
HEADING - HDG
```

Note: the statement name itself ("FIELD" or "FLD") is optional. You may omit it if you prefer. Both of the following statements are valid and accomplish the same thing (as long as they appear after a FILE, WORKAREA or TABLE statement):

```
FIELD AMOUNT N5.2
AMOUNT N5.2
```

DISCUSSION

The FIELD statement provides certain essential information about a field, such as where it is located in a record, how long it is and the type of raw data it contains. Optionally, the FIELD statement can also specify certain reporting options to be used when the field appears in a report (using the PRINT statement.) These options include the columns headings to use, how the raw data should be formatted, and more.

FIELD statements must immediately follow the FILE, TABLE or WORKAREA statement to which they pertain. You may have as many FIELD statements as you like.

FIELD statements are often kept in a copy library, and copied into reports as needed using a COPY statement.

The name of the field being defined is the first item in the FIELD statement. The next item (and the only other required item) must specify the **field's length**. The length can be specified in one of two ways:

- using the special shorthand notation (page 71). Examples: 10 or N10 or P5.2
- using the LEN parm. Example: LEN(10)

If the data type is character, no other parms are required. For non-character fields, the **data type** must be specified next. You can do this in the shorthand notation (page 71), together with the length. Or you can do it immediately after the length in a TYPE parm.

After fieldname, length and data type, all other parms are optional and may appear in any order.

- Note that the presence or absence of decimal information determines whether numeric fields are **totalled** in the report. If you want a numeric field to be totalled, be sure to specify the number of decimal digits it contains, even if that number is zero (with a DEC(0) parm or a shorthand notation such as N7.0).

PARMS

fieldname/FILLER

Specifies the name of the field being defined. All other control statements will use this name when referring to the field. You may assign any name you like within the naming conventions in the box below.

Requirements for Field Names

- field names may be from 1 to 70 characters long
- field names may contain alphanumeric characters, the "national" characters #, @, and $, underscores (_) and dashes (-).
- field names must not begin with a numeric character
- field names are not case sensitive
- field names must not be statement names or other reserved words

Examples

```
J
EMPL_NUM
EMP#
SMF30-ENCLAVE-CPU-TIME-IN-HUNDREDTHS-OF-SECONDS
```

Note: you may also specify the special word FILLER as the fieldname. Fillers fields are not actually defined and may not be referenced in the report.

n / Tn / Tn.n *(shorthand specification of type/length/decimal)*

You can specify the length (and optionally also a data type and decimal parm) using a special shorthand notation. It is not required to use this shorthand notation-- you can also use explicit LEN (and TYPE and DEC) parms as needed. Whichever syntax you choose, **some parm specifying the field length must immediately follow the fieldname.**

The table below explains the three formats allowed in the shorthand notation.

SHORTHAND NOTATION FOR DEFINING FIELDS		
SYNTAX	**DESCRIPTION**	**EXAMPLE**
n	Defines a **character field** of "n" bytes.	`FLD EMPLNAME 20`
Tn	Specifies a "**data type**", and the **length** of the field in bytes. The field, if numeric, contains no decimal digits.	`FLD EMPLNUM N3` `FLD EMPLNAME C20`
Tn.n	Specifies a (numeric) "**data type**", the **length** of the field in bytes, and the number of **decimal digits** that the field is understood to contain. Note that the field length is specified in *bytes*, while the decimal digits is the number of *digits*.	`FLD AMOUNT P5.2`
Note: a complete list of "data types," and the lengths allowed for each, can be found in the table under the TYPE parm below.		

LEN(nnn)

Specifies how many bytes the field occupies in the record or work area. Field length can be specified either with this explicit parm, or using the shorthand notation described earlier. Whichever method is used, some parm specifying the field length must immediately follow the fieldname. The lengths allowed for a field depend on its data type. The table under the TYPE parm below shows the lengths allowed for each data type.

Note: for FILLER fields only, a length of 0 is also allowed.

TYPE(datatype)

Specifies how the raw data is stored in the record or work area. The field's data type can be specified either with this explicit parm, or using the shorthand notation described earlier. When used, the explicit TYPE parm should immediately follow the length specification.

The following table shows the complete list of data types, the abbreviations allowed, and their acceptable lengths.

DATA TYPES			
DATA TYPE	**DESCRIPTION**	**LENGTH ALLOWED**	**COBOL EQUIV.**
C **CHAR**	Character data. Each byte may contain any 8-bit value. Specifying this type is optional. When no data type is specified, character data is assumed.	1 to 2,147,483,647 bytes	`PIC X`

DATA TYPES			
DATA TYPE	**DESCRIPTION**	**LENGTH ALLOWED**	**COBOL EQUIV.**
N **NUM**	Numeric data in "zoned" format. The first nibble of the last byte contains the sign information.	1-31 bytes	PIC 9
NE **NUMEDIT**	Numeric data in "edited" numeric format. Leading and trailing blanks are allowed; a single leading or trailing + or - sign is allowed; embedded punctuation is ignored; a single decimal point is allowed. An all-blank field is treated as zero.	1-256 bytes	none
P **PACK**	Numeric data in signed, packed format. The last nibble of the last byte contains the sign information.	1-16 bytes	COMP-3
PU	Numeric data in unsigned packed (also called BCD) format. The field does not contain sign information and the value is always considered positive.	1-16 bytes	none
B **BIN**	Numeric data in signed, binary format. The first bit of the first byte contains the sign information.	1-8 bytes	PIC 9 COMP SIGNED
BU	Numeric data in unsigned binary format. The field does not contain sign information and the value is always considered positive.	1-8 bytes	PIC 9 COMP UNSIGNED
F **FLOAT**	Numeric value in IBM's hexadecimal floating point representation. These fields contain a sign in the leading bit, a hexadecimal exponent in the next 7 bits, and the digits of a hexadecimal number in the remaining bytes. Note that while lengths up to 16 bytes are accepted, z/Writer actually uses only the first 8 bytes (14 hexadecimal digits) of the field. Any additional trailing digits are truncated.	2-16 bytes	COMP-1 COMP-2

COL/DISP(n *or* +/-n *or* fieldname)

Specifies the field's beginning column (or displacement) within the record or work area.

By default, the first field in a record or work area begins in column 1. And, by default, all other fields are assumed to begin immediately after the previously defined field. Use one of these parms if you want to override a field's default starting location.

When specifying a fixed location ("n"), you may use either COL or DISP, whichever is more convenient for you. The first byte of a record is COL(1) or DISP(0). (When using the "+/-n" or "fieldname" syntax, the COL and DISP parms both yield the same result.)

Note: for variable length QSAM records (V or VB) the first four bytes of the record are called the record descriptor word (RDW). Thus, the first actual data byte in such records is in column 5 (displacement 4).

Note: another way to specify the location of a field is with a REDEFINE statement.

The table below explains the three ways that you can specify a field's starting location within the COL and DISP parms.

COL AND DISP PARM SYNTAX		
SYNTAX	**DESCRIPTION**	**EXAMPLE**
n	The field starts in the column or displacement specified.	`FLD EMPLNAME 20 COL(11)` `FLD EMPLNAME 20 DISP(10)`
+/-n	The field will start 'n' bytes *before* (-) or *after* (+) its default starting position. Use this syntax if you want to skip over (or back up and redefine) a portion of the record.	`FLD DATE-DD 2 COL(-2)` `FLD NAME 20 COL(+35)`
fieldname	The field starts in the same byte that the named field starts in. (The named field must have already been defined, within the same record.)	`FLD DATE-YY 2 COL(DATE)`

Note: You can use the COL/DISP parms to backup and redefine an earlier portion of the record. However, there is an important difference between redefining an area with the COL/DISP parm as compared to using a REDEFINE statement. If you use a REDEFINE statement to redefine an array field or a field with indexes, the new fields that redefine the original field will also inherit its DIM and INDEX parms. Those properties are not inherited when you use the COL/DISP parm to redefine an earlier part of the record.

CONTAINS(contain-type))

This optional parm describes the contents of a CHAR or NUM field in more detail. That is, it provides z/Writer a "hint" as to how the field is used in your program logic. Use this parm when defining fields that contain date values, if you would like z/Writer to assist you with them. The valid values for this parm are listed in the table below. Here is an example:

```
FIELD SALES-DATE C8 CONTAINS(YYYYMMDD)
FIELD HIRE-DATE N6 CONTAINS(YYMMDD)
```

Note: only CHAR and NUM fields may use the CONTAINS parm. The field length must also match the implied length of the contains-type. For example, a C6 or N6 field may contain YYMMDD data. And a C8 or N8 field can contain a YYYYMMDD date.

When you define a field that contains a date contains-type, z/Writer does these things differently.

- when you print the field in a report, z/Writer automatically formats it as a date for you. It uses the default date display-format, which is MM/DD/YY. (You can change the date display-format used as the default, by specifying a FORMAT parm in an OPTION or REPORT statement. For example, if you would like all dates in your report to appear in DD/MM/YY format by default. you could use this statement:

```
REPORT FORMAT(DD-MM-YY)
...
PRINT REGION START-DATE END-DATE
```

- you are also allowed to specific your own date display-format for the field. This can be done using the FORMAT parm of the FIELD statement, or directly in a PRINT or TITLE statement. See the table on page 117 for a complete list of display formats.
- for all statements *other* than the PRINT and TITLE statements, the fields with CONTAIN parms are simply treated as any other character or numeric field. (For example, in MOVE statements, IF statements, COMPUTE statements, etc.) The CONTAIN parm *only* affects how the field is formatted in the report.

CONTAIN-TYPES (ALLOWED IN THE CONTAINS PARM)		
CONTAIN-TYPE	**DESCRIPTION**	**REQUIRED FIELD LENGTH**
YYYYMMDD	The field contains a date in YYYYMMDD format. It does not include any delimiters.	8
YYMMDD	The field contains a date in YYMMDD format. It does not include any delimiters. If you display this field with a date display format that shows 4 digits for the year, z/Writer assigns the century bytes (either 19 or 20) for you. It does this by comparing the YY portion of the date to the century-cutoff value. (That value is 50 by default, and can be overridden with a CENTURY parm in an OPTIONS statement.) When YY is *greater than* the cutoff value, '19' is assigned as the century digits. Otherwise, '20' is assigned.	6

DEC(n/0)

Specifies how many of the digits in the numeric field are considered to be decimal digits. When not specified, zero decimal digits is assumed. The number of decimal digits can be specified either with this explicit parm, or using the shorthand notation described on page 71.

Note: the DEC parm has a slightly different meaning for fields with the FLOAT data type. By definition, float fields have a floating "decimal" point (technically a "hexadecimal" point) in their raw format, and thus a varying number of "decimal" digits. So for these fields, the DEC parms does not specify the number of "decimal" digits in the raw data. Instead, the DEC parms specifies how many decimal digits to preserve when the raw data is converted from floating point to fixed point decimal (for internal processing or to display in the report).

Note: the presence or absence of decimal information determines whether numeric fields are totalled in reports. If you want a numeric field to be totalled, be sure to specify the number of decimal digits it contains, even if that number is zero.

DIM(n [,n] ...)

The presence of this parm indicates that the field being defined is an array. Use this parm to specify the number of elements in each dimension of the array. You must use numeric literals within this parm — field names are not allowed. z/Writer allows you to specify up to 2,147,483,647 elements per dimension, however memory constraints may limit this as a practical matter.

z/Writer supports up to 32,767 dimensions.

Subscript Notation for Arrays

In your program code, fields defined with the DIM parm normally must be referenced using a subscript. Subscripts are placed in square brackets after the field name. (The subscript must *not* be separated from the fieldname with a space.) Within the brackets, there must be one element specifier per defined dimension of the field. The first element of each array dimension has subscript "1". Each subscript may be specified as either a numeric literal, a numeric field, or a numeric expression. A numeric field used as a subscript may itself be subscripted.

Following is an example of defining a one-dimensional array, and using it in a MOVE statement:

```
FLD ADDR_LINE 30 DIM(4)
...
MOVE ADDR_LINE[3] TO OUT_ADDR
```

Inheritance of the DIM Parm

If you use a REDEFINE statement to redefine an array field, the new fields that redefine the original field will also inherit its DIM parm. (This applies to all fields until either an ENDREDEF statement is encountered, or until you have redefined up to the end of the original array field.) Thus, you will also need to use subscripts when referencing any field that redefines a part of an array field.

For example:

```
FLD DATE 6 DIM(10)

REFEDINE DATE
FLD DATE-YY 2
FLD DATE-MM 2
FLD DATE-DD 2
ENDREDEFINE
...
MOVE DATE-MM[8] TO WORK2
```

FORMAT(display-format)

Specifies how the field's raw data should be formatted (by default) when it is printed in a report (by a PRINT or TITLE statement.) See the table on page 117 for a complete list of display formats. (This default can still be overridden with a different display format specified directly in the PRINT or TITLE statement.)

HEADING('line1|line2| ...')

Specifies the default column headings to use when printing this field in a report (using the PRINT statement.) (This default can still be overridden with different column headings directly in the PRINT statement.)

Use one or more "|" characters within the text to indicate how the text should be broken onto multiple column heading lines. Each column heading is centered over the column by default. So you do not need

to include pad characters in the text to help center it. (If you *do not* want centered headings, you can use padding characters on the left or right of the heading text, as necessary.)

INDEX(indexfld [,indexfld] ...)
Specifies one or more "index fields" to use when locating the field within a record or workarea. In this parm, you may name a previously defined numeric field. Or you can provide a valid fieldname that has not yet been defined. In that case, z/Writer defines it for you as a new 4-byte binary field. Each time you reference a field that was defined with an INDEX parm, the values of all of its index fields are summed together and added to the field's defined location (that is, the location defined by its COL or DISP parm, if any.) This aggregate value determines the field's location within the record or workarea at that point in the program.

Note: one use of indexes is to work with arrays. In most cases, however, specifying the DIM parm (and using array subscripts) is a much easier way to work with arrays.

Inheritance of the INDEX Parm

If you use the REDEFINE statement to redefine a field that has an INDEX parm, the new fields that redefine the original field will also inherit that INDEX parm. (This continues until either an ENDREDEF statement is encountered, or until you have redefined up to the end of the original indexed field.)

INIT(value)
Specifies an initial value to be placed in the field before the program begins execution. The value must be a character literal for character fields, or a numeric literal for numeric fields. For arrays, (fields defined with the DIM parm), all occurrences of the field will be initialized to the given value.

If no INIT or REINIT parm is specified, fields defined *for a WORKAREA* will automatically be initialized to:

- numeric zero, for all types of numeric fields
- blanks, for other fields

Files, DB2 cursors and tables all have their entire record area initialized to blanks once before program execution. So, when no INIT or REINIT parm is specified, their fields will initially contain blanks regardless of data type.

LJ / CJ / RJ
Specifies how the data should be justified (within its report column) when it is printed in a report (using the PRINT statement.) The values mean left-justified, center-justified and right-justified, respectively.

REINIT(value)
Specifies an initial value to be placed in the field before the program begins execution, just like the INIT parm.

This parm is only meaningful for auto-cycle reports. The difference from INIT is that the initial value is *again* placed in the field at the start of each cycle of the report is performed. That is, just before the next record is read from the primary input file and the program execution begins again. Read more details under the INIT parm above.

EXAMPLES

```
FIELD EMPL-NUM 3   INIT('999')
FIELD AMOUNT N5.2  INIT(0)
FLD LAST-NAME 20
FLD LAST-NAME C20  CJ
FLD LAST-INITIAL 1 COL(LAST-NAME)
FLD FILLER 19
FLD DEPT-NUM N3
FLD SALES-AMOUNT P8.2 FORMAT(PIC'$$$,$$9.99')
FLD LOOP-COUNTER B2
FLD FLOAT-AMOUNT F4.2  HEADING('TRANSACTION|AMOUNT')
```

FILE Statement

PURPOSE

This non-executable statement simply defines the characteristics of a single file to z/Writer. Before a file can be used in a run, it must first be defined using the FILE statement.

FEATURES

Use the FILE statement to:

- define the **type of file** (for example, whether it's VSAM or QSAM)
- specify whether the file is an input or output file, or whether it will be updated (used for both input and output)
- tell how you want to access a VSAM file (for example, sequentially or randomly.)
- specify that the file should be pre-sorted before executing the user program

SYNTAX

FILE STATEMENT SYNTAX

```
FILE    filename
        [ INPUT/OUTPUT/UPDATE                                          ]
        [ TYPE(SAM/ESDS/KSDS/RRDS/TEMP/CARD)                           ]
        [ F/FB/V/VB[(nnnnn)]                                           ]
        [ SEQ/DIR/SKIP                                                 ]  VSAM only
        [ PRESORT(fieldname[(ASC/DESC)] ... [#EQUALS])                 ]
        [ UNIT(xxxxx/SYSDA                                             ]  TEMP only
        [ SPACE(CYL/TRK/nnnnn, nnn [,nnn])                             ]  TEMP only


Abbreviations Allowed:
ASC - A
DESC - D
```

DISCUSSION

This statement by itself does not perform any I/O operation on the file. This statement simply defines a filename to z/Writer so that subsequent control statements can refer to that file. After a file has been defined using this statement, READ, WRITE and various other statements may be used to perform I/O operations on the file.

You may have as many FILE statements as you like in a run. Their order is not important — except in one case. If you are producing an auto-cycle report (see page 9), be sure that the first file you define is the file that z/Writer should read automatically.

For most file definitions, you will need one DD statement in the JCL for each file that you define and use in a run. However, CARD type files use the SYSIN DD (which is also used for the program control statements.) And TEMP type files do not require any DD at all.

The fields for the file must be defined immediately after the FILE statement. Use FIELD statements to define the fields. Often the FILE and FIELD statements for a file are kept together in a "copy library" member. (Then simply use a COPY statement to copy them into programs as needed.)

The entire record area for files is initialized to blanks before program execution. However, individual fields within the record area can be initialized to your own value, using the INIT parm in the FIELD statement.

When a file is defined, z/Writer automatically creates a field by the same name (as the file). It is defined as a character field whose length is the defined length of the file. You may use this field just like any other character field (for example, in a MOVE statement.)

z/Writer also maintains certain built-in fields for each file automatically. Some of these fields are "read-only." Read-only means that you can examine the contents of the field, but not modify it. The built-in fields for each file are shown in the table below. Note that if your program has more than one file, you will need to qualify these built-in field names with the file name in order to uniquely reference them. (For example, SALES.#EOF).

Built-In Fields Available for Files

FIELDNAME	TYPE & LENGTH	READ ONLY	DESCRIPTION
#COUNT	4-byte binary	Yes	Number of successful reads from an INPUT or UPDATE file, or number of successful writes for an OUTPUT file.
#EOF	1-byte character	Yes	Indicates that the most recent operation attempted to read past the last record in the file ("end of file"). Y = the last I/O operation was a POSITION or READ statement which raised the end of file condition N = the last I/O operation was not a POSITION or READ statement which raised the end of file condition
#RECSIZE	4-byte binary	No	After a READ, this field contains the length of the record just read. Before a WRITE or an REWRITE, the user should ensure that this field contains the (new) length of the record.

Built-In Fields Available for Files			
FIELDNAME	TYPE & LENGTH	READ ONLY	DESCRIPTION
#RRN	4-byte binary	Yes	*Defined only for RRDS files.* Stands for "relative record number". After a non-keyed READ to a RSDS file, z/Writer places the RRN of the record that it returns here. After a non-keyed WRITE, z/Writer places the RRN of the record written here. (For keyed reads and writes to RRDS files, the user specifies the desired RRN using the KEY parm of the relevant statement, not this built-in field.)
#STATUS	1-byte character	Yes	Indicates the status of the file after each I/O operation. **(blank)** = uninitialized (no operations performed on file yet) **Y** = last operation on file was successful **N** = last operation on file was unsuccessful. After a keyed read, it usually indicates a missing record. After a sequential read, it usually indicates EOF. Or it can indicate that some other error occurred while trying to perform the operation.

PARMS

filename

Specifies the name of the file being defined. All other control statements will use this name when referring to this file.

Note: in most cases, the filename specified here must also be the DDNAME of a DD in the execution JCL. Files of type CARD and TEMP are exceptions.

You may assign any name you like following the naming conventions in the box below.

Requirements for File Names

- file names may be from 1 to 8 characters long
- file names may contain alphanumeric characters and the "national" characters #, @, and $
- file names must not begin with a numeric character
- file names are not case sensitive
- file names must not be statement names or other reserved words
- file names must not be the name of a workarea, table or DB2 cursor

Examples

```
SALESIN
SALESOUT
MERGE2
```

INPUT/UPDATE/OUTPUT

Specifies the type of processing that will be performed on the file. If you will only be reading from the file (whether sequentially or randomly) specify INPUT. INPUT is also the default if you omit this parm.

If you will only be writing new records to a file, specify OUTPUT. To read records, and then possibly modify or delete those records, specify UPDATE.

TYPE(QSAM/ESDS/KSDS/RRDS/TEMP/CARD)

Specifies the type of access method to use when reading this file. If not specified, QSAM is assumed. Valid file types are listed in the following table:

FILE TYPES ALLOWED IN THE TYPE PARM	
FILE TYPE	**DESCRIPTION**
QSAM/ SAM	Standard sequential files, including tapes and disk datasets. The QSAM access method will be used. QSAM files may not be used with UPDATE processing.
ESDS	A VSAM ESDS file. The IDCAMS access method will be used. The file is always processed sequentially.
KSDS	A VSAM KSDS file. The IDCAMS access method will be used. The file may processed sequentially, randomly, or a combination of both of those (i.e., random access(es) followed by sequential access(es).)
RRDS	A VSAM RRDS file. The IDCAMS access method will be used. The file may processed sequentially, randomly, or a combination of both of those (i.e., random access(es) followed by sequential access(es).)

FILE TYPES ALLOWED IN THE TYPE PARM (CONTINUED)	
FILE TYPE	**DESCRIPTION**
TEMP	Defines a temporary, sequential file to be accessed using QSAM. Since z/Writer dynamically allocates TEMP files, you do not need to supply a DD statement for these files in your JCL. Typically, a FILE statement first defines an OUTPUT type TEMP file. Records are written to the temporary file in that phase. Then, a later phase in the same program specifies a REUSE statement to define the same temporary file as an input file. That phase will read the records written by the earlier phase.
CARD	The SYSIN file. The first records read from the SYIN file contain the program statements. Once the DATA statement has been reached in SYSIN, no more programming statements are read. All records after the DATA statement are data records. They are read as data when the program executes a READ to the CARD file.

F/FB/V/VB[(nnnnn)]

Specifies whether the records in the file are *fixed* (F/FB) or *variable* (V/VB) in length. If this parm is not specified, fixed length records are assumed for QSAM files, while variable length is assumed for VSAM files.

Optionally, you may also specify the **record length** in parentheses immediately after the format character(s). For all variable length files, specify the length of the largest record that could be encountered during the run. For variable length QSAM files, be sure that the length includes the 4 bytes used as the "record descriptor word" (RDW) at the beginning of each record. When no length is specified, the length implied by the fields defined after the FILE statement is used as the record length.

Note: when this parm is omitted, z/Writer assumes a fixed-length file whose records are the length indicated by the field definitions that follow.

Note: for fixed length VSAM files, specifying “F” here will improve the efficiency of any sort performed on that file.

Note: CARD files are always treated as fixed length files of record length 80. You can omit this parm to default to this. However, you may not specify any different record length, nor the V or VB format.

Note: it is not important to distinguish between F and FB, or between V and VB files. Both the F and FB parms mean fixed record length; both the V and VB parms mean variable length. The traditional “B“ formats are also allowed purely for convenience.

SEQ/DIR/SKIP *(allowed only for* VSAM *files)*

Specifies the manner in which a VSAM file will be accessed. The following table explains the meanings of the values for this parm:

ACCESS TYPE PARM	
FILE TYPE	**DESCRIPTION**
Values Permitted for VSAM **files defined as INPUT or** UPDATE	
SEQ	Files will be read in sequential order. This is the default and the only value permitted for ESDS files. For KSDS and RRDS files, specifying this parm implies that no keyed reads will be performed to the file during the run.
DIR	*For KSDS and RRDS files only.* This is the default for KSDS and RRDS files. It means that the file may be accessed directly (using a key) and/or sequentially. Any keys specified in the READ statement can be in random order. A READ statement with a KEY parm results in a direct read, while a READ statement without a KEY parm results in a sequential read.
SKIP	*For KSDS and RRDS files only.* This parm means that any keyed READ statement will use skip access. The keys specified in the READ statement *must be* in key sequence order. This access method can be more efficient when used correctly. But it will result in an I/O error if keyed reads are attempted that are not in key sequence. Note that sequential reads (from the last record read) are also permitted with this parm, and can be mixed among skip reads. (A READ statement without a KEY parm results in a sequential read.)
Values Permitted for VSAM **files defined as** OUTPUT	
SEQ	Files will be written in sequential order. This is the default for all output files. It is also the only value permitted for ESDS files. When writing to (loading) empty VSAM files, this SEQ access method must be used.
DIR	*For KSDS and RRDS files only.* This parm means that all writes to the file will be performed using direct access. The keys contained within the record being written can be in random order. This parm must not be used to load an empty file.
SKIP	*For KSDS and RRDS files only.* This parm means that all writes to the file will be performed using skip access. The keys contained within the record being written *must be* in key sequence order. This access method can be more efficient when used correctly. But it will result in an I/O error if writes are attempted that are not in key sequence. This parm must not be used to load an empty file.

UNIT(xxxxx/SYSDA)

Allowed only for file's of type TEMP. Specifies the z/OS UNIT to use when allocating this temporary file. The default is SYSDA.

SPACE(CYL/TRK/nnnnn nnn [,nnn])

Allowed only for file's of type TEMP. Specifies the units and amount of disk space to request when allocating this temporary file. The default units to request is CYL (cylinders). Other choices for unit are TRK (tracks) and nnnnn (blocks of size nnnnn).

The next parm(s) specify how many such units to allocate for primary (and optionally secondary) allocation.

If you know that a large number of records will be written to your temporary file, you should use this parm to specify that a larger number of cylinders (than the default) be allocated.

PRESORT(fieldname[(ASC/DESC)] ... [#EQUALS])

When the PRESORT parm is specified, z/Writer presorts the whole file before beginning to execute the user program. Thus, when the program reads records from the file, theose records are read in sorted order. (The net effect is just the same as if you had sorted the input file in a separate, earlier job step.) The PRESORT parm is allowed only for INPUT files.

Note: only one file per phase may be defined with a PRESORT parm.

The PRESORT parm can name any number of sort fields to use. (All fields must come from the file being defined.) The field specified first is the primary sort field; the next field is the secondary sort field and so on.

By default, the file will be sorted in ascending sequence on each field. You may specify DESC (or just D) after a fieldname to sort that field into descending sequence. (You may also specify ASC or A for ascending sequence, although that is the default.)

The final, optional #EQUALS parm indicates how to sort multiple records, all of whose sort fields contain the same value. When #EQUALS is specified as the last item in the PRESORT parm, those duplicate-key records will remain in the same relative order as in the unsorted file. When this parm is not specified, the system Sort program will decide how to order such duplicate records.

EXAMPLES

```
FILE SALES
FILE SALES TYPE(ESDS)
FILE EMPL TYPE(KSDS) V(150) SKIP
FILE STATUS-CODES TYPE(CARD)
FILE JOBNUMS TYPE(TEMP)
FILE EMPL PRESORT(LAST_NAME FIRST_NAME #EQUALS)
```

GOTO Statement

PURPOSE

Unconditionally transfers execution of the program to the statement having the specified label.

SYNTAX

GOTO STATEMENT SYNTAX

```
GOTO label
```

PARMS

label

Specifies the program statement to be executed next. (Labels are optional names, appended with a colon, that precede a statement itself.) Labels must follow the naming conventions in the box below.

Requirements for Statement Labels

- labels may be from 1 to 70 characters long
- labels may contain alphanumeric characters, the "national" characters #, @, and $, underscores (_) and dashes (-).
- label names are not case sensitive
- label names must not begin with a numeric character
- labels must end with a colon (which is not part of the label name itself)

Examples

```
WRAPUP:
A-100-WRITE-OUTPUT:
A-100-WRITE-OUTPUT-EXIT:
```

EXAMPLES

```
IF EMPL.#STATUS = 'N' GOTO WRAPUP
...
WRAPUP: PRINT 'END OF RUN'
```

IF Statement

PURPOSE

The IF statement begins an “if-structure”. The purpose of an if-structure is to conditionally execute (at most) one set of statements within the structure.

SYNTAX

IF STRUCTURE SYNTAX

```
IF conditional-expression
other statements

[ ELSEIF conditional-expression
other statements  ] ...

[ ELSE
other statements ]

ENDIF
```

DISCUSSION

An if-structure always begins with an IF statement and ends with an ENDIF statement. Within the structure, one or more optional ELSEIF statements are allowed. And a final, single ELSE statement may also be used. IF statements may be nested to any level.

This is how if-structures are processed.

- The IF statement, as well as any ELSEIF statements contain a conditional expression.
- Those conditional expressions are evaluated in turn, until the first one is found that is true.
- When a conditional expression is “true”, the “other statements” that follow are then executed; after that, control passes to the first statement after the ENDIF statement. (You are allowed to have zero “other statements,” in which case control is immediately passed to the first statement after the ENDIF statement.)
- If none of those conditional expressions is true, then one of the following occurs:
  - if there is a final ELSE statement within the if-structure, the “other statements” following it are executed; after that, control passes to the first statement after the ENDIF statement.
  - otherwise, control simply passes to the first statement after the ENDIF statement.

Note: statement labels are not permitted within if-structures.

Conditional Expressions

As mentioned, the IF statement, and any ELSEIF statements, simply contain a "conditional expression." A conditional expression is just one or more "tests" (or comparisons) between fields and/or literals.

CONDITIONAL EXPRESSION SYNTAX

Conditional expressions take the form:

```
[NOT] test [ AND/OR [NOT] test ] [ AND/OR [NOT] test ]  ...
```

Parentheses can be used to group two or more tests (or parenthesized groups of tests) together. Nesting is allowed to any level. When priority is not indicated by parentheses, "AND"ed tests are performed before "OR"ed tests at the same level.

If the conditional expression has more than one test, those tests must be connected using the keywords AND and OR. Tests can also be grouped within parentheses, and nested to indicate the desired order of evaluation. You may also negate either a single test or a parenthesized group of tests by preceding it with the keyword NOT.

The use of "AND", "OR", "NOT" and parentheses is the same as in all common programming languages (including COBOL and BASIC) and will not be elaborated upon here.

Test Syntax

Now let's look at the syntax for the individual tests themselves. z/Writer supports a few syntactical features that are not standard in other languages.

TEST SYNTAXES

```
OPERAND1 COMPARATOR OPERAND2
OPERAND1 COMPARATOR OPERAND2 THRU OPERAND3
OPERAND1 COMPARATOR OPERAND2 [OPERAND3]  [OPERAND4]  ...
```

Syntax of a Simple Test

The syntax of a simple test is:

```
OPERAND1 COMPARATOR OPERAND2
```

Operand1 can be:

- a field name
- a numeric or character literal
- a built-in function

- a computational expression, as long as it does not begin with an open parenthesis

Operand2 can be:

- a field name
- a numeric or character literal
- a built-in function
- a computational expression (which may begin with an open parenthesis)

The **comparator** can be any symbol from the following table.

COMPARATORS ALLOWED IN CONDITIONAL EXPRESSIONS	
SYMBOL	MEANING
= EQ	Equal to
¬= <> NE	Not equal to
< LT	Less than
<= LE ¬>	Less than or equal to
> GT	Greater than
>= GE ¬<	Greater than or equal to

Here is an example of a simple test in an IF statement:

```
IF AMOUNT = SELECT-AMOUNT
IF AMOUNT GT 100.00
```

Data Types of Operands

Normally, both of the operands involved in a test will be of the same general data type. That is, both will be character data or both will be numeric data. (With numeric comparisons, you are allowed to mix different specific numeric data types. For example, you may compare a binary numeric field with a packed numeric field, etc.)

However, z/Writer will also allow you to compare a numeric field with a character field. In this case, a character-by character comparison of the fields is performed (as if both operands were character data).

This can be useful when checking for special values in numeric fields. For example:

```
IF PACKED-AMOUNT = X'FFFFFFFF'
```

Comparing an Operand to a Range of Values

Conditional expressions may also contain tests in this format:

```
OPERAND1 =/¬=  OPERAND2 THRU OPERAND3
```

This syntax compares operand1 to a *range of values*. For example:

```
IF AMOUNT = 100 THRU 199.99
```

The above test is true when the AMOUNT field contains any value between 100.00 and 199.99, inclusive.

Each of the operands can be either a field or a literal.

The comparator must be either "equals" ('=', EQ) or "not equals" ('¬=', '<>', NE). The other comparators may not be used with ranges. When "not equal" is used, the test is true as long as operand1 does not fall within the inclusive range specified.

Comparing an Operand to a List of Values

Conditional expressions may also contain tests in this format:

```
OPERAND1 =/¬= OPERAND2 [OPERAND3] [OPERAND4]  ...
```

This syntax compares operand1 to a *list of values*. You may separate the operands in the list with spaces and/or commas. For example:

```
IF AMOUNT = 0, 100, 9999999
```

The above test is true when the AMOUNT field contains any of these three values: zero, one hundred, or 9999999.

Each of the operands can be either a field or a literal.

The comparator must be either "equals" ('=', EQ) or "not equals" ('¬=', '<>', NE). The other comparators may not be used with operand lists. When "not equal" is used, the test is true as long as operand1 does not equal any of the operands in the list.

Combining Ranges and Lists

You may also combine lists and ranges within a single test. For example:

```
IF AMOUNT = 0, 5, 10, 100 THRU 199.99, 500, 1000 THRU 1999.99
```

Keyword Tests

z/Writer also supports one keyword test: NUMERIC. The format of this test is:

```
IF field NUMERIC
```

This test returns true if the tested field contains only: zero or more leading spaces, followed only by the characters '0' through '9'. It returns false: if any other character occurs within the field; if any spaces other than leading spaces are found; or if the field contains all spaces.

For example:

```
IF AMOUNT NUMERIC
    PRINT 'VALID AMOUNT FIELD'
ELSE
    PRINT 'INVALID AMOUNT FIELD'
ENDIF
```

Keyword Tests

You may also use the special keywords SPACE (SPACES) and ZERO (ZEROES, ZEROS) as “operand2” in your test. The keyword SPACES causes operand1 to be tested for all blanks, regardless of operand1’s data type. The keyword ZEROS tests for a value of zero in operand1’s data type. For character operand1, it is compared to all character 0’s. For numeric operand1’s, it is compared to a zero numeric value.

EXAMPLES

```
IF AMOUNT = TEST-AMOUNT
    PRINT 'EQUALS TEST AMOUNT'
ELSEIF AMOUNT = 400
    PRINT 'EQUALS 400'
ELSEIF AMOUNT = 500 THRU 599
    PRINT 'BETWEEN 500 AND 599 (INCLUSIVE)'
ELSEIF AMOUNT = 2, 4, 6
    PRINT 'EQUALS 2, 4 OR 6'
ELSEIF AMOUNT = 1, 3, 5, 700-799, 1000-1099, 9999
    PRINT '1, 3, 5, SEVEN HUNDRED SOMETHING, ONE THOUSAND SOMETHING OR 9999'
ELSE
    PRINT 'DID NOT EQUAL ANY OF THE ABOVE'
ENDIF

IF DEPT-NUM = '2' THRU '4' AND (AMOUNT < 100 OR AMOUNT > 5000)
    MOVE '2' TO STATUS-FLAG
    PRINT 'STATUS 2' EMPL-NUM
ENDIF
```

INCLUDE Statement

PURPOSE

This statement (allowed only in auto cycle runs) "includes" the current input record and terminates execution of the program for that record. "Includes" means that the fields in the record *will be* included in calculating the totals (printed at control breaks and at the end of the run.)

SYNTAX

INCLUDE STATEMENT SYNTAX

```
INCLUDE
```

DISCUSSION

This statement is allowed only in auto cycle runs (see page 9). When it is executed, the normal flow of the user program ends (for the current record). That is, any statements following the INCLUDE statement are not executed for the current record. Before reading the next input record, the auto cycle code *will* add the fields from this record to any totals that are being accumulated. Then it reads the next record and begins program execution from the beginning for that record.

This statement is similar to the EXCLUDE statement (page 62). However, the EXCLUDE statement terminates execution of the program for the current record *without* including the record in the totals.

There are no parms for the INCLUDE statement.

LISTOFF Statement

PURPOSE

Stops the listing of control statements (to the SYSPRINT DD) as they are read and processed. By default, control statements are listed as they are processed.

SYNTAX

LISTOFF STATEMENT SYNTAX

```
LISTOFF
```

EXAMPLES

```
FILE SALES
LISTOFF
COPY SALESDEF
LISTON
...
```

LISTON Statement

PURPOSE

Resumes the listing of control statements (to the SYSPRINT DD) as they are read and processed. (This is also the initial, default mode.)

Note: listing resumes with the next statement *after* the LISTON statement.

SYNTAX

LISTON STATEMENT SYNTAX

LISTON

EXAMPLES

```
FILE SALES
LISTOFF
COPY SALESDEF
LISTON
...
```

MACRO Statement

PURPOSE

Specifies character strings that should replace “macros” wherever they appear in the program code (and any instream data.) This statement is often used to pass variable data (such as selection criteria) into “model” programs.

SYNTAX

MACRO STATEMENT SYNTAX

```
MACRO  %name = value1
```

DISCUSSION

You may choose any name you like for a macro, as long as it follows these rules:

- macro names always begin with a percent sign (%)
- the remaining characters should follow the same naming rules as for fieldnames (see page 70)

The value that should replace all occurrences of the macro name is specified after the equals sign (=). You may specify your value in quotes or ticks. To embed that same character (quote or tick) *within* the string, use two of those characters together (just as with regular character literals). Note that the exterior quotes or ticks are not part of the value that will replace the macro name.

If your value is a simple one-word alphanumeric string, it is not necessary to enclose it in quotes or ticks.

After the MACRO statement for a given macro name, all future occurrences of that macro name in the SYSIN stream will be replaced by the value specified for it. This applies to all program statements, comments, quoted literal texts as well as any instream data that may follow the program (associated with a CARD file.) In the control listing, both the “before” and the “after” versions of the statement are listed.

MACRO statements are normally placed near the top of a program.

EXAMPLES

```
MACRO %SELECT-DATE = 20140101
MACRO %RATE = 0.00186
MACRO %NAME = "JONES, JR"
```

MOVE Statement

PURPOSE

Moves the contents of one field to another field, changing its data representation if required. When moving numeric fields, the number of decimal digits in the source field may also be altered to conform to the defined decimal digits of the target field.

SYNTAX

MOVE STATEMENT SYNTAX

```
 MOVE fieldname/literal TO fieldname

or

 MOVE CORR[ESPONDING] recordname TO recordname
```

DISCUSSION

You can use the MOVE statement in two ways:

- to move data from a single literal or field to a single target field
- to move data from all like-named fields in one record area (or workarea) to the correspondly named fields in another record area (or workarea)

Simple MOVEs

Use the first syntax above to move data from a single source to a single target. When moving numeric data to numeric field, z/Writer **converts the data**, if necessary, to the target format. (For example from a binary field to a packed field.) It also adjusts **decimal digits** (with rounding), if necessary, to match the target field's precision. When data conversion is performed, the source field is first validity-checked, to prevent S0C7's, etc. However, when the source and target field are the data type, the same length and have the same number of decimal digits, an simple "as-is" move is performed. In this case, source field data is not validity-checked.

If *either* the source or target field is character, then no data conversion is performed, even if the other field is numeric. For character moves, data is truncated or right-padded with blanks when the two fields are of different lengths.

You may use a single MOVE statement to assign a common value to **all elements of an array**. Just name the array as the target field without using any subscripts. (Of course, you can also move to just a single element of an array, by using a subscripted array as the target field.)

Note that you may also use this syntax to move to and/or from **whole record areas**. When a file is defined (with the FILE statement), z/Writer automatically creates a field by the same name (as the file). It is defined as a character field whose length is the defined length of the file. You may use this field just like any other character field in a MOVE statement.

In addition to standard numeric and character literals, you may also use these two special keywords as your source literal:

- SPACE or SPACES: The target field is completely filled with blanks, regardless of its data type.
- ZERO, ZEROS or ZEROES: sets the target field to the zero value appropriate for it data type. For character and character-numeric targets, the target is filled with character 0's. For packed or floating point targets, the field is set to a packed or floating point zero. For other numeric data types, the target field is filled with hex zeros.

MOVE CORRESPONDING

This syntax allows you to move a large number of fields from one record area to another record area. (A record name is the name of a defined field, table or WORKAREA.) Each field in the source record is moved to the like-named field in the target area, if such a field exists. Fields in the target record that do not have a corresponding field in the source record will not be changed.

EXAMPLES

```
MOVE LAST-NAME TO FIRST-NAME
MOVE 'JONES' TO LAST-NAME
MOVE BIN-AMOUNT TO PACK-AMOUNT
MOVE SPACES TO OUTPUT-RECORD
MOVE 0 TO AMOUNT-ARRAY
MOVE 999 TO AMOUNT-ARRAY[2,3]

MOVE CORR INPUT-RECORD TO OUTPUT-RECORD
```

NEWPHASE Statement

PURPOSE

Specifies that a new z/Writer phase follows. This statement marks the end of the previous phase and the beginning of a new one.

SYNTAX

NEWPHASE STATEMENT SYNTAX

```
NEWPHASE
      [ CENTURY(nn/50)              ]
      [ DATEDELIM('x','/')          ]
      [ FORMAT(display-format ...)  ]
      [ PICBASE2('x'/'@')           ]
      [ PICBASE10('x'/'#')          ]
```

DISCUSSION

Most z/Writer programs do not require multiple phases. For example, you can produce multiple reports in a single phase. (See "Printing Multiple Reports" on page 13.)

The main reason for a multi-phase program is when you need to use the *output* from one phase as the *input* to a subsequent phase. For example, if you need to print percent of total, you will need an earlier phase to compute the totals and pass that (in a temporary file) to the next phase. The second phase will then be able to calculate percent of total for each detail record as it is read again.

When you begin a new phase, everything from the previous z/Writer is forgotten (except for global OPTION statement options.) None of the fields or files from the previous phase(s) are automatically available. You must define new files, work areas, etc. However, if you do need to use the same file as was defined for a previous phase, you can easily define it again for the current phase with the REUSE statement.

The report for each report is written to a separate DD, which must be present in the execution JCL. The first report (for which there is no NEWPHASE statement) is written to the ZWOUT001 DD. If you then code a NEWPHASE statement to begin the program code of a second phase, its report will be written to ZWOUT002. The next NEW PHASE statement would produce a report written to ZWOUT003, and so on. (Of course, within each phase you are allowed to produce additional reports, specifying your own DD for those.)

PARMS

The parms on this statement are the same parms that are available in the OPTION statement. See a description of those parms beginning on page 106.

EXAMPLE

```
FILE SALES FB(80)
FLD REGION  5  COL(17)
FLD AMOUNT N6.2

FILE TOTALS TYPE(TEMP) OUTPUT
REGION 5
REGTOT N8.2 INIT(0)

WORKAREA
PREVREG 5 INIT(' ')

SORT SALES USING REGION

*************** PROCEDURE ***********************
READ SALES
DOWHILE SALES.#STATUS = 'Y'

  IF SALES.REGION <> PREVREG
      IF PREVREG NOT = SPACES
          MOVE PREVREG TO TOTALS.REGION
          WRITE TOTALS
      ENDIF
      MOVE SALES.REGION TO PREVREG
      MOVE 0 TO REGTOT
  ENDIF

  REGTOT = REGTOT + SALES.AMOUNT
  READ SALES
ENDDO

MOVE PREVREG TO TOTALS.REGION
WRITE TOTALS

NEWPHASE
REUSE TOTALS

READ TOTALS
DOWHILE #STATUS = 'Y'
    PRINT 'TOTAL FOR' REGION '=' REGTOT
    READ TOTALS
ENDDO
```

ONERROR Statement

PURPOSE

Specifies how z/Writer should handle certain error situations if they arise. You may also specify the completion code to use for the job step if an error occurs.

SYNTAX

ONERROR STATEMENT SYNTAX

```
ONERROR
   [ PRTSIZE[(BESTFIT/TRUNCATE/WARN/STOP [n])          ]
   [ INVDATA[(ZERO/WARN/WARNALL/STOP [n] )             ]
   [ OVERFLOW[(ZERO/WARN/WARNALL/STOP [n] )            ]
   [ DIVBYZERO[(ZERO/WARN/WARNALL/STOP [n] )           ]
```

DISCUSSION

You may specify any number of error conditions on a single ONERROR statement. The error handling settings for all conditions *not* specified in the statement is not be affected.

For each error condition, specify one of the handling options allowed for that parm. Optionally, you may also specify a job step completion code to be used if the error does occur. (Otherwise, each error handling option implies its own default completion code setting, as shown in the tables below.)

The completion code affected is the final program completion code that z/Writer passes to z/OS at the end of its execution. This completion code is maintained in the #RETCODE built-in field. Note that an error only *raises* the completion code to the setting that you specify in this statement (or to the default setting.) An error never lowers the completion code if it already contains a higher setting.

This is an executable statement, meaning that it takes effect only after it has been executed. Errors will be handled according to the most recent ONERROR statement executed (for the applicable error condition.) By using multiple statements, you can handle the same error in different ways for different parts of your program.

You may also specify an error condition with empty parentheses after it; this resets the error condition back to its default setting

If this statement is not used, z/Writer handles errors with the default action and completion code as explained below.

PARMS

PRTSIZE[(BESTFIT / TRUNCATE / WARN / STOP [n])]

This condition can arise when z/Writer is preparing numeric data to put in a report line (or in a title, total line, etc.) The PRTSIZE condition is raised when there is not enough room to show all of a numeric field's significant digits (and a negative sign if necessary) in the report column.

By default, z/Writer uses the BESTFIT handling. That means that, rather than showing a misleading number with missing leading digits, an approximate value is shown for the field. For example, if there is not room to show the full value 99.234, just 99 may be shown instead. Or if there is not room to show the full value 123,456, just 123K may be shown instead. If the report column is too small to show even that, then z/Writer indicates this error by putting *S* in the report line. (The S is an error code and indicates a "size" error.)

By default, the PRTSIZE error does not change the completion code of the job step.

The following table shows the settings you can choose for the PRTSIZE condition if you do not want this default handling.

OPTIONS FOR THE PRTSIZE ERROR

OPTION	MEANING	DEFAULT COMP CODE
BESTFIT	z/Writer shows the value of the field to the greatest precision that fits in the report column. It tries in turn each of the following, until it finds one that fits in the column: a) just the whole portion of the value (rounding out any decimal digits; b) the value rounded to thousands (nnnK); the value rounded to millions (nnnM); and so on. If no fit is found, it prints asterisks in the column with an error code of S ("size"). This best fit logic is the default handling for the PRTSIZE error. When this option is chosen, the default is not to change the completion code if the PRTSIZE errors occur. However, you can specify your own completion code after this option, if desired.	**0**
TRUNCATE	z/Writer shows as many digits as will fit, truncating one or more leading digits (and/or possibly a leading negative sign.) This handling, while it can produce very misleading results, does mimic the handling of certain other widely used reporting tools. When this option is chosen, the default is not to change the completion code if the PRTSIZE errors occur. However, you can specify your own completion code after this option, if desired.	**0**

OPTIONS FOR THE PRTSIZE ERROR		
OPTION	**MEANING**	**DEFAULT COMP CODE**
WARN	z/Writer prints a message when a value does not fit within its report column. (Only one message is printed per field that experiences this error condition.) The report will show a size error indicator (**S**) in this column. When this option is chosen, the default is to raise the completion code to 4 if the PRTSIZE errors occur. However, you can specify your own completion code after this option, if desired.	**4**
STOP	A message will be printed when a value does not fit within its report column, and the program will be halted. When this option is chosen, the default is to raise the completion code to 12 if the PRTSIZE errors occur. However, you can specify your own completion code after this option, if desired.	**12**
n	This optional parm (allowed after any of the keyword parms above) specifies the completion code to use if the PRTSIZE errors occur. If z/Writer encounters a PRTSIZE error at any time during the run, it will use this completion code when it exits back to the operating system.	**N/A**

INVDATA[(ZERO / WARN / WARNALL / STOP [n])

This condition arises when the raw data being processed contains an invalid value. (For example, a field defined as packed actually contains hex zeros; or a field defined as NUM contains a non-numeric character.)

By default, z/Writer prints a warning message (once per field) and shows the actual raw data found. (This is printed in the control listing, not in the report.) z/Writer then proceeds as though a valid value of zero had been found for the field. The completion code of the job step is raised to 4.

The following table shows the settings you can choose for this error condition (as well as two others), if you do not want this default handling.

OPTIONS FOR THE INVDATA, OVERFLOW AND DIVBYZERO ERRORS		
OPTION	**MEANING**	**DEFAULT COMP CODE**
ZERO	z/Writer uses a value of zero for the field. No message is printed. When this option is chosen, the default is not to change the completion code if the error occurs. However, you can specify your own completion code after this option, if desired.	**0**

OPTIONS FOR THE INVDATA, OVERFLOW AND DIVBYZERO ERRORS		
OPTION	MEANING	DEFAULT COMP CODE
WARN	z/Writer uses a value of zero for the field. One message is printed per field that experiences this error condition. When this option is chosen, the default is to raise the completion code to 4 if the error occurs. However, you can specify your own completion code after this option, if desired.	4
WARNALL	z/Writer uses a value of zero for the field. A message is printed each time that this error occurs (rather than just once per field.) When this option is chosen, the default is to raise the completion code to 4 if the error occurs. However, you can specify your own completion code after this option, if desired.	4
STOP	A message will be printed when this error occurs, and the program will be halted. When this option is chosen, the default is to raise the completion code to 12 if the error occurs. However, you can specify your own completion code after this option, if desired.	12
n	This optional parm (allowed after any of the keyword parms above) specifies the completion code to use if the specified error occurs. If z/Writer encounters the error at any time during the run, it will use this completion code when it exits back to the operating system.	N/A

OVERFLOW[(ZERO / WARN / WARNALL / STOP [n])

This condition arises when a numeric overflow occurs during processing. Overflows can arise when converting very large numbers, multiplying large numbers, dividing large numbers by very small numbers and in other situations.

By default, z/Writer prints a message and halts the execution of the program. The completion code of the job step is raised to 12.

The table above (under INVDATA) shows the settings you can choose for this condition if you do not want the default handling.

DIVBYZERO[(ZERO / WARN / WARNALL / STOP [n])

This condition arises when a division by zero is attempted during processing.

By default, z/Writer prints a message and halts the execution of the program. The completion code of the job step is raised to 12.

The table above (under INVDATA) shows the settings you can choose for this condition if you do not want the default handling.

EXAMPLES

```
ONERROR PRTSIZE(TRUNCATE) INVDATA(WARN 0) OVERFLOW(ZERO 8)
ONERROR INVDATA()      /* REVERT TO DEFAULT */
ONERROR DIVBYZERO(0)   /* DEFAULT ACTION, BUT DO NOT RAISE COMPLETION CODE */
```

OPEN Statement

PURPOSE

Opens one file or DB2 cursor for subsequent processing. The OPEN statement does not cause a record to be read from (or written to) the file, or a row to be retrieved from a cursor.

SYNTAX

OPEN STATEMENT SYNTAX

```
OPEN    filename/cursorname
```

DISCUSSION

You do not normally need to code an explicit OPEN statement. z/Writer handles file OPENs for you automatically. However, this statement (along with the CLOSE statement) is useful if you wish to read through a file, or DB2 cursor, more than one time in the same phase.

After an OPEN statement, you can check the result of the operation by examining the file's #STATUS built-in field. The table below shows its possible values after the operation.

#STATUS Built-In Field Values After an OPEN Statement

Y - file successfully opened
N - file not opened

PARMS

filename/cursorname

Specifies the name of the file or DB2 cursor to open. The file or cursor must have been previously defined using a FILE or CURSOR statement. This parm is required.

EXAMPLES

```
OPEN EMPL
READ EMPL
DOWHILE #EOF <> 'Y'
```

```
   PRINT EMPL_NUM EMPL_NAME
   READ EMPL
ENDDO
CLOSE EMPL

OPEN EMPL /* READ 2ND PASS */
...
```

OPTION Statement

PURPOSE

Specifies one or more special options for the run as a whole. (That is, options that apply to all phases.)

SYNTAX

OPTION STATEMENT SYNTAX

```
OPTION
       [ CENTURY(nn/50)              ]
       [ DATEDELIM('x','/')          ]
       [ FORMAT(display-format ...)  ]
       [ PICBASE2('x'/'@')           ]
       [ PICBASE10('x'/'#')          ]
       [ VERIFY                      ]
```

DISCUSSION

You may specify as many options as you like on a single OPTION statement. In addition, you may have as many separate OPTION statements as you like.

OPTION statements should normally appear before all other control statements.

PARMS

CENTURY(nn/50)

Specifies the "century cutoff" year. This value is used when assigning a century to date fields that only have 2 bytes for the year (for example, YYMMDD dates). The value defines a 100-year window across the current and the previous century. The default value of 50 sets that window from 1951 to 2050. Date fields whose YY is greater than 50 are assigned to the previous century. All other dates are assigned to the current century. Use this option if you would like to specify a different window to use with your YY date fields.

DATEDELIM('x','/')

Specifies one character to be used when formatting dates in the report (to separate the MM, DD and YYYY parts.) The default is to use a diagonal slash. European users may prefer to use a dash or a dot to format their dates:

```
OPTION DATEDELIM('-')
```

FORMAT(display-format ...)

Specifies one or more display formats to use as the default for all reports in the run. (Use the identical FORMAT parm in a REPORT statement if you just want to set default formats for a *single* report.)

You may specify one display format for each data type. That is: one numeric display format, one date display format and one character display format. The order of the display formats is not important. For example:

```
OPTION FORMAT(DOTSEP DD-MM-YY)
```

The above statement sets two default display format that are very useful for making reports in many countries other than the USA. It specifies that dates should be formatted as DD/MM/YY and that numeric values should be formatted using dots instead of commas (for example, 123.456,78).

A complete list of the display formats available appears in the table on page 117. That table also shows the standard default display formats used when this FORMAT parm is not specified.

Note that this parm only specifies the *default* display format(s) to use. Different display formats can still be specified for individual fields in the report. (Do that with the FORMAT parm of the FIELD statement, or directly in a PRINT or TITLE statement.)

PICBASE2('x'/'@')

Specifies the character to be interpreted in PICTUREs as the Base 2 scaling character. (See a description of this in Appendix C, "Syntax of PICTURE Display Formats" on page 195.) You may specify any 1-byte character or hex literal here. The default character is the "at" sign.

PICBASE10('x'/'#')

Specifies the character to be interpreted in PICTUREs as the Base 10 scaling character. (See a description of this in Appendix C, "Syntax of PICTURE Display Formats" on page 195.) You may specify any 1-byte character or hex literal here. The default character is the "number" or "pound" sign.

VERIFY

Specifies that the control statements should only be verified and any error messages printed. The program will not be executed. This is a global option and applies to all program phases in the run.

EXAMPLES

```
OPTION VERIFY
```

PERFORM Statement

PURPOSE

Unconditionally executes one or more consecutive paragraphs of the program. Afterward, execution resumes with the statement following the PERFORM statement.

SYNTAX

PERFORM STATEMENT SYNTAX

```
PERFORM label [THRU label]
```

DISCUSSION

Within the performed paragraphs, it is permitted to branch outside of the range of those paragraphs. However, this is not usually a good coding practice. If you do branch out of the paragraphs being performed, it is strongly recommended that you branch back into them to complete execution of the performed paragraphs. That allows the processing of the PERFORM statement to wrap up cleanly.

The paragraphs performed must be within the same phase that executes the PERFORM. (Phases are delimited by the NEWPHASE statement.)

Within the code being performed, you are allowed to have additional PERFORM statements. Such nesting of PERFORMs is allowed to any level.

A paragraph of code begins with a label and ends at the last statement before the next label (or the last statement of the program.) (The syntax rules for labels can be found on page 85.)

Empty paragraphs are permitted. (That is, two consecutive labels with no statements between them.) You may want to use an empty paragraph as the “thru” label in a PERFORM.

PARMS

label

Specifies the label of the first (and optionally the last) paragraph to be performed. When only this first label is present, that single paragraph is executed.

[THRU label]

When the THRU parm is also present, it specifies the final paragraph to perform. This paragraph must be located somewhere *after* the first paragraph in your program code. All paragraphs from the first paragraph up to and including the final “thru” paragraph are performed.

EXAMPLE

```
PERFORM SWAP-FIELDS
PERFORM A-WRITE-100 THRU A-WRITE-100-EXIT
...

SWAP-FIELDS:
X=A
A=B
B=X

A-WRITE-100:
WRITE OUT01
TOTW=TOTW+1

A-WRITE-100-EXIT:

B-100-READ:
READ EMPL
...
```

POSITION Statement

PURPOSE

Positions a logical "pointer" in a random access VSAM file so that sequential reads can begin from that point. This statement does not perform an actual read, so the file's record area remains unchanged by it.

SYNTAX

POSITION STATEMENT SYNTAX

```
POSITIONfilename
        KEY(fieldname/'literal')
        [GEN]
        [KEQ/KGE]
```

DISCUSSION

After a POSITION statement, you can check the result of the operation by examining the file's #STATUS built-in field. The table below shows its possible values after a POSITION statement.

#STATUS Built-In Field Values After a POSITION Statement

Y - file positioned successfully (a record meeting the specified key criteria exists in the file, but has not been read yet)
N - file not positioned successfully (no matching record exists in the file -- do not perform a sequential (non-keyed) read until successfully establishing a position within the file.)

A record meets the POSITION criteria by having a key which either:

1) exactly matches the key (or partial key, if GEN specified) in the statement's KEY parm, or

2) is the first key with a value greater than the key (or partial key) in the statement's KEY parm, if KGE is also specified.

After a successful POSITION, the next sequential READ statement returns the first record whose key meets the criteria specified above. (You may then perform additional sequential reads to see if other records exist which also meet your requirements.)

Note: if the POSITION is *not* successful, do not attempt to perform a sequential READ at that point. This can result in a VSAM logical error and the program will end.

Note: other ways to position a keyed VSAM file for sequential reads are:

1) before any I/O operation are performed for a file, the file is positioned at the first record of the file. You can read from the beginning of the file with sequential READ statements.

2) After a (successful) keyed read, the file is left positioned after the record read. You can then read subsequent records with sequential READ statements.

Note: (A sequential READ statement is one that does not have a KEY parm.)

PARMS

filename

Specifies the name of the file to position. The file must have been previously defined using a FILE statement. It must have been defined as either a KSDS or RRDS file. This parm is required.

KEY(fieldname/'literal')

This parm is required. It specifies a field (or literal) with the key value that you want to position the file at. You may specify a key value that is shorter than the VSAM file's defined key length; in that case you must also specify the GEN parm.

Note: no data conversion is performed on the key field. Byte-by-byte comparisons are used to match the KEY parm with the key values stored in the file.

After a successful POSITION statement, the file will be positioned so that the next sequential READ statement will return the first record with a matching key (or partial key when GEN is used). If no such match exists, and the KGE parm was also specified on the POSITION statement, the sequential READ will return the record with the next higher key (or partial key).

[GEN]

Indicates that the KEY parm contains a "generic" key. A generic (or partial) key is shorter than the full key length defined for the file. Use this parm to position a file using only some leading portion of a key value.

[KEQ/KGE]

Specifies the key matching criteria to use with the KEY parm.

- KEQ ("key equal"), the default, indicates that a record is considered a match if it's key (or partial key when GEN is used) value exactly matches the KEY parm.
- KGE ("key greater than or equal to") indicates that a record is considered a match if it's key (or partial key when GEN is used) value is greater than or equal to the KEY parm.

EXAMPLES

```
POSITION DEPT KEY(TEST-DEPT-PREFIX) GEN
IF DEPT.#STATUS = 'Y'
  READ DEPT
  DOWHILE DEPT.#STATUS = 'Y' AND EMPL.DEPT-PREFIX = TEST-DEPT-PREFIX
     PRINT DEPT-NUM DESCRIPTION
     READ DEPT
  ENDDO
ENDIF
```

PRINT Statement

PURPOSE

Prints one line to the report output. The same syntax is also used for the PRINTMODEL statement (page 121).

SYNTAX

PRINT AND PRINTMODEL STATEMENT SYNTAX

```
PRINT/PRINTMODEL
      [ (reportname)                                                    ]
      [ item1[(parms)]  item2[(parms)]  ...                             ]
      [ ADVBEFORE(n/1/PAGE)                                             ]
      [ ADVAFTER(n/0/PAGE)                                              ]
```

Each **item** can be:

- a fieldname, including built-in fields (ex: AMOUNT)
- a built-in function (ex: #MAX(SALES_QTR1, SALES_QTR2))
- a numeric literal (ex: 1000)
- a literal text (ex: 'DONE: _____')

The syntax of the **parms** available for the items is:

```
fieldname/function[(
           [  +n/@n/@fldname          ]
           [  n                       ]
           [  'hdg1|hdg2|...'         ]
           [  display-format          ]
           [  LJ/CJ/RJ                ]
           [  BIZ                     ]
           [  BIR                     ]
           [  BRK/BRKSUM/BRKLAST/NOBRK]  )]

literal[(
           [  +n/@n/@fldname          ]
           [  n                       ]
           [  'hdg1|hdg2|...'         ]
           [  display-format          ]
           [  BRK/BRKSUM/BRKLAST/NOBRK]  )]
```

DISCUSSION

Use the PRINT statement to:

- specify the fields (and any literal texts) to print in the body of your report (or your export file)
- specify where to place each item on the report line
- specify the column headings for each column in the report
- specify the desired display format to use for each item on the report line
- specify whether or not a field should be included in the total line, and optionally specify what value should appear for that field in the total line
- specify how to advance the "carriage" before and/or after the report line is printed

Only the *first* PRINT (or PRINTMODEL) statement in the program code is used for the customization of column headings and total lines. For this reason, you should ensure that all miscellaneous PRINT statements (that print lines only on special occasions, such as during breaks or on errors) are located *after* the PRINT statement for the main report body. The first PRINT statement is called the ***primary* PRINT** statement.

The REPORT and EXPORT statements have options to suppress column headings and/or grand totals from a report, if you do not want them.

Empty PRINT statements are allowed. An empty PRINT statements prints a blank line in your report.

PARMS

(reportname)

This parm is optional. When used, this parm *must* be the first item on the PRINT statement. The report name must be enclosed in parentheses.

When report name is not present, the PRINT statement prints a line to the primary (and often only) report in the current phase.

Use this parm if you are printing to more than one report during a single phase. This parm names the report that you want this PRINT statement to print a line to. Reports are defined and named using the REPORT (or EXPORT) statement (page 128).

fieldname/function/literal[(parms)]

Specifies one item to include in the report line. Fieldnames indicate that the contents of that field should appear in the report line. You may also print the contents of a built-in function. Numeric literals appear as formatted numbers in the report line. Literal texts (in apostrophes or quotes) print "as is" in the report line.

Optionally, you can specify one or more parms in parentheses after the fieldname, function or literal. (There must be no space before the open parenthesis.) The parms let you customize the location and/or appearance of an item in the report line.

Parms Available After a Field Name or Literal

The following table shows the parms that can be specified for each item. All parms are optional and they may be specified in any order, separated by spaces and/or commas. Note that some of these parms are irrelevant for literal items and will be ignored.

PARMS ALLOWED FOR ITEMS IN THE PRINT STATEMENT		
SYNTAX	**DESCRIPTION**	**EXAMPLE**
Parms Affecting an Item's Location in the Print Line		
These parms specify where to begin the item's data in the report line. By default, z/Writer begins the first item in column 1, and each subsequent item one space past the end of the previous item. Only one of these parms may be used for an item.		
+n	Means begin this column "n" bytes after the end of the previous column. For example, you can specify +0 if *no spaces* are wanted after the previous column. **Note:** the "+" is required here in order to distinguish this parm from the width parm ("n"). z/Writer's default is to leave 1 space between report columns. (You can change this default with the COLSPACE parm in the REPORT statement.)	`PRINT 'NAME=' NAME(+0)`
@n	Means begin the data in column "n" of the report line. Columns are numbered beginning with 1.	`PRINT NAME(@125)`
@fieldname	Means that the report column should begin in the same location as the named field began (in the first PRINT statement for the same report.) This parm can help you align columns in various report lines.	`PRINT TOT-AMOUNT(@AMOUNT)`
Parms Affecting an Item's Appearance in the Report		
n	Specifies an override width (in bytes) to use for this report column. When omitted, z/Writer picks a default width to use. This parm is useful for shortening columns that would otherwise use up too much space in the report line. Note that this parm is ignored if a date type display format is used to display the item.	`PRINT DESCRIPTION(10)`

PARMS ALLOWED FOR ITEMS IN THE PRINT STATEMENT		
SYNTAX	DESCRIPTION	EXAMPLE
'hdg1|hdg2|...'	Specifies the column heading to use for this field. You can use one or more "|" characters within the text to indicate how the heading text should be broken onto multiple column heading lines. z/Writer does not limit the number of column heading lines you may specify. When omitted, z/Writer uses the column heading text from the HEADING parm of the FIELD statement, if any. Otherwise the fieldname itself (broken apart at each dash and underscore) is used as the column heading. **Note:** column headings are discussed on page 14.	`PRINT SEX('S|E|X')`
LJ/CJ/RJ	Specifies how the data should be justified *within* the area reserved for that column in the report line. The values mean left-justified, center-justified and right-justified, respectively. **Note:** that this parm determines how data is justified within the item's actual *data area* in the report. If the column heading for the item is larger than the data area, your data may not appear to be justified under the column headings. In such cases, try embedding leading or trailing blanks within your column heading texts to achieve the result you want.	`PRINT DEPT-NUM(LJ)`

display-format	You may specify the name of any of z/Writer's "display formats" that are valid for the data type of the item. The display format determines how the data will be formatted in the report line. (The table on page 117 lists all of z/Writer's display formats.)
	
	When omitted, z/Writer uses: 1) the display format from the FORMAT parm of the FIELD statement, if any, or 2) a default display format.
		`PRINT AMOUNT(CURRENCY)`
		`PRINT SSN(PIC'999-99-9999')`
		`PRINT TEL(PIC'(999) 999-9999')`

BIZ	("Blank-if-zero"). Use this with a numeric field if you want the report column to be left blank for zero values. In reports where most instances of a field are zero, this parm can help make significant data stand our better. It can also reduce clutter in a busy report.	`PRINT ERROR-COUNT(BIZ)`
BIR	("Blank-if-repeat"). Specifies that this report column should be left blank if its contents would be a repeat of its contents in the previous instance of this report line.	`PRINT REGION(BIR)`

PARMS ALLOWED FOR ITEMS IN THE PRINT STATEMENT		
SYNTAX	**DESCRIPTION**	**EXAMPLE**
Parms Affecting the Data Shown in Total LInes *Meaningful only when used in the first* PRINT *statement of the program. Ignored otherwise. This parm does not affect the value printed when the* PRINT *statement is executed in the program. It only affects the total lines that* z/Writer *prints for you at control breaks and at the end of the report. Only one of these parms may be used for an item.*		
BRK	This parm indicates that this field should appear in total lines. If the field is a quantitative numeric field, its total value (SUM) will print in the total line. For other types of fields, the value of the field from the last record in the control group (LAST) will print in the total line. **Note:** this parm alone does not create a control break for a field. Use the BREAK statement to do that.	`PRINT EMPL_NUM(BRK)`
NOBRK	This parm indicates that this field should *not* appear in total lines. Normally, all quantitative numeric fields, and all control break fields that were constant for the control group appear in the total line. Use this parm to keep such a field out of the total line.	`PRINT EMPL_NUM(NOBRK)`

BRKSUM	This parm indicates that the total value (SUM) of this field for the relevant control group should print in total lines. That is, the sum of the values of this field for all records included in the control group just ended. Naturally, this parm is only allowed for numeric fields. (But the numeric field does not have to be quantitative.)	`READ SALES`
		`PRINT EMPL_NUM(BRKLAST)`
		`REGION(BRKLAST)`
		`EMPL_COUNT(BRKSUM)`

BRKLAST	This parm indicates that the value of this field from the last record in the control group (LAST) should print in total lines.	`READ SALES`
		`PRINT EMPL_NUM(BRKLAST)`
		`REGION(BRKLAST)`
		`EMPL_COUNT(BRKSUM)`

Display Formats

The following table shows the complete list of display formats. These can be used in several statements, not just the PRINT statement. For example, display formats can be used in FIELD, OPTION, REPORT and TITLE statements.

DISPLAY FORMATS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
Display Formats That Can Be Used With Any Type Of Data		
ASIS	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 (unless changed with a FORMAT parm in an OPTION or REPORT statement.)	ABC
QCHAR	The data is enclosed within quotation marks. Otherwise, the data is not reformatted at all. This format is useful for formatting character fields in comma-delimited files. **Note:** you can use the QCHAR parm of the REPORT statement to choose a character other than the double quotation mark to use 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.) And for displaying status or flag fields containing non-printable values.	C1C2C3
BITS	Each byte of data is expanded into an 8-byte character string of 0's and 1's, showing the individual bits within the data. This format is useful for displaying the individual bits within status or flag fields.	11000001
Display Formats That Can Be Used Only With Numeric Data		
EDIT	The numeric value is edited according to common American presentation conventions. Formatting includes suppression of leading zeros, the use of commas as thousands separators and a decimal point before any decimal digits. A floating negative sign precedes negative numbers. This is the **default display format** for all numeric fields (unless changed with a FORMAT parm in an OPTION or REPORT statement.)	1,234.56 –1,234.56
DOTSEP	The numeric value is edited according to presentation conventions common in Europe and other international areas. Formatting includes suppression of leading zeros, the use of "dots" as thousands separators and a comma before any decimal digits. A floating negative sign precedes negative numbers.	1.234,56 –1.234,56

DISPLAY FORMATS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
NOCOMMA	Same as EDIT, except that commas are not inserted among the digits. This format is useful for formatting numeric fields in comma-delimited files.	1234.56 –1234.56
CURRENCY DOLLAR	Same as EDIT, but a floating currency sign will precede the first significant digit. The currency sign is a dollar sign by default. (Use the CURRCHAR parm of the REPORT statement to choose a character other than the dollar sign to use with this display format.)	$1,234.56 –$1,234.56
DISP DISPLAY	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
PIC PICTURE	A "picture" is used to describe exactly how the numeric value should be formatted. This is useful for formatting special purpose numbers, such as telephone numbers, social security numbers, numbers of Kb, Mb, Gb and etc. The details of z/Writer's PICTURE syntax are explained in Appendix C, "Syntax of PICTURE Display Formats" on page 195.	(800) 555–1212 123–45–6789 1.29MB
Display Formats That Can Be Used Only With Fields Containing a Date ***(See page 73)***		
MM-DD-YY	MM/DD/YY This is the **default display format** for all fields defined as containing a date.	2/1/98 12/31/00
MM-DD-YYYY	MM/DD/YYYY	2/1/1998 12/31/2000
DD-MM-YY	DD/MM/YY	1/2/98 31/12/00
DD-MM-YYYY	DD/MM/YYYY	1/2/1998 31/12/2000
YY-MM-DD	YY/MM/DD	98/1/2 00/12/31
YYYY-MM-DD	YYYY/MM/DD	1998/1/2 2000/12/31
SHORTEURO	DD MMM YY	31 DEC 10

DISPLAY FORMATS		
DISPLAY FORMAT	**DESCRIPTION**	**EXAMPLE**
SHORTEURO4	DD MMM YYYY	31 DEC 2010
SHORTUSA	MMM DD, YYYY	DEC 31, 2010
LONGEURO	DD MMMMMMMMM YY	31 DECECMBER 10
LONGEURO4	DD MMMMMMMMM YYYY	31 DECECMBER 2010
LONGUSA	MMMMMMMMM , DD YYYY	DECECMBER 31, 2010

ADVBEFORE(n/1/PAGE)

This "advance before" parm specifies that special spacing is desired *before* printing the report line. When this parm is omitted, the report line is printed after advancing a single "line feed." (This results in a standard, single spaced report.)

A numeric literal "n" specifies the number of line feeds to advance before printing the report line. For example, a value of 2 would produce a double-spaced report. A value of 0 can be used to overprint the previous report line, if the printer hardware supports this.

Specifying PAGE means to advance to a new page before printing the report line.

Note: this parm is ignored in the PRINTMODEL statement. It is also ignored when used for an export file.

ADVAFTER(n/0/PAGE)

This "advance after" parm specifies that special spacing is desired *after* printing the report line. When this parm is omitted, z/Writer does not advance the printer immediately after printing the report line.

A numeric literal "n" specifies the number of line feeds to advance after printing the report line. For example, a value of 1 would result in a blank line following the report line.

Specifying PAGE means advance to a new page after printing the report line.

Note: if possible, we recommend using only the ADVBEFORE parm to obtain any special spacing that your report may require.

Note: this parm is ignored in the PRINTMODEL statement. It is also ignored when used for an export file.

EXAMPLES

```
PRINT EMPL-NUM AMOUNT('SALES|AMOUNT' DOTSEP)
PRINT SSN(PIC'999-99-9999')
PRINT SMF73FG1(HEX) ADVBEFORE(PAGE)
PRINT AMOUNT(NOCOMMAS) ',' LAST_NAME(QCHAR) ',' FIRST_NAME(QCHAR) /* COMMA DELIMITED */
PRINT FIRST_NAME('FIRST|INITIAL', +2, 1)
PRINT (ERRORS) 'BAD DATE FOUND FOR' EMPL-NUM
```

```
PRINT 'SALES' 'SALES'
PRINT 'DATE ' 'TIME '
PRINT '_____' '_____' ADVBEFORE(0) /* UNDERLINE HEADINGS */


FILE EMPL TYPE(KSDS)
...

REPORT LINES(55)
REPORT (PARTTIME) LINES(55)

PRINT EMPL_NAME HIRE_DATE STATUS

IF STATUS = 'P'
   PRINT (PARTTIME) EMPL_NAME HIRE_DATE STATUS
ENDIF

TITLE 'LIST OF ALL EMPLOYEES'
TITLE (PARTTIME) 'LIST OF PART TIME EMPLOYEES'
```

PRINTMODEL Statement

PURPOSE

This statement is identical to the PRINT statement, except for one thing: it is a declaratory (rather than executable) statement and does not cause a line to be printed in the report. It is useful for formatting summary reports which do not show the detail data from the input records.

SYNTAX

The syntax of this statement is identical to the syntax of the PRINT statement shown on page 112. All of the PRINT statement keywords and parms are allowed in the PRINTMODEL statement. However, some of the parms (for example, ADVBEFORE) are not meaningful and are ignored.

DISCUSSION

When making a summary report, use this statement the way you would use a PRINT statement for a regular detail report. The fields listed on the PRINTMODEL statement will determine:

- the column headings to use for the report
- the fields that will appear in the total lines (at control breaks and at the end of the report)
- the layout (order and spacing) of the total line columns

Remember, in order to have any effect, this declaratory statement must *precede* any PRINT statements. z/Writer uses the first PRINT or PRINTMODEL statement of the program to determine column headings and total line layout.

READ Statement

PURPOSE

Reads one record from an INPUT or UPDATE file.

SYNTAX

READ STATEMENT SYNTAX

```
READ    filename
        [KEY(fieldname/'literal')
        [GEN]
        [KEQ/KGE]
```

DISCUSSION

Depending on the file type and the parms present on this statement, either a sequential or a direct ("keyed") read will be performed.

Sequential Reads

If the READ statement does not have a KEY parm, a sequential read is performed. This reads the next record at the file's current position. (That will be the record after the previous record read, or the first record in the file if there were no previous read attempts.) Sequential reads are allowed for all file types.

> **Note:** a program error will occur if you attempt to perform a sequential read on a keyed VSAM file when the file is not "positioned" at any record. This can happen after an unsuccessful READ or POSITION statement.

A file is correctly "positioned" in these cases:

1) when the program starts execution (it is positioned before the first record in the file),

2) after a successful keyed READ or POSITION statement, and

3) after a successful sequential read.

Direct Reads

If the READ statement does have a KEY parm, a direct read is attempted. Direct reads are allowed only to VSAM files defined as KSDS or RRDS. (The key for a RRDS file must be a 4-byte binary field.) You may specify either a full key or a partial ("generic") key.

> **Note:** after a successful keyed READ, you can follow it with one or more *sequential* READs to read successive records after that file record.

Record Length

After a successful read, the file's record area contains the new record. The length of that record is available in the file's #LENGTH built-in field. Here are more details about an INPUT file's #LENGTH built-in field.

For **fixed-length** files, z/Writer initializes #LENGTH one time to the file's fixed record length (from the ACB at open time.). You can examine this field, but you should not change it.

For **variable-length** VSAM files, z/Writer initializes the #LENGTH built-in field to the *maximum* defined record length from the ACB at open time. Then, after each successful READ, the length of the newly read record is put in #LENGTH for you.

RRN

For RRDS files, the RRN of the record read is available in the file's #RRN built-in field.

Status of a Read Operation

After a READ statement, you can check the result of the operation by examining the file's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the filename, if the phase has definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a READ.

#STATUS Built-In Field Values After a READ Statement

Y - file successfully read
N - no record read. For sequential reads, this normally indicates EOF; for keyed reads, this normally indicates that no record matching the key (or partial key) was found.

You can also test for end of file after a READ using the file's #EOF built-in field. The table below shows the possible values of #EOF after a READ.

#EOF Built-In Field Values After a READ Statement

Y - file has reached the end-of-file
N - file has not reached end-of-file

PARMS

filename

Specifies the name of the file to read. The file must have been previously defined using a FILE statement. It must have been defined as either an INPUT or an UPDATE file.

[KEY(fieldname/'literal')]

Specifies the field (or literal) containing the key value of the record you want to read. You may specify a key value that is shorter than the VSAM file's defined key length; in that case you must also specify the GEN parm.

Note: no data conversion is performed on the key field. Byte-by-byte comparisons are used to match the KEY parm with the key values stored in the file.

[GEN]

Indicates that the KEY parm contains a "generic" key. A generic key is shorter than the full key length defined for the file. Use this parm to read a record when you only want to match some leading portion of the key.

[KEQ/KGE]

Specifies the key matching criteria desired:

- KEQ ("key equal"), the default, indicates that a record is considered a match if it's key value (or partial key, if GEN is specified) exactly matches the KEY parm.
- KGE ("key greater than or equal to") indicates that a record is considered a match if it's key value (or partial key, if GEN is specified) is greater than or equal to the KEY parm.

EXAMPLES

```
FILE SALES TYPE(ESDS)
FILE EMPL  TYPE(KSDS)
...
READ SALES
READ EMPL KEY('040')
READ EMPL KEY(SALES.EMPL-NUM)
READ EMPL KEY(SHORT-KEY) GEN KGE
READ EMPL KEY('2') GEN
READ EMPL
```

REDEFINE Statement

PURPOSE

The REDEFINE statement allows you to "back up" and redefine an earlier field. The REDEFINE statement may be used only within field definitions. That is, it may appear only among the FIELD statements that immediately follow FILE, TABLE and WORKAREA statements.

SYNTAX

REDEFINE STATEMENT SYNTAX

```
REDEFINEfieldname

Alternate Spellings:
REDEFINE - REDEF
```

DISCUSSION

After a REDEFINE statement, the next field defined will begin in the same byte of the record as the field named on the REDEFINE statement. After that next FIELD statement, you may continue defining additional fields there, each beginning immediately after the previous field. Or you can skip directly to the first byte *after* the original field being redefined by coding an ENDREDEFINE statement.

PARMS

fieldname

Specifies the name of a previously defined field in the same record area (or work area). FIELD statements after this REDEFINE will begin in the starting column of the named field. This parm is required.

EXAMPLE

```
FLD SALES-DATE       6
REDEFINE SALES-DATE
FLD SALES-DATE-YY    2
FLD SALES-DATE-MM    2
FLD SALES-DATE-DD    2 /* COMPLETE REDEFINE. ENDREDEF IS OPTIONAL */

FLD CUSTOMER         15

FLD TELEPHONE        N10
REDEFINE TELEPHONE
FLD AREA-CODE        N3
ENDREDEF                  /* PARTIAL REDEFINE. ENDREDEF IS NEEDED */

FLD TAX              N4.2
```

RELEASE Statement

PURPOSE

Releases (unlocks) a VSAM record that has been read for update, without actually updating (or deleting) it. (A record must have been successfully read from an UPDATE file before this statement is executed.)

SYNTAX

RELEASE STATEMENT SYNTAX

```
RELEASE filename
```

DISCUSSION

You are never required to use this statement. However, for performance reasons, you might wish to use it. The RELEASE statement releases a record that you have read from an UPDATE file, once you know that you will not need to delete or rewrite that record. That releases VSAM's "lock" on the record so that other users can access it. The record's data remains in the record area for your program to continue to use.

When you specify UPDATE on a FILE statement, the records read from that file are available for update processing. Each time a record is read from an update file, VSAM puts a lock on the record until you take one of the following actions:

- you update that record by executing a REWRITE statement for the same file (normally after changing the contents and/or length of the record)
- you delete that record by executing a DELREC statement for the same file
- you explicitly release the record by executing a RELEASE statement for the same file
- you READ a new record from the same file, which releases the lock on the current record (and puts a lock on the newly read record)

Status of a Release Operation

After the RELEASE statement, you can check the result of the operation by examining the file's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the filename, if the phase has

definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a RELEASE.

#STATUS Built-In Field Values After a RELEASE Statement

Y - record successfully released
N - record not released.

PARMS

filename

Specifies the name of the file whose outstanding record should be released. The file must have been previously defined (in a FILE statement) as an UPDATE file. Also, a record must have been successfully read from that file, and not yet rewritten or deleted.

EXAMPLE

```
FILE EMPL UPDATE TYPE(KSDS)
...

READ EMPL         /* READ FIRST RECORD FOR UPDATE */

DOWHILE EMPL.#STATUS = 'Y'
  IF EMPL.EMPL-NUM = '444'
    DELREC EMPL    /* DELETE RECORD FROM FILE */
  ELSEIF EMPL.EMPL-NUM = '555'
    MOVE 'I' TO EMPL.STATUS   /* MARK REC 555 AS INACTIVE */
    MOVE 40 TO EMPL.#LENGTH   /* SHORTEN INACTIVE RECORD */
    REWRITE EMPL              /* REWRITE SHORTER 555 RECORD */
  ELSE
    RELEASE EMPL    /* LEAVE RECORD UNCHANGED ON FILE. */
  ENDIF
  ... /* OTHER PROCESSING */
  READ EMPL /* READ NEXT RECORD FOR UPDATE */
ENDDO
```

REPORT Statement

PURPOSE

This declarative statement overrides one or more default options for the primary report. Or, this statement can specify the name of a *new report* to be written in the current program phase (along with any options to use).

SYNTAX

REPORT STATEMENT SYNTAX

```
REPORT
       [ (reportname)                          ]
       [ COLSEP('xxx'/'_')                     ]
       [ COLSPACE(N/1)                         ]
       [ CURRCHAR('x'/'$')                     ]
       [ FORMAT(display-format ...)            ]
       [ LINES(NN/60)                          ]
       [ NOCC                                  ]
       [ NOCOLHDGS                             ]
       [ NODATE                                ]
       [ NOGRANDTOTALS                         ]
       [ NOPAGE                                ]
       [ NOUNDER                               ]
       [ QUOTECHAR('x'/'"')                    ]

Abbreviations Allowed:
NOCOLHDGS - NOCOLHDG
NOGRANDTOTALS - NOGRANDTOTAL, NOGRAND
```

DISCUSSION

Only a single REPORT statement is allowed per report, but it may contain as many options as you like.

The report name parm, if used, must be the first parm on the REPORT statement. The report name must be enclosed in parentheses. It specifies the name of a new report to be written in the current phase. (Program phases are discussed on page 97.) After the report name, the other options may be specified in any order.

Primary Report

When the REPORT statement does not begin with a report name (in parentheses), the statement applies to the *primary report*. The primary report is the one that all PRINT statements without a report name write to. (Reports for the unnamed, primary report go to the ZWOUT001 DD for the first phase, ZWOUT002 for the primary report in the second phase, and so on.)

A REPORT statement is not required for the primary report. But you can use one to override the defaults options for the primary report.

Additional Reports

When the REPORT statement begins with a report name (in parentheses), the statement defines a *new report* for the current phase. You may have as many reports (and/or export files) in a single phase as you like.

To write to the new report, use the same report name parm (again, in parentheses) as the first item in a PRINT statement. Also use this report name as the first parm in any TITLE statements for the report.

The DDNAME used for the new report will be the report name itself.

Note that automatic control break processing (using the BREAK statement) is only available for the primary report in each phase.

REPORT Statement Location

As a declaratory (rather than executable) statement, the *exact* location of your REPORT statement is not critical. However, when used, it is *mandatory* that it appear before the first PRINT or TITLE statement for that report.

Record Length of Report

You can choose the record length to use for your report file in either of two ways:

- specify DCB LRECL information in the output DD in your JCL
- write the report to an existing fixed length dataset

In either case, z/Writer will format your report lines to that maximum length. If no DCB LRECL or dataset label information is specified, z/Writer picks a default length of 133.

PARMS

(reportname)

This parm is not allowed for the primary REPORT statement in a phase. When present, this must be the first item on the REPORT statement.

When a report name is not present, the REPORT statement specifies options for the first (and often only) report in the current phase. All PRINT and TITLE statements that do not have a report name parm apply to this unnamed REPORT statement.

Use this parm if you are printing to more than one report output during a single phase. The reports after the first one must have a name. You use this parm to specify the name you want to use for the report. (The naming rules are like those for file names (page 78). However, the report name must also be valid for use as a DDNAME.) The name may not be the name of a file, table or workarea in the program. Use this same report name (again in parentheses) later in the PRINT and TITLE statements for this report.

When using this parm, you must also supply a DD with this same report name in your execution JCL. The report will be written to that output DD.

COLSEP('xxx'/'_')

Specifies a separator text that should appear between each column of the report. When not specified, the default is to put one space between report columns. If the COLSPACE parm is also used in the statement, its value must be consistent with the size of the separator text.

COLSPACE(n/1)

Specifies the default number of spaces to leave between each column in the report. When not specified, the default is to put one space between report columns. If the COLSEP parm is also used in the statement, this value must be consistent with the size of the separator text.

CURRCHAR('x'/'$')

Specifies the character to use as the currency symbol. Items formatted with the CURRENCY display format will use this character. (Display formats are listed on page 117.) You may specify any 1-byte

FORMAT(display-format ...)

Specifies one or more default display formats to use in this report. (Use the identical FORMAT parm in the OPTIONS statement if you want to set default formats for *all* reports in the run.)

You may specify one display format for each data type. That is: one numeric display format, one date display format and one character display format. The order of the display formats is not important. For example:

```
REPORT FORMAT(DOTSEP DD-MM-YY)
```

The above statement sets two default display format that are very useful for making reports in many countries other than the USA. It specifies that dates should be formatted as DD/MM/YY and that numeric values should be formatted using dots instead of commas (for example, 123.456,78).

A complete list of the display formats available appears in the table on page 117. That table also shows the standard default display formats used when this FORMAT parm is not specified.

Note that this parm only specifies the *default* display format(s) to use. Different display formats can still be specified for individual fields or columns in the report. (Do that with the FORMAT parm of the FIELD statement, or directly in a PRINT or TITLE statement.)

LINES(nn/60)

Specifies the number of lines to print on each page of the report.

NOCC

Normally z/Writer puts a "carriage control" character (' ', '0', '-' or '1') at the beginning of each line printed to a report. (This character is actually *before* "column 1" of the report line.) Use this option to suppress the carriage control character from your report lines. For example, you might want to do this when writing your report output to a file for further processing.

NOCOLHDGS/NOCOLHDG

Suppresses column headings in the report. By default, reports get columns headings based on the fields in the first PRINT statement (for that report) found in the program. (See page 14 for a discussion of column headings.)

NODATE

Normally z/Writer inserts the current date at column 1 of the first title line. Use this option to suppress this date.

NOGRANDTOTALS/NOGRANDTOTAL/NOGRAND

Suppresses grand totals from printing at the end of an auto-cycle report.

NOPAGE

Normally z/Writer inserts the page number in the right-hand columns of the first title line. Use this option to suppress this page number.

NOUNDER

Normally z/Writer underscores the report's column headings. Use this option to suppress the underscores and print only the column heading texts.

QUOTECHAR('x'/'"')

Specifies the character to use as the quote character. Items formatted with the QCHAR display format will be surrounded with this character. (Display formats are listed on page 117.) You may specify any 1-byte character or hex literal here. The default quote character is the double quotation mark.

EXAMPLES

```
FILE EMPL TYPE(KSDS)
...

REPORT LINES(55)
REPORT (PARTTIME) LINES(55)

PRINT EMPL_NAME HIRE_DATE STATUS

IF STATUS = 'P'
   PRINT (PARTTIME) EMPL_NAME HIRE_DATE STATUS
ENDIF

TITLE 'LIST OF ALL EMPLOYEES'
TITLE (PARTTIME) 'LIST OF PART TIME EMPLOYEES'
```

RETRIEVE Statement

PURPOSE

Retrieves one record from a table. When a record is "retrieved" it is copied from the table's internal storage area into the record area that was defined for it (with FIELD statements immediately following the TABLE statement.)

SYNTAX

RETRIEVE STATEMENT SYNTAX (FOR NON-KEYED TABLES)

```
RETRIEVE tablename
       [ NEXT/FIRST/ENTRY(numeric-literal/field) ]
```

RETRIEVE STATEMENT SYNTAX (FOR KEYED TABLES)

```
RETRIEVE tablename
       [ NEXT/FIRST/KEY(fieldname/'literal')     ]
```

DISCUSSION

z/Writer supports two types of tables.

- **Sequential tables.** Tables defined without the KEY parm are sequential tables. Records are maintained in the order in which they are "stored".
- **Keyed tables.** Tables defined with the KEY parm are keyed tables. z/Writer maintains keyed tables in key order as balanced binary trees.

The type of table and the keyword parm used on this statement determine which record will be retrieved from the table.

Status of a Retrieve Operation

After the RETRIEVE statement, you can check the result of the operation by examining the table's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the tablename, if the

phase has definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a RETRIEVE.

#STATUS Built-In Field Values After a RETRIEVE Statement

Y - record successfully retrieved. The table's record area contains the new record.
N - no record retrieved. For sequential retrievals, this indicates end-of-table. For retrievals by key or entry number, this indicates that no matching record was found (or that there was an error in the key/entry value). The table's record area remains unchanged.

PARMS

tablename

Specifies the name of the table to retrieve a record from. The table must have been previously defined using a TABLE statement.

NEXT / FIRST/ENTRY(numeric-literal / field)
NEXT / FIRST/KEY(fieldname / 'literal')

Specifies which record to retrieve from the table. If this parm is omitted, the default is to retrieve the "next" record from the table. Note that the KEY parm may only be specified for tables whose definition included the KEY parm. The ENTRY parm is only allowed for sequential tables.

The following table explains the meaning of each value for this parm.

TABLE RETRIEVAL OPTIONS	
OPTION	**MEANING**
NEXT	*Allowed for any table.* Retrieves the next record from the table. That is the record after the previous record retrieved from the table. (Or the first record if there have been no prior retrievals.) For keyed tables, the "next record" is the record with the next higher key value. For sequential tables, the "next record" is the record that was added after the previous record retrieved.
FIRST	*Allowed for any table.* Retrieves the first record from the table. For keyed tables, this is the record with the lowest key value. For sequential tables, this is the first record that was added to the table.

TABLE RETRIEVAL OPTIONS	
OPTION	**MEANING**
KEY(fieldname/'literal')	*Allowed only for keyed files (those defined with the KEY parm in the TABLE statement.)* Retrieves the record whose key field matches the KEY parm's value. If no matching record is found, the contents of the table's record area is not changed, and the table's #STATUS field is set to 'N'. **Note:** no data conversion is performed on the key field. Byte-by-byte comparisons are used to match the KEY parm with the key values stored in the table. The literal or field specified here must be the same length as the key length defined for the table.
ENTRY(numeric-literal/field)	*Allowed only for sequential (non-keyed) files (those defined without the KEY parm in the TABLE statement.)* Retrieves a record according to the order in which it was added to the table. Thus, ENTRY(1) would return the first record that was added to the table; ENTRY(2) would return the second record that was added to the table, and so on. And ENTRY(tablename.#COUNT) would return the last record added to the table. The value specified here can be either a numeric literal, or the name of a numeric field. If no record is found for the entry specified, the contents of the table's record area is not changed and the table's #STATUS field is set to 'N'. #STATUS is also set to 'N' if the entry value is a field whose value is not numeric at execution time.

EXAMPLES

```
RETRIEVE MY-SEQ-TABLE FIRST
RETRIEVE MY-SEQ-TABLE
RETRIEVE MY-SEQ-TABLE ENTRY(PART-NUM)
RETRIEVE MY-KEY-TABLE FIRST
RETRIEVE MY-KEY-TABLE
RETRIEVE MY-KEY-TABLE KEY('123')
RETRIEVE MY-KEY-TABLE KEY(WORKKEY)
```

REUSE Statement

PURPOSE

Specifies that a file or table defined in an earlier report phase also be defined for the current phase.

SYNTAX

REUSE STATEMENT SYNTAX

```
REUSE   filename
        [ INPUT/OUTPUT/UPDATE                                          ]
        [ PRESORT(fieldname[(ASC/DESC)] ... [#EQUALS])                 ]

REUSE   tablename
```

DISCUSSION

By specifying the REUSE statement, you can avoid retyping the FILE and FIELD statements for a file all over again in the current phase.

Most aspects of the file definition are copied from the original FILE statement for a reused file. However, the access type always defaults to INPUT for the reused file. (You can override this on the REUSE statement, if desired.) And any PRESORT parm specified in the original FILE statement is ignored. (You can, however, specify a new PRESORT parm on the REUSE statement.)

The REUSE statement is often used in conjunction with TEMP type files. Values are computed and written to the TEMP file in one phase. A later phase then "reuses" the same file as an input file, reading the records back in.

> **Note:** z/Writer closes the reused file at the end of the earlier phase(s) where it is used. When used in the current report, it will be opened anew. That means that if you use a file as an OUTPUT file in multiple phases, any records written by the earlier phase will be overwritten when the file is reopened in a later phase.

PARMS

filename

Specifies the name of a file defined in an earlier phase.

INPUT/OUTPUT/UPDATE

Specifies the type of process that will be performed on the file in this phase. Refer to the description under the FILE statement (page 81).)

PRESORT(fieldname[(ASC/DESC)] ... [#EQUALS])

Specifies that z/Writer should presort the whole file before beginning to execute the user program. Refer to the description under the FILE statement (page 84).)

tablename

Specifies the name of a table defined in an earlier phase.

EXAMPLE

```
FILE SALES FB(80)
FLD REGION  5  COL(17)
FLD AMOUNT N6.2

FILE TOTALS TYPE(TEMP) OUTPUT
REGION 5
REGTOT N8.2 INIT(0)

WORKAREA
PREVREG 5 INIT(' ')

SORT SALES USING REGION

*************** PROCEDURE ************************
READ SALES
DOWHILE SALES.#STATUS = 'Y'

  IF SALES.REGION <> PREVREG
      IF PREVREG NOT = SPACES
          MOVE PREVREG TO TOTALS.REGION
          WRITE TOTALS
      ENDIF
      MOVE SALES.REGION TO PREVREG
      MOVE 0 TO REGTOT
  ENDIF

  REGTOT = REGTOT + SALES.AMOUNT
  READ SALES
ENDDO

MOVE PREVREG TO TOTALS.REGION
WRITE TOTALS

NEWPHASE
REUSE TOTALS

READ TOTALS
DOWHILE #STATUS = 'Y'
    PRINT 'TOTAL FOR' REGION '=' REGTOT
    READ TOTALS
ENDDO
```

REWRITE Statement

PURPOSE

Rewrites a VSAM record that has been read for update, replacing the record in the VSAM file with the current contents of the file's record area. (A record must have been successfully read from an UPDATE file before this statement is executed.)

SYNTAX

REWRITE STATEMENT SYNTAX

```
REWRITE filename
```

DISCUSSION

When you specify UPDATE on a FILE statement, the records read from that file are available for update processing. Each time a record is read from an update file, VSAM puts a lock on the record until you take one of the following actions:

- you update that record by executing a REWRITE statement for the same file (normally after changing the contents and/or length of the record)
- you delete that record by executing a DELREC statement for the same file
- you explicitly release the record by executing a RELEASE statement for the same file
- you READ a new record from the same file, which releases the lock on the current record (and puts a lock on the newly read record)

Record Lengths

Here is how an UPDATE file's #LENGTH built-in field is used.

For **fixed-length** files, z/Writer initializes #LENGTH one time to the file's fixed record length (from the ACB at open time.). You can examine this field, but you should not change it.

For **variable-length** VSAM files, z/Writer initializes the #LENGTH built-in field to the *maximum* defined record length from the ACB at open time. Then, after each successful READ, the length of the newly read record is put in #LENGTH for you. If you then rewrite the record, you must move the correct length to #LENGTH (if the new record has a different length.)

Status of a Rewrite Operation

After the REWRITE statement, you can check the result of the operation by examining the file's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the filename, if the phase has

definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a REWRITE.

#STATUS Built-In Field Values After a REWRITE Statement
Y - record successfully rewritten **N** - record not rewritten.

PARMS

filename

Specifies the name of the file whose current record should be rewritten. The file must have been previously defined (in a FILE statement) as an UPDATE file. Also, a record must have been successfully read from that file, and not yet deleted or released.

EXAMPLE

```
FILE EMPL UPDATE TYPE(KSDS)
...

READ EMPL         /* READ FIRST RECORD FOR UPDATE */

DOWHILE EMPL.#STATUS = 'Y'
  IF EMPL.EMPL-NUM = '444'
    DELREC EMPL    /* DELETE RECORD FROM FILE */
  ELSEIF EMPL.EMPL-NUM = '555'
    MOVE 'I' TO EMPL.STATUS   /* MARK REC 555 AS INACTIVE */
    MOVE 40 TO EMPL.#LENGTH   /* SHORTEN INACTIVE RECORD */
    REWRITE EMPL              /* REWRITE SHORTER 555 RECORD */
  ELSE
    RELEASE EMPL    /* LEAVE RECORD UNCHANGED ON FILE. */
  ENDIF
  ... /* OTHER PROCESSING */
  READ EMPL /* READ NEXT RECORD FOR UPDATE */
ENDDO
```

SHOW Statement

PURPOSE

Prints a literal text or the contents of a field in the *control listing* (not in the report). This statement (often used along with the TRACEON statement) can be very useful when debugging new programs.

SYNTAX

SHOW STATEMENT SYNTAX

```
SHOW fieldname/'literal'
```

DISCUSSION

Only a single field or literal may be specified on the SHOW statement.

When a fieldname is specified, both the name of that field and its formatted contents are printed in the control listing (SYSPRINT).

When a literal text is specified, just that text itself prints in the control listing.

Note: use the SHOWHEX statement if you need to see the contents of the field in hex format.

PARMS

fieldname/'literal'
Specifies a field or literal to show in the control listing.

EXAMPLE

```
SHOW 'AT HEADING ROUTINE'
SHOW LAST-NAME
```

SHOWHEX Statement

PURPOSE

Like the SHOW statement, this statement prints the contents of a field in the *control listing* (not in the report). However this statement prints the raw contents of the field in character and hex format.

SYNTAX

SHOWHEX STATEMENT SYNTAX

```
SHOWHEX fieldname/'literal'
```

DISCUSSION

Only a single field or literal maybe specified on the SHOWHEX statement.

When a fieldname is specified, both the name of that field and its raw contents are printed in both character and hex format in the control listing (SYSPRINT).

When a literal text is specified, just that text itself prints in the control listing (in character and hex format.)

PARMS

fieldname/'literal'
Specifies a field or literal to show in hex format in the control listing.

EXAMPLE

```
SHOW 'FOLLOWING AMOUNT IS CORRUPTED:'
SHOWHEX AMOUNT
```

STOP Statement

PURPOSE

Stops execution of the program.

SYNTAX

STOP STATEMENT SYNTAX

```
STOP
```

DISCUSSION

You are not required to use a STOP statement in your program.

The run will end normally when:

- an auto-cycle run reaches EOF on the primary input, or
- when control reaches the last statement in your program, for standard (non auto-cycle) runs.

You can use this statement to stop the run at some other point during the processing of your code.

EXAMPLE

```
IF CODE <> 1 THRU 99
   SHOW 'INVALID CODE FOUND'
   SHOW CODE
   STOP
ENDIF
```

STORE Statement

PURPOSE

Stores a new record in a sequential table, or stores a new or modified record in a keyed table.

SYNTAX

STORE STATEMENT SYNTAX

```
STORE tablename
```

DISCUSSION

The STORE statement copies the current contents of the table's record area (the FIELD statements immediately following the TABLE statement) into the table's internal storage area. This is handled a little differently for sequential and keyed tables.

As a reminder, z/Writer supports two types of tables.

- **Sequential tables.** Tables defined without the KEY parm are sequential tables. For sequential tables, the record being stored is always added at the end of the table.
- **Keyed tables.** Tables defined with the KEY parm are keyed tables. Internally, z/Writer maintains keyed tables in key order as balanced binary trees. The STORE statement causes the record to be inserted into the tree according to the value of its key. If the table already has a record with that key, then the contents of the record are replaced with the table's current record area.

Status of a Store Operation

After the STORE statement, you can check the result of the operation by examining the table's #STATUS built-in fie. (Remember that you will need to qualify #STATUS with the tablename, if the phase has definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a STORE.

#STATUS Built-In Field Values After a STORE Statement

Y - record successfully stored. The table's internal structure contains the new record (or updated) record.
N - no record stored. For keyed tables, this indicates that there was an error with the key value. The internal table remains unchanged.

PARMS

tablename

Specifies the name of the table to store a record in. The table must have been previously defined using a TABLE statement.

EXAMPLES

```
STORE MY-SEQ-TABLE

STORE MY-KEY-TABLE
IF MY-KEY-TABLE.#STATUS <> 'Y'
   SHOW 'ERROR ADDING RECORD TO TABLE'
   STOP
ENDIF
```

TABLE Statement

PURPOSE

Defines a table to z/Writer.

SYNTAX

TABLE STATEMENT SYNTAX

```
TABLE  TABLENAME
       [ KEY(fieldname/nn,nn)       ]
```

DISCUSSION

A table can be thought of as a sort of "in-memory file." z/Writer supports two types of tables.

- **Sequential tables.** Tables defined without the KEY parm are sequential tables. When storing records in sequential tables, each record is simply added to the end of table, after all of the existing records. When retrieving records from a sequential table, you normally start at the beginning of the table and read through it sequentially. Thus, a sequential table is similar to an in-memory "flat file." (Unlike a flat file, however, it is also possible to directly access any record in a sequential table by referencing its "entry number".)
- **Keyed tables.** Keyed tables are tables defined with the KEY parm. z/Writer maintains keyed tables as balanced binary trees. Such structures are very efficient and easy to process. When storing records in a keyed table, z/Writer inserts them directly into the correct location based on their key value. Thus, one advantage of a binary table is that it is *always* sorted in key order, even while you are still adding new records to it. Therefore there is no need to sort the table before doing a "binary search." *All* retrievals from keyed tables are efficient binary searches. These table are easy to use since you can mix retrievals and stores without needing a sort command to get the table in order. All of the housekeeping associated with binary tables is automatically handled for you. A keyed table is similar to an in-memory "VSAM file." Records can be directly retrieved with their unique key value. (They can also be read sequentially, in order of their keys.)

The fields for the table must be defined immediately after the TABLE statement. Use FIELD statements to define the fields.

Note: use the STORE statement to add a record to a table, or to change a record in a keyed table. Use the RETRIEVE statement to retreive a record from a table. The DELTABREC statement can be used to delete a record from a keyed table.

Both types of table have some built-in fields available to examine. These are shown in the table below.

Built-In Fields Available for Tables

FIELDNAME	TYPE & LENGTH	READ ONLY	DESCRIPTION
#EOF	1-byte character	Yes	Indicates whether an attempt was made to sequentially retrieve a record past the last record in the table. (Keyed retrievals do not set the #EOF built-in field, even when the desired key is higher than any key in the table.) **Y** - the last operation was a sequential RETRIEVE statement which raised the end of file condition **N** - the last operation was not a sequential RETRIEVE statement which raised the end of file condition
#STATUS	1-byte character	Yes	Status of the most recent STORE/RETRIEVE operation. **(blank)** - uninitialized (no operations have been performed on table yet) **Y** - last STORE/RETRIEVE operation was successful **N** - last STORE/RETRIEVE operation was unsuccessful. Indicates missing record or end-of-table after RETRIEVE statements, or duplicate key after a STORE statement.
#COUNT	4-byte binary	Yes	Number of entries in the table.

PARMS

tablename

Specifies the name of the table being defined. All other control statements will use this name when referring to this file.

You may assign any name you like following the naming conventions in the box below.

Requirements for Table Names

- table names may be from 1 to 70 characters long
- table names may contain alphanumeric characters, the “national” characters #, @, and $, underscores (_) and dashes (-).
- table names must not begin with a numeric character
- table names are not case sensitive
- table names must not be statement names or other reserved words

```
Examples
SALES_TABLE
ACCOUNTS
```

KEY(field/nn,nn)

The presence of this parm indicates that the table is a keyed table. This parm specifies where the key is located within the table records. (Tables defined without this parm are sequential tables.)

In this parm, you may specify a single fieldname (which must be defined somewhere among the FIELD statements that follow the TABLE statement.

Or you can specify two numeric literals, instead. The first number specifies the column that the key begins in. (Columns are numbered starting with “1”). The second number specifies the length, in bytes, of the key in the record.

EXAMPLES

```
TABLE MY-SEQ-TABLE
TABLE MY-KEY-TABLE  KEY(1,3)

TABLE MY-KEY-TABLE  KEY(EMPL_NUM)
FLD EMPL_NUM 3
FLD EMPL_NAME 10
FLD HIRE_DATE 8 CONTAINS(YYYYMMDD)
```

TITLE Statement

PURPOSE

This is a declaratory statement that specifies how one title line will look in a report. TITLE statements can also be used to build your own column heading lines.

SYNTAX

TITLE STATEMENT SYNTAX

```
TITLE:
           [ (reportname)                          ]
           [   item1[(parms)]  item2[(parms)]  ... ]
           [ / item1[(parms)]  item2[(parms)]  ... ]
           [ / item1[(parms)]  item2[(parms)]  ... ]
```

Each **item** can be:

- a fieldname, including built-in fields (ex: AMOUNT)
- a built-in function (ex: #MAX(SALES_QTR1, SALES_QTR2))
- a numeric literal (ex: 1000)
- a literal text (ex: 'INITIAL WHEN READ: _____')

The syntax of the **parms** available within parentheses for fieldnames and literals is:

```
fieldname/function[(
           [  +n/@n                  ]
           [  n                      ]
           [  display-format         ]
           [  LJ/CJ/RJ               ]
           [  BIZ                    ]  )]

'literal'[(
           [  +n/@n                  ]
           [  n                      ]  )]
```

DISCUSSION

How Titles Are Printed

You may have as many TITLE statements for a report as you like. Each TITLE statement results in one title line at the top of the report. (TITLE statements are treated as comments for EXPORT outputs. Export files do not contain titles.)

As a declaratory (rather than executable) statement, the exact location of your TITLE statements is not critical. However, they must appear *after* the first PRINT statement. (This allows the titles to be centered correctly over the body of the report.)

The relative order of the TITLE statements is important. Title lines are printed in the same order in which they are found in the control statements.

z/Writer performs a "page eject" before printing the first title line, which puts it at the top of a new page. Before each remaining title line, a single "line feed" is performed before printing. An extra blank line is printed after the final title line. (This separates the titles from whatever follows next — the column headings or the report body.)

Empty TITLE statements are also allowed and result in a blank title line.

The TITLE statement consists of zero to three print expressions, separated with slashes. A "print expression" is just a list of fieldnames and/or literal texts to print, along with any optional parms for them. Fieldnames indicate that the contents of that field should appear in the title. Literal texts (in apostrophes or quotes) print "as is" in the title.

Optional parms (in parentheses after a fieldname or literal) let you customize the location and/or appearance of an item within the title line.

How to Justify Parts of a Title

Report titles are commonly divided into three parts: a left-hand part, a centered part, and a right-hand part. Counting out the columns by hand to format titles this way is tedious. And it must be redone each time you make any change to the title text or the report columns.

z/Writer lets you use **slashes** to greatly simplify this alignment of your title lines over the report. If a TITLE statement has **no slashes**, the single print expression will be centered over the report. If there is **one slash** in the TITLE statement, the print expression before the slash will be left aligned and the print expression after the slash will be right aligned. If there are **two slashes** in the statement, 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 (that is, to have two consecutive slashes.)

Auto-Completion of Titles

In many cases, z/Writer also helps you complete the first title line. If that first title contains only a single print expression (no slashes), that print expression will be centered across the report for you. z/Writer then adds the system date to the title, starting in column 1. And it adds the page number, aligned with the right margin of the report.

> **Note:** you can suppress either or both of these auto completions by using the NODATE and/or NOPAGE parms in the REPORT statement. Also, if you use a "@n" parm or a slash in the first TITLE statement, this completion logic is not performed (to avoid possibly overlapping your title layout.)

Built-In Fields Available for Titles

z/Writer has a number of built-in fields that are often used in titles. The most obvious ones are fields containing the current date and the page number. You can see a complete list of z/Writer's built-in fields

in Appendix A, "Built-In Fields" on page 182. A few of the most useful ones for titles are listed in the box below.

z/Writer Built-In Fields for Titles	
Field Name	**Description**
#DATE	An 8-byte character field containing the system date in MM/DD/YY format.
#DAYNAME	A 9-byte character field containing the day of the week ("MONDAY")
#TIME **#TIME12**	An 8-byte character field containing the system time in 12-hour format (ex: "12:45 PM").
#TIME24	An 8-byte character field containing the system time in 24-hour format (ex: "13:45:59").
#PAGENUM **#PAGE**	A numeric field containing the current page number of the report. (Learn more about this built-in field on page 182.)

PARMS

(reportname)

This parm is optional. When used, this *must* be the first parm on the TITLE statement. The report name must be enclosed in parentheses.

When report name is not present, the TITLE statement defines a title line for the primary (and often only) report in the current phase.

Use this optional parm if you are printing to more than one report output during a single phase. This parm names the report that you want this TITLE statement to be used for. (Reports are defined and named using the REPORT statement. See page 128.)

fieldname/'literal'[(parms)]

Specifies one item to include in the title line. Fieldnames indicate that the contents of that field should appear in the title. Literal texts (in apostrophes or quotes) print "as is" in the title.

Optionally, you can specify one or more parms in parentheses after the fieldname or literal. (There must be no space before the open parenthesis.) The parms let you customize the location and/or appearance of an item in the title line. The following table shows the parms that can be specified for each item. All parms are optional and they may be specified in any order, separated by spaces and/or commas. Note that some of these parms are irrelevant for literal items and will be ignored.

PARMS ALLOWED FOR ITEMS IN THE TITLE STATEMENT		
SYNTAX	**DESCRIPTION**	**EXAMPLE**
Parms Affecting an Item's Location in the Title		
+n	Means begin this item "n" bytes after the end of the previous item in the title. For example, you can specify +0 if *no spaces* are wanted after the previous item. **Note:** the "+" is required here in order to distinguish this parm from the width parm ("n"). z/Writer's default is to leave 1 space between items in the title.	`TITLE 'NAME=' NAME(+0)`
@n	@n means begin the data in column "n" of the title. Columns are numbered starting with 1. **Note:** use of this parm on the first TITLE statement prevents the auto completion of the title line (see page 148).	`TITLE 'JOB REPORT'(@40)` `'PAGE'(@125) #PAGENUM`
Parms Affecting an Item's Appearance in the Title		
n	Specifies an override width (in bytes) to use for this title item. When omitted, z/Writer picks a default width for each item.	`TITLE DESCRIPTION(10)`
LJ/CJ/RJ	Specifies how the contents of the item should be justified (*within* the area reserved for it in the title line). The values mean left-justified, center-justified and right-justified, respectively. **Note:** you can use slashes in the TITLE statement to justify *whole sections* of the title.	`TITLE DEPT-NUM(LJ)`
display-format	You may specify the name of any of z/Writer's "display formats" that are valid for the data type of the item. The display format determines how the data is formatted in the title line. (The table on page 117 lists all of the display formats.) When omitted, z/Writer uses: 1), the display format from the FORMAT parm of the FIELD statement, if any, or 2) a default display format.	`TITLE AMOUNT(CURRENCY)`
BIZ	(Stands for "blank-if-zero"). Use this with a numeric field if you want the area reserved for it in the title line to be left blank for zero values.	`TITLE ERROR-COUNT(BIZ)`

EXAMPLES

```
TITLE 'SALES REPORT'
TITLE #DATE / 'FOR STORE' STORE-NUM / 'PAGE' #PAGENUM(PIC'ZZ9')
TITLE 'COST CENTER: 1123' / 'ACCOUNTING DEPT' / 'GROUP' ACCT-GROUP
TITLE ' JOB'(@1) 'SALES'(@10) 'SALES'(@20) 'EMPL'(@30)
TITLE 'NAME'(@1) ' DATE'(@10) ' TIME'(@20) 'NAME'(@30) 'DESRIPTION'(@40)



FILE EMPL TYPE(KSDS)
...

REPORT LINES(55)
REPORT (PARTTIME) LINES(55)

PRINT EMPL_NAME HIRE_DATE STATUS

IF STATUS = 'P'
   PRINT (PARTTIME) EMPL_NAME HIRE_DATE STATUS
ENDIF

TITLE 'LIST OF ALL EMPLOYEES'
TITLE (PARTTIME) 'LIST OF PART TIME EMPLOYEES'
```

TRACEOFF STATEMENT6

PURPOSE

Turns program flow tracing off.

SYNTAX

TRACEOFF STATEMENT SYNTAX

```
TRACEOFF
```

DISCUSSION

Program flow tracing causes the program's statements to be printed (in the control listing) as they are executed. This feature can be useful in debugging new programs. Program flow tracing is off by default.

This is an executable (not declaratory) statement. Tracing begins when the program executes a TRACEON and ends when it executes a TRACEOFF statement. You can use these statement to activate tracing only for selected sections of a program.

EXAMPLES

```
READ SALES

TRACEON
IF #STATUS = 'Y'
   GOTO READ-OK
ELSE
    SHOW 'READ ERROR'
    STOP
ENDIF
TRACEOFF

PRINT EMPL-NUM EMPL-NAME
...
```

TRACEON STATEMENT

PURPOSE

Turns program flow tracing on.

SYNTAX

TRACEON STATEMENT SYNTAX

TRACEON

```
Alternate Spellings:
TRACEON - TRACE
```

DISCUSSION

Program flow tracing causes the program's statements to be printed (in the control listing) as they are executed. This feature can be useful in debugging new programs. Program flow tracing is off by default.

This is an executable (not declaratory) statement. Tracing begins when the program executes a TRACEON and ends when it executes a TRACEOFF statement. You can use these statement to activate tracing only for selected sections of a program.

EXAMPLES

```
READ SALES

TRACEON
IF #STATUS = 'Y'
   GOTO READ-OK
ELSE
    SHOW 'READ ERROR'
    STOP
ENDIF
TRACEOFF

PRINT EMPL-NUM EMPL-NAME
...
```

WHEN Statement

PURPOSE

The WHEN statement is used within a “case-structure”. The purpose of a case-structure is to conditionally execute (at most) one set of statements within the structure.

SYNTAX

CASE STRUCTURE SYNTAX

```
CASE fieldname

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]

[ WHEN  [NOT] value/range [value/range ...]
other statements  ]
...

[ ELSE
other statements  ]

ENDCASE
```

DISCUSSION

The WHEN statement presents one set of values to compare to the test field (named in the CASE statement.) It is followed by any number of “other statements” to be executed if a match is found among its values. A complete discussion of the “case-structure” can be found under the CASE statement on page 32.

WORKAREA Statement

PURPOSE

Allows you to define fields in working storage.

SYNTAX

WORKAREA STATEMENT SYNTAX

```
WORKAREA [name]
```

DISCUSSION

The WORKAREA statement has one optional parm, the NAME parm (described below.)

This statement should be immediately followed by one or more FIELD statements. These fields are not located within any file or table record area. They may be used to store any "working storage" values required by your program logic.

You may have multiple WORKAREA statements if you like. Most programs will use a single WORKAREA statement to define all necessary working variables. It is normally located just after all file and table definitions, near the beginning of a program.

Initial Values

Here is how z/Writer initializes the contents of a workarea. First, the entire workarea is cleared to blanks. Then, for each numeric field defined in the workarea, that field is initialized to zero if both of the following conditions are met:

- the field was not defined with its own INIT or REINIT value
- the field was not redefined by other fields (that is, it was not named in a REDEFINE statement)

PARMS

name

This optional parm assigns a name to the workarea being defined. When present, the name parm *must* be on the same line as WORKAREA. (Unlike most z/Writer statements, the WORKAREA statement may not be continued onto additional lines.) The naming rules for workareas are the same as for file names (page 78)

Use the workarea name when you need to qualify fieldnames defined within that workarea. (For example, if a field by the same name also exists in a file definition, or in another workarea.) For example:

```
WORKAREA COUNTERS
FLD ERRORS P3 INIT(0)
...
COUNTERS.ERRORS = COUNTERS.ERRORS +1
```

When a workarea name is not specified, the fields within that workarea can not be qualified.

EXAMPLES

```
WORKAREA MYWORK
FLD COUNTER P3 INIT(0)
FLD WORKKEY 5
REDEFINE WORKKEY
FLD WORKKEY-DEPT 1
FLD WORKKEY-SEQ N4

WORKAREA
BUILD-DATE 8
REDEFINE BUILD-DATE
BUILD-YYYY 4
BUILD-MM   2
BUILD-DD   2
```

WRITE Statement

PURPOSE

Writes a *new* record to an OUTPUT or UPDATE file.

SYNTAX

WRITE STATEMENT SYNTAX

```
WRITE   filename
```

DISCUSSION

For QSAM, ESDS and RRDS files, WRITE appends a new record to the end of the file.

For KSDS files, WRITE attempts to add a new record to the file using the key value contained in the key field in the file's record area. An error will occur if the file already contains a record with the same key value.

Note: do not use the WRITE statement to rewrite a record that has been read for update. Use the REWRITE statement for that. You may perform WRITEs to an UPDATE file, but only for adding new records to the file.

Note: after a WRITE to an RRDS file, the RRN of the newly written record can be found in the file's #RRN built-in field.

Record Lengths

Here is how an OUTPUT or UPDATE file's #LENGTH built-in field is used.

For **fixed-length** files, z/Writer initializes #LENGTH one time to the file's fixed record length (from the DCB or ACB at open time.). You can examine this field, but you should not change it.

For **variable-length** files (including variable length VSAM files), z/Writer initializes the #LENGTH built-in field one time to the *maximum* defined record length (from the DCB or ACB at open time.) After that, you are responsible for maintaining the correct value in #LENGTH. Specifically, before a WRITE to a variable-length file, you must ensure that the file's #LENGTH built-in field contains the correct length of the record being added.

Status of a Write Operation

After the WRITE statement, you can check the result of the operation by examining the file's #STATUS built-in field. (Remember that you will need to qualify #STATUS with the filename, if the phase has

definitions for multiple files and/or tables.) The table below shows the possible values of #STATUS after a WRITE.

#STATUS Built-In Field Values After a WRITE Statement

Y - record successfully written
N - record not written. This may indicate no space left in the file, or some other error.

PARMS

filename

Specifies the name of the file to write a new record to. The file must have been previously defined as an OUTPUT or UPDATE file.

EXAMPLE

```
FILE EMPL UPDATE TYPE(KSDS)
...

READ EMPL  KEY('400')         /* MODIFY REC 400 */
IF EMPL.#STATUS = 'Y'
    MOVE 'I' TO EMPL.STATUS   /* MARK 400 AS INACTIVE */
    MOVE 40 TO EMPL.#LENGTH   /* SHORTEN INACTIVE RECORD */
    REWRITE EMPL              /* REWRITE SHORTER 400 RECORD */
ENDIF


READ EMPL  KEY('500')         /* CHANGE REC 500'S KEY */
IF EMPL.#STATUS = 'Y'
    DELREC EMPL               /* REMOVE KEY 500 FROM FILE */
    MOVE '599' TO EMPL.KEY    /* USE SAME REC WITH DIFF KEY */
    WRITE EMPL               /* ADD A NEW 599 REC TO FILE */
ENDIF

STOP
```

Chapter 6. z/Writer's DB2 Option

This chapter explains how to use z/Writer's available DB2 Option to access DB2 data.

What Is z/Writer's DB2 Option?

z/Writer's DB2 Option enables z/Writer to access data from any of your shop's DB2 tables. With the DB2 Option, you can easily report on all of your DB2 data, using the same powerful, easy z/Writer language that you already know! And z/Writer lets you combine DB2 data with other data from flat files and VSAM files -- all in one report.

And, because z/Writer handles the interface to DB2 for you **dynamically**, there is no need for pre-compiles or any other extra steps. You just code DB2 reports and run them as quickly and easily as you do now with regular files.

How It Works

The DB2 Option adds two new z/Writer statements for you to use. (It also extends the use of several existing statements.)

New Statements Just for DB2

- The new CURSOR statement defines one DB2 input source. The CURSOR statement is required for any run that accesses DB2 data.
- The new FETCH statement fetches one row from a DB2 input source. The FETCH statement is not required. You can omit it and let z/Writer's auto-cycle logic handle the fetches for you automatically.

You can think of DB2 tables as being almost like regular files. You just code a CURSOR statement to describe the DB2 data that you want to access. This is the equivalent of the FILE statement for regular files. Then you use FETCH statements to retrieve one row at a time of data from your cursor. That is similar to a READ statement for a regular file. Also available (but not required) are the OPEN and CLOSE statements, which work for both files and DB2 cursors.

Now let's just take a quick look at each of the two new statement. Afterwards, we will go into more of the details.

The CURSOR Statement - A Quick Look

The CURSOR statement is the key to using DB2 data in your z/Writer program. You might think of it as a very powerful FILE statement. It not only names a DB2 input, but can also specify selection criteria and sort order, compute statistics, summarize data -- even merge data from other tables!

The QUERY Parm of the CURSOR Statement

In short, the CURSOR statement has all the power of a SQL "query." That is because you will actually include an SQL "query" (or "full select" statement) right in your CURSOR statement (in the QUERY parm.) Your query will begin with the word "SELECT" and can contain any number of WHERE conditions, as well as ORDER BY, HAVING, GROUP BY clauses, etc. It may include sub-select clauses. In general, any

query that could be specified in an SQL DECLARE CURSOR statement is allowed in the QUERY parm of z/Writer's CURSOR statement.

> **Note:** the query may *not* include an INTO clause. (Fetches are always performed "into" the fields that z/Writer automatically defines for the cursor. See page 164 for more details.)

Your QUERY parm must be written as a valid SQL query. However, z/Writer does not attempt to syntax-check the contents of your QUERY parm. Any errors in it will be detected later by DB2. Some of them are detected while the CURSOR statement is first being examined. Other errors are only detected by DB2 when the cursor is actually opened, after your program has begun execution.

z/Writer's CURSOR statement, in fact, establishes an actual SQL cursor (internally), which z/Writer later uses to fetch data from your DB2 table.

For example, here is a CURSOR statement with a very simple QUERY parm:

```
CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)
```

This simple query selects all of the columns from the sample IBM project table named DSN81110.PROJ. (This table also exists at your own shop, in case you would like to try these examples yourself. It may have a slightly different name depending on your version of DB2.)

Note that, like the FILE statement, the non-executable CURSOR statement does not actually return any data to the program. It simply defines the data that will be returned later (with FETCH statements.) Technically, the CURSOR statement (when later opened) creates a DB2 "result table." This result table contains rows of data columns that match the specifications in your QUERY parm. These rows will later be returned ("fetched"), one at a time, to your program. You may do this with explicit FETCH statements in your code. Or you can let z/Writer fetch the rows for you automatically in an auto-cycle report.

DB2 Report Example

Here is a complete z/Writer program that uses the above CURSOR statement to print a report from the DB2 project table.

```
OPTION DB2SSID('DBBG')

CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)

PRINT
  PROJNO
  PROJNAME
  DEPTNO
  RESPEMP
  PRSTAFF
  PRSTDATE
  PRENDATE
  MAJPROJ

TITLE 'CONTENTS OF PROJ DB2 TABLE'
```

Looks examine this short program in more detail.

The first line, OPTION DB2SSID('DBBG'), specifies which of your shop's DB2 subsystems should be used for the run. This option is always required for runs that use DB2 data. It should be coded early in the program. This parm is required so that z/Writer can find the DB2 data that you want. This parm also

These Control Statements:

```
OPTION DB2SSID('DBBG')

CURSOR PROJECT QUERY(SELECT * FROM DSN8111O.PROJ)

PRINT
  PROJNO
  PROJNAME
  DEPTNO
  RESPEMP
  PRSTAFF
  PRSTDATE
  PRENDATE
  MAJPROJ

TITLE 'CONTENTS OF PROJECT DB2 TABLE'
```

Produce this Report:

```
09/04/12                  CONTENTS OF PROJECT DB2 TABLE               PAGE     1

PROJNO  _______ PROJNAME _______ DEPTNO RESPEMP PRSTAFF  _PRSTDATE_  _PRENDATE_ MAJPROJ

AD3100 ADMIN SERVICES          D01    000010     6.50 1982-01-01 1983-02-01
AD3110 GENERAL AD SYSTEMS      D21    000070     6.00 1982-01-01 1983-02-01 AD3100
AD3111 PAYROLL PROGRAMMING     D21    000230     2.00 1982-01-01 1983-02-01 AD3110
AD3112 PERSONNEL PROGRAMMG     D21    000250     1.00 1982-01-01 1983-02-01 AD3110
AD3113 ACCOUNT.PROGRAMMING     D21    000270     2.00 1982-01-01 1983-02-01 AD3110
IF1000 QUERY SERVICES          C01    000030     2.00 1982-01-01 1983-02-01
IF2000 USER EDUCATION          C01    000030     1.00 1982-01-01 1983-02-01
MA2100 WELD LINE AUTOMATION    D01    000010    12.00 1982-01-01 1983-02-01
MA2110 W L PROGRAMMING         D11    000060     9.00 1982-01-01 1983-02-01 MA2100
MA2111 W L PROGRAM DESIGN      D11    000220     2.00 1982-01-01 1982-12-01 MA2110
MA2112 W L ROBOT DESIGN        D11    000150     3.00 1982-01-01 1982-12-01 MA2110
MA2113 W L PROD CONT PROGS     D11    000160     3.00 1982-02-15 1982-12-01 MA2110
OP1000 OPERATION SUPPORT       E01    000050     6.00 1982-01-01 1983-02-01
OP1010 OPERATION               E11    000090     5.00 1982-01-01 1983-02-01 OP1000
OP2000 GEN SYSTEMS SERVICES    E01    000050     5.00 1982-01-01 1983-02-01
OP2010 SYSTEMS SUPPORT         E21    000100     4.00 1982-01-01 1983-02-01 OP2000
OP2011 SCP SYSTEMS SUPPORT     E21    000320     1.00 1982-01-01 1983-02-01 OP2010
OP2012 APPLICATIONS SUPPORT    E21    000330     1.00 1982-01-01 1983-02-01 OP2010
OP2013 DB/DC SUPPORT           E21    000340     1.00 1982-01-01 1983-02-01 OP2010
PL2100 WELD LINE PLANNING      B01    000020     1.00 1982-01-01 1982-09-15 MA2100

GRAND TOTAL                                    73.50
```

Figure 9. A complete DB2 auto-cycle report

allows you to run your program first on a test system, and then easily switch it over to a production system.

The second line is the simple CURSOR statement that we saw earlier. It selects all columns of data from all rows of IBM's sample project DB2 table. The cursor name that we provided (PROJECT) is an arbitrary name. You can choose any name for the cursor. This same name will be used later in any OPEN, FETCH and CLOSE statements for this cursor. There are some other optional parms that can also be used on the CURSOR statement. But the cursor name and the QUERY parm are always required.

The last two statements are just the regular PRINT and TITLE statements that you are already familiar with.

The PRINT statement prints each of the columns from the project table. Note that we did not need to define any of those columns (or "fields" in z/Writer terms.) z/Writer obtained the name of each result column from DB2 and defined a like-named z/Writer field for each one. During a fetch operation, data is returned from DB2 and loaded into these z/Writer fields for your program to use. If you would like to see all of the fields created for your CURSOR statement (along with their length and type), just add the SHOWFLDS parm to your CURSOR statement:

```
CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ) SHOWFLDS
```

Note that this is an auto-cycle report, since there are no explicit FETCH statements for the primary input. The primary (and only) input in this program is the DB2 cursor for PROJECT.

The output from this simple program is shown in Figure 9 (page 161).

The FETCH Statement -- A Quick Look

The sample DB2 report in Figure 9 (page 161) did not use the FETCH statement. That caused z/Writer to perform an auto-cycle report. That is, it fetched each row from our primary input (the DB2 cursor) until it reached "end of file" (no more rows). For each row fetched, it performed the single executable program statement -- the PRINT statement. It also printed a grand total line, which is the default.

If you prefer, you can code explicit FETCH statements in your program. That way you are free to code any kind of program flow that you want.

The FETCH statement simply names a previously defined DB2 cursor. It has no other parms. Its function is to fetch the next row from the result table defined by the cursor. The rows' data columns are loaded into the fields defined for your cursor by z/Writer (see page 164).

The status of the FETCH operation can be determined in several ways:

- you can check the standard #STATUS built-in field, just as for a file. ("Y" means successful; "N" means not successful.)
- you can explicitly check for EOF by looking at the #EOF built-in field, as for a file. ("Y" means no more rows; "N" means another row was returned.)
- or, you can check the special #SQLCODE built-in field, which exists only for DB2 cursors. This is a 2-byte binary numeric field containing the actual SQL code returned by DB2 to z/Writer.

Figure 10 (page 163) shows the same short program that we saw earlier in Figure 9 (page 161). But this time the program uses explicit FETCH statements. (Note that while you may also use the OPEN and CLOSE statements if you like, they are not required. z/Writer performs them by default when they are needed.)

In this program, we first do a single FETCH to "prime the pump." Then we enter a do-while loop for as long as the #EOF built-in field is not "Y." In the loop we print the current contents of the DB2 cursor's fields and then do another FETCH. When EOF is reached, the do-while loop ends and the program ends.

Note that the only difference between this report and the report on page 161 is the absence of grand totals. Auto-cycle reports print grand totals automatically. In non-auto-cycle reports, any desired totalling logic must be coded by the programmer.

These Control Statements:

```
OPTION DB2SSID('DBBG')
CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)

FETCH PROJECT
DOWHILE #EOF <> 'Y'
   PRINT
     PROJNO
     PROJNAME
     DEPTNO
     RESPEMP
     PRSTAFF
     PRSTDATE
     PRENDATE
     MAJPROJ
   FETCH PROJECT
ENDDO

TITLE 'CONTENTS OF PROJECT DB2 TABLE'
```

Produce this Report:

```
09/04/12                  CONTENTS OF PROJECT DB2 TABLE               PAGE     1

PROJNO  ______PROJNAME_______ DEPTNO RESPEMP PRSTAFF  PRSTDATE  PRENDATE  MAJPROJ

AD3100 ADMIN SERVICES         D01    000010     6.50 1982-01-01 1983-02-01
AD3110 GENERAL AD SYSTEMS     D21    000070     6.00 1982-01-01 1983-02-01 AD3100
AD3111 PAYROLL PROGRAMMING    D21    000230     2.00 1982-01-01 1983-02-01 AD3110
AD3112 PERSONNEL PROGRAMMG    D21    000250     1.00 1982-01-01 1983-02-01 AD3110
AD3113 ACCOUNT.PROGRAMMING    D21    000270     2.00 1982-01-01 1983-02-01 AD3110
IF1000 QUERY SERVICES         C01    000030     2.00 1982-01-01 1983-02-01
IF2000 USER EDUCATION         C01    000030     1.00 1982-01-01 1983-02-01
MA2100 WELD LINE AUTOMATION   D01    000010    12.00 1982-01-01 1983-02-01
MA2110 W L PROGRAMMING        D11    000060     9.00 1982-01-01 1983-02-01 MA2100
MA2111 W L PROGRAM DESIGN     D11    000220     2.00 1982-01-01 1982-12-01 MA2110
MA2112 W L ROBOT DESIGN       D11    000150     3.00 1982-01-01 1982-12-01 MA2110
MA2113 W L PROD CONT PROGS    D11    000160     3.00 1982-02-15 1982-12-01 MA2110
OP1000 OPERATION SUPPORT      E01    000050     6.00 1982-01-01 1983-02-01
OP1010 OPERATION              E11    000090     5.00 1982-01-01 1983-02-01 OP1000
OP2000 GEN SYSTEMS SERVICES   E01    000050     5.00 1982-01-01 1983-02-01
OP2010 SYSTEMS SUPPORT        E21    000100     4.00 1982-01-01 1983-02-01 OP2000
OP2011 SCP SYSTEMS SUPPORT    E21    000320     1.00 1982-01-01 1983-02-01 OP2010
OP2012 APPLICATIONS SUPPORT   E21    000330     1.00 1982-01-01 1983-02-01 OP2010
OP2013 DB/DC SUPPORT          E21    000340     1.00 1982-01-01 1983-02-01 OP2010
PL2100 WELD LINE PLANNING     B01    000020     1.00 1982-01-01 1982-09-15 MA2100
```

Figure 10. A complete DB2 report using explicit FETCHs (not auto-cycle)

The CURSOR Statement -- More Details

We have learned the basic functions of the new CURSOR and FETCH statements for DB2 reports. Now let's look at the CURSOR statement a little more closely. This powerful statement will be the heart of your z/Writer runs that use DB2 data.

Where Does z/Writer Put the Data It Fetches from DB2?

After a FILE statement you normally code a number of FIELD statements. These describe the data fields that will be filled in after a record has been read from the file.

But after a CURSOR statement, FIELD statements are *not allowed*. Instead, z/Writer queries the DB2 subsystem as to what columns of data will be returned by the select statement in your QUERY parm. z/Writer then automatically defines one field for each column that DB2 will return.

The fields that z/Writer defines for you have these characteristics:

- the **field name** is the same as the DB2 column name, if it has one.

  Some result columns do not come directly from a DB2 column and do not have an SQL "name." For example, if you select AVG(AMOUNT), that result column is not given a name by SQL. In such cases, z/Writer assigns its own field name of the form CALC_nn. That is, the first unnamed column will be put in a field named CALC_01, the next one in CALC_02, and so on. (See an example of this on page 173.)

- for DB2 columns defined with packed, binary, float or character SQL **data types**, the z/Writer field will have the corresponding z/Writer data type. For all other types of DB2 columns, the z/Writer field is just defined as a character field.

- the **length** of the z/Writer field is the same as the SQL length of the column, *except* for decimal fields. For decimal fields, z/Writer uses the "scale" value from the SQL length parm to determine the corresponding length in bytes for the field. It also uses SQL's "precision" value to set the DECIMAL parm for the z/Writer field.

Queries That Only Return Certain DB2 Columns; the SELECT Clause

The sample run on page 170 showed a very basic CURSOR statement. It returned *all columns* of all rows in the table. Now let's make a report that only returns to us a few columns that we actually need for a report. To limit the columns returned, just name specific columns in the SELECT clause of your QUERY parm (instead of using "*" which means *all* columns).

For example, Figure 11 below only uses two columns from the DB2 project table:

Note that returning unnecessary columns in the result table does not harm your final report at all. But when CPU efficiency is a prime concern, you will probably want to limit the columns returned to just those that you actually need for your run.

Using the ORDER BY Clause

We also added an **ORDER BY** clause to our QUERY parm in Figure 11. That ensures that the rows that we fetch are passed to us in project number order.

These Control Statements:

```
OPTION DB2SSID('DBBG')
CURSOR PROJECT QUERY(SELECT PROJNO, PROJNAME FROM DSN81110.PROJ
                           ORDER BY PROJNO)
PRINT
  PROJNO
  PROJNAME

TITLE 'PROJNO''S WITH DESCRIPTIONS'
```

Produce this Report:

```
 09/04/12 PROJNO'S WITH DESCRIPTIONS PAGE     1

PROJNO  ______  PROJNAME  ______

AD3100 ADMIN SERVICES
AD3110 GENERAL AD SYSTEMS
AD3111 PAYROLL PROGRAMMING
AD3112 PERSONNEL PROGRAMMG
AD3113 ACCOUNT.PROGRAMMING
IF1000 QUERY SERVICES
IF2000 USER EDUCATION
MA2100 WELD LINE AUTOMATION
MA2110 W L PROGRAMMING
MA2111 W L PROGRAM DESIGN
MA2112 W L ROBOT DESIGN
MA2113 W L PROD CONT PROGS
OP1000 OPERATION SUPPORT
OP1010 OPERATION
OP2000 GEN SYSTEMS SERVICES
OP2010 SYSTEMS SUPPORT
OP2011 SCP SYSTEMS SUPPORT
OP2012 APPLICATIONS SUPPORT
OP2013 DB/DC SUPPORT
PL2100 WELD LINE PLANNING

GRAND TOTAL
```

Figure 11. A DB2 report with a SELECT and ORDER BY clause

Using the WHERE Clause to Return Selected Rows from a DB2 Table

The sample run on page 170 showed a very basic CURSOR statement. It returned all columns of *all rows* in the table.

Often, you will not want to retrieve every row of a DB2 table. In those cases, SQL offers the WHERE clause for use in your QUERY. The SQL WHERE clause allows you to narrow down the rows that will later be retrieved.

Figure 12 shows an example of a QUERY parm with a WHERE clause.

This query is similar to the earlier one in Figure 9 (page 161). But this time the cursor contains a WHERE clause to select only the rows whose PRSTAFF column is greater than 1. The resulting report shows only the rows from the table that meet this WHERE condition.

These Control Statements:

```
OPTION DB2SSID('DBBG')

CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ
                       WHERE PRSTAFF > 1 )

PRINT
  PROJNO
  PROJNAME
  DEPTNO
  RESPEMP
  PRSTAFF
  MAJPROJ

TITLE 'PRSTAFF GREATER THAN 1'
```

Produce this Report:

```
09/05/12           PRSTAFF GREATER THAN 1          PAGE     1

PROJNO  ______PROJNAME_______ DEPTNO RESPEMP PRSTAFF MAJPROJ
AD3100 ADMIN SERVICES         D01    000010     6.50
AD3110 GENERAL AD SYSTEMS     D21    000070     6.00 AD3100
AD3111 PAYROLL PROGRAMMING    D21    000230     2.00 AD3110
AD3113 ACCOUNT.PROGRAMMING    D21    000270     2.00 AD3110
IF1000 QUERY SERVICES         C01    000030     2.00
MA2100 WELD LINE AUTOMATION   D01    000010    12.00
MA2110 W L PROGRAMMING        D11    000060     9.00 MA2100
MA2111 W L PROGRAM DESIGN     D11    000220     2.00 MA2110
MA2112 W L ROBOT DESIGN       D11    000150     3.00 MA2110
MA2113 W L PROD CONT PROGS    D11    000160     3.00 MA2110
OP1000 OPERATION SUPPORT      E01    000050     6.00
OP1010 OPERATION              E11    000090     5.00 OP1000
OP2000 GEN SYSTEMS SERVICES   E01    000050     5.00
OP2010 SYSTEMS SUPPORT        E21    000100     4.00 OP2000

GRAND TOTAL                                    67.50
```

Figure 12. Using the WHERE clause to select only certain rows from a DB2 table

Note also that while we *selected* all columns from the PROJ table, we did not print all of them in this report. (We did not print the PRSTDATE and PRENDATE columns.) You can just print the columns that you need.

Using Working Storage Fields in Your Query

Many WHERE clauses simply compare a DB2 column to a constant, literal value (as we did in Figure 12). But sometimes you will not know the selection criteria until the program execution begins. For example, you might read a control card into your program which specifies a date range to report on.

For these cases, the WHERE clause is allowed to refer to working storage fields within your z/Writer program. For example, you may want to select rows where a certain DB2 column's value matches a value in a workarea field of your program. In your WHERE clause you may refer to any *previously* defined field from a file, table, workarea or even from an earlier DB2 cursor. Just prefix the field name with a colon (:) when using it in your QUERY parm. Do not leave a space between the colon and the field name.

For example, in Figure 13 we compare the DB2 PRSTAFF column to the contents of a working storage field named TEST-STAFF.

Our report program assigned a value of "3" to TEST-STAFF before performing the first FETCH. Therefore, the report contains only the rows of the PROJ table with PRSTAFF greater than 3.

When is the DB2 Result Table Created?

In Figure 13 notice that we set the value in TEST-STAFF *before* we did any fetches. That is important.

When the first FETCH is needed, z/Writer first *opens the cursor*, and then performs that first fetch. The result table defined by the cursor is actually created by DB2 when the cursor is opened. The value in your working storage fields at open time is the value that DB2 will use to select rows for the result table. If you later change the value in such working storage fields, it will not affect the existing result table.

On the other hand, if you change the value in a working storage field and then close and re-open the cursor, the previous result table will no longer exist. Instead, a new result table is created for the cursor, using the current value in any referenced working storage fields.

How z/Writer Fields Are Passed to DB2

The following types of data are recognized by both z/Writer and SQL:

- character
- decimal (packed)
- binary
- float (hexadecimal floating point)

When you refer to a working storage field with one of these data types (in your QUERY parm), z/Writer passes its value to DB2 using the corresponding SQL data type. Data from a working storage field of *any other* type (namely z/Writer's character-numeric data types N and NE) is passed to DB2 as character data. That is because DB2 does not have a data type for such character-numeric fields.

These Control Statements:

```
OPTION DB2SSID('DBBG')

WORKAREA
TEST-STAFF P6.1 INIT(3)

CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ
                     WHERE PRSTAFF > :TEST-STAFF)

PRINT
  PROJNO
  PROJNAME
  DEPTNO
  RESPEMP
  PRSTAFF
  PRSTDATE
  PRENDATE
  MAJPROJ

TITLE 'PRSTAFF GREATER THAN CUTOFF VALUE'
```

Produce this Report:

```
09/05/12               PRSTAFF GREATER THAN CUTOFF VALUE              PAGE     1

PROJNO  _______ PROJNAME _______ DEPTNO RESPEMP PRSTAFF  PRSTDATE  PRENDATE  MAJPROJ
AD3100 ADMIN SERVICES          D01    000010     6.50 1982-01-01 1983-02-01
AD3110 GENERAL AD SYSTEMS      D21    000070     6.00 1982-01-01 1983-02-01 AD3100
MA2100 WELD LINE AUTOMATION    D01    000010    12.00 1982-01-01 1983-02-01
MA2110 W L PROGRAMMING         D11    000060     9.00 1982-01-01 1983-02-01 MA2100
OP1000 OPERATION SUPPORT       E01    000050     6.00 1982-01-01 1983-02-01
OP1010 OPERATION               E11    000090     5.00 1982-01-01 1983-02-01 OP1000
OP2000 GEN SYSTEMS SERVICES    E01    000050     5.00 1982-01-01 1983-02-01
OP2010 SYSTEMS SUPPORT         E21    000100     4.00 1982-01-01 1983-02-01 OP2000

GRAND TOTAL                                   53.50
```

Figure 13. A WHERE clause that refers to a working storage field in the program

Passing Exotic SQL Data Types in Your Query

SQL defines many more types of data than z/Writer. (For example, dates, times, and timestamps.) z/Writer provides a special method to allow you to pass such "exotic" SQL data types to DB2 in your query.

Let's assume that you have a 10-byte character working storage field. In this field you plan to store a date in the standard SQL format. How can you pass this character field to DB2 as a "date" type field? The answer is to add the special SQLTYPE and SQLLEN parms to the FIELD statement that defines your 10-byte character field:

```
FLD MY-DATE 10  SQLTYPE(384) SQLLEN(10)
```

The statement above defines a field call MY-DATE that is a 10-byte character field. In addition, it specifies a SQLTYPE and SQLLEN parm. These parms are *only used* when MY-DATE is passed to DB2 in a CURSOR statement's QUERY parm.

As z/Writer executes all of the program's regular (non-DB2) statements, MY-DATE is treated as a regular 10-byte character field. However, if you refer to :MY-DATE within the CURSOR statement's QUERY parm, z/Writer will pass that 10-byte area to DB2 as a type 384 field (an SQL date). The length passed to DB2 will be 10 (from the SQLLEN parm, not from the z/Writer field's length). If we had not specified the SQLTYPE and SQLLEN parms on the FIELD statement, z/Writer would have just passed the data to DB2 as a 10-byte character field.

Here are some points to keep in mind about the SQLTYPE and SQLLEN parms:

- SQLLEN will usually have the same length as the regular z/Writer field. When this is the case, it is not necessary to specify the SQLLEN parm on your FIELD statement.

  However, some SQL types require that the length parm be specified as a 1-byte binary scale (number of digit positions) followed by a 1-byte binary precision (number of decimal digits). In such case, you will use SQLLEN to specify a single halfword size value that SQL will re-interpret as two, 1-byte binary values. You can compute this value by multiplying the scale value by 256, and then adding the precision value. Remember that for decimal (packed) fields, the *scale* (digits) is not the same as the *length* (in bytes) of the field.

- the SQLTYPE and SQLLEN parms can also be used to pass "null" values to DB2 for a working storage value. See "Passing Null Values to SQL in Your Query" on page 174 for the details and an example.

Opening the Cursor Multiple Times

Many runs just define a single DB2 cursor and then fetch all of its rows to print a report. For these kinds of runs, you do not need to OPEN nor CLOSE the DB2 cursor yourself. z/Writer will do that for you (just before the first row is fetched, and at the end of the run.)

However, there are some occasions when you need to open a DB2 cursor over and over during the same run. Each time the DB2 cursor is opened, a new result table is prepared for it. When the query involves working storage fields, the values in those fields may have changed since the previous open. Each time you open the cursor, you can potentially get a different result table. It is in these cases that you will need to perform OPENs and CLOSEs to the cursor yourself.

For example, in Figure 14 we defined two DB2 cursors. One cursor is for the Project table and the other is for the Department table. Since we defined the Project cursor first, that becomes the primary input for the report. z/Writer will read through the Project table sequentially during its auto-cycle logic. z/Writer will not perform any fetches to the second, Department, cursor on its own. Accessing that cursor is entirely up to the program code.

In our auto-cycle program, each time z/Writer fetches the next row from the primary input (PROJECT), our code opens the DEPT cursor and FETCHes one row from it. We then CLOSE that cursor, so that it can be opened again for the next primary input row. This single FETCH from the Department table uses the DEPTNO column from the PROJECT cursor to look up the Department table's row for that department number. The selected row contains the complete department name. So now our report includes data from the Project table, plus the department name taken from the Department table.

These Control Statements:

```
OPTION DB2SSID('DBBG')

CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)

CURSOR DEPT QUERY(SELECT * FROM DSN81110.DEPT
                  WHERE DEPTNO = :DEPTNO )
*
********* READ DEPT RECORD FOR PROJECT'S DEPT ************
OPEN  DEPT   /* CREATE FRESH RESULT TABLE   */
FETCH DEPT   /* FETCH SINGLE ROW            */
CLOSE DEPT   /* CLOSE CURSOR FOR NEXT USE   */

PRINT
     PROJNO
     PROJNAME
     PROJECT.DEPTNO   /* DEPTNO IS IN BOTH CURSORS. QUALIFY IT */
     DEPTNAME         /* DEPT NAME COMES FROM 2ND CURSOR */
     RESPEMP
     PRSTAFF

TITLE 'PROJECTS LISTING'
TITLE 'WITH DEPARTMENT INFO'
```

Produce this Report:

```
09/08/12                          PROJECTS LISTING                          PAGE    1
                                WITH DEPARTMENT INFO

PROJNO  PROJNAME          DEPTNO  DEPTNAME                RESPEMP  PRSTAFF
AD3100 ADMIN SERVICES         D01   DEVELOPMENT CENTER          000010     6.50
AD3110 GENERAL AD SYSTEMS     D21   ADMINISTRATION SYSTEMS      000070     6.00
AD3111 PAYROLL PROGRAMMING    D21   ADMINISTRATION SYSTEMS      000230     2.00
AD3112 PERSONNEL PROGRAMMG    D21   ADMINISTRATION SYSTEMS      000250     1.00
AD3113 ACCOUNT.PROGRAMMING    D21   ADMINISTRATION SYSTEMS      000270     2.00
IF1000 QUERY SERVICES         C01   INFORMATION CENTER          000030     2.00
IF2000 USER EDUCATION         C01   INFORMATION CENTER          000030     1.00
MA2100 WELD LINE AUTOMATION   D01   DEVELOPMENT CENTER          000010    12.00
MA2110 W L PROGRAMMING        D11   MANUFACTURING SYSTEMS       000060     9.00
MA2111 W L PROGRAM DESIGN     D11   MANUFACTURING SYSTEMS       000220     2.00
MA2112 W L ROBOT DESIGN       D11   MANUFACTURING SYSTEMS       000150     3.00
MA2113 W L PROD CONT PROGS    D11   MANUFACTURING SYSTEMS       000160     3.00
OP1000 OPERATION SUPPORT      E01   SUPPORT SERVICES            000050     6.00
OP1010 OPERATION              E11   OPERATIONS                  000090     5.00
OP2000 GEN SYSTEMS SERVICES   E01   SUPPORT SERVICES            000050     5.00
OP2010 SYSTEMS SUPPORT        E21   SOFTWARE SUPPORT            000100     4.00
OP2011 SCP SYSTEMS SUPPORT    E21   SOFTWARE SUPPORT            000320     1.00
OP2012 APPLICATIONS SUPPORT   E21   SOFTWARE SUPPORT            000330     1.00
OP2013 DB/DC SUPPORT          E21   SOFTWARE SUPPORT            000340     1.00
PL2100 WELD LINE PLANNING     B01   PLANNING                    000020     1.00

GRAND TOTAL                                                                73.50
```

Figure 14. Opening a DB2 Cursor Multiple Times; Using Two DB2 Tables

Note that there is a column named DEPTNO in two of our cursors (PROJECT and DEPT). For that reason, we were required to qualify DEPTNO when we included it in the PRINT statement. Without the qualification, z/Writer would not know which of the two fields to print.

However, we did not need to qualify DEPTNO in the CURSOR statement for DEPT (in our WHERE clause):

```
WHERE DEPTNO = :DEPTNO
```

Here is the explanation. The first DEPTNO in the WHERE parm (without a colon) can only refer to the column by that name in the table being selected "from" (DEPT). Fieldnames without a leading colon always refer to columns in the DB2 table being selected from (not to z/Writer working storage fields). The reference to :DEPTNO (with a colon) is not ambiguous because, while this CURSOR statement for DEPT is being examined, the only z/Writer field named DEPTNO exists in the earlier PROJECT cursor. However, for any statement *after* the DEPT cursor statement, there are two z/Writer fields with this same name. From that point on, you need to qualify DEPTNO when referencing one of those fields. That is what we did in the PRINT statement.

Note that while not required, you are *allowed* to qualify the :DEPT-NO working storage field in the WHERE clause, if you like. Then it would look like this:

```
WHERE DEPTNO = :PROJECT.DEPTNO
```

Multiple DB2 Cursors

In the example in Figure 15, we have added a third DB2 cursor to the previous example. The primary input (PROJECT) includes the employee number of the employee responsible for the project (in the RESPEMP field). So, we added a cursor for the DB2 Employee table. The QUERY parm selects just the row for the employee number in the RESPEMP field (of the PROJECT cursor.)

Now our report still reads sequentially through all rows in the PROJECT table. (This is done automatically in the auto-cycle report.) For each PROJECT row retrieved, we then open and fetch a single row from two other DB2 cursors. From the DEPT cursor, we get the full name of the department owning the project. And from the EMPL cursor, we get the last name of the employee responsible for the project.

Our report now includes all of this data, coming from three different DB2 sources.

These Control Statements:

```
OPTION DB2SSID('DBBG')

CURSOR PROJECT QUERY(SELECT * FROM DSN81110.PROJ)

CURSOR DEPT QUERY(SELECT * FROM DSN81110.DEPT
                  WHERE DEPTNO = :DEPTNO)

CURSOR EMPL QUERY(SELECT * FROM DSN81110.EMP
                  WHERE EMPNO = :RESPEMP)
*
********* READ DEPT RECORD FOR PROJ'S DEPTNO **********
OPEN  DEPT   /* CREATE FRESH RESULT TABLE   */
FETCH DEPT   /* FETCH SINGLE ROW            */
CLOSE DEPT   /* CLOSE CURSOR FOR NEXT USE   */

********* READ EMPL RECORD FOR PROJ'S RESP EMP ********
OPEN  EMPL   /* CREATE FRESH RESULT TABLE   */
FETCH EMPL   /* FETCH SINGLE ROW            */
CLOSE EMPL   /* CLOSE CURSOR FOR NEXT USE   */

PRINT
     PROJNO
     PROJNAME
     PROJECT.DEPTNO   /* DEPTNO IS IN BOTH CURSORS. QUALIFY IT */
     DEPTNAME         /* DEPT NAME COMES FROM 2ND CURSOR */
     RESPEMP
     LASTNAME('RESPEMP|NAME') /*LAST NAME COMES FROM 3RD CURSOR */
     PRSTAFF

TITLE 'PROJECT LISTING'
TITLE 'WITH DEPARTMENT AND MANAGER INFO'
```

Produce this Report:

```
09/08/12                              PROJECT LISTING                              PAGE    1
                                  WITH DEPARTMENT AND MANAGER INFO


                                                                          RESPEMP
PROJNO  PROJNAME          DEPTNO  DEPTNAME                  RESPEMP   NAME       PRSTAFF
AD3100 ADMIN SERVICES      D01    DEVELOPMENT CENTER        000010  HAAS          6.50
AD3110 GENERAL AD SYSTEMS  D21    ADMINISTRATION SYSTEMS    000070  PULASKI       6.00
AD3111 PAYROLL PROGRAMMING D21    ADMINISTRATION SYSTEMS    000230  JEFFERSON     2.00
AD3112 PERSONNEL PROGRAMMG D21    ADMINISTRATION SYSTEMS    000250  SMITH         1.00
AD3113 ACCOUNT.PROGRAMMING D21    ADMINISTRATION SYSTEMS    000270  PEREZ         2.00
IF1000 QUERY SERVICES      C01    INFORMATION CENTER        000030  KWAN          2.00
IF2000 USER EDUCATION      C01    INFORMATION CENTER        000030  KWAN          1.00
MA2100 WELD LINE AUTOMATION D01   DEVELOPMENT CENTER        000010  HAAS         12.00
MA2110 W L PROGRAMMING     D11    MANUFACTURING SYSTEMS     000060  STERN         9.00
MA2111 W L PROGRAM DESIGN  D11    MANUFACTURING SYSTEMS     000220  LUTZ          2.00
MA2112 W L ROBOT DESIGN    D11    MANUFACTURING SYSTEMS     000150  ADAMSON       3.00
MA2113 W L PROD CONT PROGS D11    MANUFACTURING SYSTEMS     000160  PIANKA        3.00
OP1000 OPERATION SUPPORT   E01    SUPPORT SERVICES          000050  GEYER         6.00
OP1010 OPERATION           E11    OPERATIONS                000090  HENDERSON     5.00
OP2000 GEN SYSTEMS SERVICES E01   SUPPORT SERVICES          000050  GEYER         5.00
OP2010 SYSTEMS SUPPORT     E21    SOFTWARE SUPPORT          000100  SPENSER       4.00

(additional lines not shown)
```

Figure 15. A Run Using Three DB2 Cursors

Calculations in Queries

As we mentioned earlier, the select statement in your QUERY can be almost any query that could be specified in an SQL DECLARE CURSOR statement. Figure 16 below is an example of a more complex query. It includes a *subselect* clause. And it lets SQL handle the *averaging* and *summarization* logic that your program would otherwise have to perform. This example summarizes the Employee table

These Control Statements:

```
OPTION SSID('DBBG')
REPORT NOGRANDTOTALS

CURSOR EMPL QUERY(
       SELECT HIREYEAR, AVG(SALARY)
       FROM (SELECT YEAR(HIREDATE) AS HIREYEAR, SALARY
       FROM DSN81110.EMP) AS NEWEMP
       GROUP BY HIREYEAR )
       SHOWFLDS

PRINT
  HIREYEAR(NOCOMMAS)
  CALC_01('AVERAGE|SALARY' PIC'ZZZ,ZZ9.99')

TITLE
TITLE 'AVERAGE SALARIES'
```

Produce this Report:

```
09/05/12   PAGE     1
  AVERAGE SALARIES

            AVERAGE
 HIREYEAR    SALARY
      1947  23,840.00
      1949  40,175.00
      1958  46,500.00
      1963  29,250.00
      1964  15,900.00
      1965  39,733.33
      1966  24,960.00
      1967  26,250.00

(additional lines not shown)
```

Figure 16. A Summary Type DB2 Report Showing Average Values

records by hire year. It then returns one row per year, along with the average salary for those employees.

Note that we used CALC_01 as the name of the calculated column, AVG(SALARY), that the select returned. Columns in the result table that do not come directly from a column in the DB2 table are named in this manner (CALC_01, CALC_02, ...).

Passing Null Values to SQL in Your Query

By default, z/Writer passes all working storage fields referenced in your query to DB2 as fields that cannot contain a null value.

If you need to be able to pass "null" values in your queries, you can use the SQLTYPE and SQLLEN parms in the appropriate FIELD statement to accomplish this yourself. Follow these steps:

- define your z/Writer field as a character field that is 2 bytes longer than the actual data it will contain
- set the SQLTYPE to the *odd* type code that you want (for example 487). Odd SQL types are data types that may contain null values.
- set the SQLLEN to the correct length for the data you will be passing. For character data, this will be the length of the z/Writer field minus 2. For decimal data, it will be the special value (precision * 256 + scale) required by DB2.
- now redefine the z/Writer field. Redefine the first 2 bytes as character data (for your null indicator). Redefine the rest of the field however you like (binary, decimal, character, etc.)

Here is what z/Writer does when it passes fields with an *odd SQLTYPE* to DB2:

- it uses your SQLTYPE and SQLLEN just as you provide them.
- it passes the first 2 bytes of your z/Writer field as the null indicator for the value.
- it passes the rest of the z/Writer field as the actual data value

For example, to pass a null decimal value to DB2 in a working storage field, you could use this:

```
FLD MY-SQL-AMT C10 SQLTYPE(487) SQLLEN(3842) /* SCALE=15,PREC=2. 3842=15*256 + 2 */
REDEFINE MY-SQL-AMT
  FLD MY-AMT-NULL-IND C2    /* NULL INDICATOR FOR AMT */
  FLD MY-AMT P8.2           /* PL8.2 IS SCALE 15 (DIGITS), PREC 2 DECIMAL DIGITS */
ENDREDEFINE

CURSOR MYDB2 QUERY(SELECT * FROM MYTABLE WHERE AMOUNT > :MY-SQL-AMT)
...

IF DEPT = 1
   MOVE X'FFFF' TO MY-AMT-NULL-IND /* AMT IS "NULL" */
ELSE
   MOVE X'0000' TO MY-AMT-NULL-IND /* AMT IS NOT "NULL" */
   MOVE DEPT-CUTOFF TO MY-AMT
ENDIF


OPEN MYDB2  /* GET RESULT TABLE FOR CUTOFF AMT, IF IT EXISTS */
...
FETCH MYDB2 /* GET A ROW FROM THE RESULT TABLE */
...
```

Testing for Null Values Returned By a Fetch

Some DB2 columns returned to z/Writer in the result table may contain a special "null" value. When z/Writer receives a null value into a field (during a fetch), it assigns a value of zeros (if numeric) or blanks to that field.

However, a null value is not the same as a valid value of zero (or blanks.) So, if your query returns columns that can contain "null" values, you will probably want to check whether a FETCH has returned an actual value or a "null" value.

z/Writer's IF statement has a special test for this purpose. It's syntax is "IF fieldname NULL". For example:

```
CURSOR MYTABLE QUERY(SELECT AMOUNT, EMPLNUM FROM EMPLTAB)
...
IF AMOUNT NULL
   PRINT 'AMOUNT IS NULL'
ELSE
   PRINT 'AMOUNT IS' AMOUNT
ENDIF
```

Chapter 7. z/Writer's IMS Option

This chapter discusses z/Writer's available IMS Option. Note that it is way beyond the scope of this manual to teach anyone how IMS works. But this chapter will enable experienced IMS programmers to work with their IMS data using z/Writer.

What Is z/Writer's IMS Option?

z/Writer's IMS Option enables z/Writer to access data from any of your shop's IMS databases. With the IMS Option, you can easily report on all of your IMS data using the same powerful, easy z/Writer language that you already know! And z/Writer lets you combine IMS data with other data from flat files and VSAM files -- all in one report.

How to Run z/Writer with IMS

To run z/Writer *without* using IMS data, you execute module ZWRITER in your JCL. When you do want to access IMS with z/Writer, the EXEC statement in your JCL will now execute IBM's program DFSRRC00 (as for other IMS batch jobs). And you will specify ZWIMS (not ZWRITER) as the name of the application program that IMS should execute

You may also need an additional STEPLIB DD in your JCL to identify the library containing your shop's run-time IMS modules (such as ASMTDLI).

No changes are required to your PCBs or PSBs. z/Writer accepts your existing PSBs and PCBs. (PL/1 format PCBs, however, are not supported.)

What Does the IMS Option Do?

The IMS Option makes a new z/Writer statement available for your use. And it adds some new built-in fields and functions that let you access IMS status information after each IMS operation.

New DLI Statement for IMS

- The new DLI statement is used to pass your requests to IMS. These can be requests such as "Get Next" or "Get Unique" or "Insert", etc. Most DLI requests will result in data being returned from IMS to a field in your program, where you can process it as you like.

The DLI statement is the method that you will use to perform all of your work on an IMS database. Note that no FILE statement is necessary for an IMS database.

New Built-In Fields for IMS

A number of new built-in fields become available with the IMS Option (that is, when running ZWIMS rather than ZWRITER). These built-in fields let you examine (but not change) the contents of the PCB used in the most recent DLI statement. (Or, if no DLI statement has yet been executed, the contents of the first PCB passed to z/Writer.) IMS places status and result information in the PCB after each DLI request. The new built-in fields for IMS runs are shown in "New Built-In Fields for IMS Option" (page 178).

The DLI Statement

The DLI statement is the key to accessing and maintaining IMS data from your z/Writer program. It is similar to a CALL statement, in that you specify a variable number of parms to go with it. z/Writer, in fact, then does a CALL to the IMS application program interface, passing these parms on to it. IMS then performs the requested function, usually returning a segment of IMS data to your program. And IMS fills in the PCB with status information to help you evaluate the data returned.

DLI STATEMENT SYNTAX

```
DLI PCB(num-expr)
    FUNC(char-expr)
    IOAREA(fieldname)
    [SSA(char-expr [,char-expr] ...]
```

PARMS

The first two parms (PCB and FUNC) are always required. The second parm, IOAREA, is also required in most cases (with most "functions"). The SSA parm may or may not be required, depending on the function you specify.

PCB(num-expr)

Required. The numeric expression specifies the number of the PCB to use for the request. This is the numerical order of the PCB within the PSB that was passed to z/Writer (by IMS) at startup. The numeric expression can be a literal number, a numeric field, or a numeric expression. Example: PCB(1)

FUNC(char-expr)

Required. This parm provides the 4-byte IMS function code to be performed. It can be a 4-byte character literal, a character field, or a character expression. Example: FUNC('GN ')

IOAREA(fieldname)

Required (except for functions such as CKPT, etc.) This parm tells z/Writer where to store the segment returned by IMS (or where data that is to be passed to IMS is located.) It can be the name of any field in a workarea or in a file. Example: IOAREA(WORKROOT)

SSA(char-expr [,char-expr] ...)

Optional. Use this parm if the IMS function you specify requires "segment search arguments". Some functions (such as Get Unique, 'GU ') require the use of one or more SSA's to identify the segment to be returned. The character parms should be formatted as valid SSA's in IMS's required format. You may specify up to 15 SSA's in the single SSA parm. Each parm be a character literal, a character field or a character expression. Separate them with blanks and/or a comma.

Example: SSA('PARTROOT(PARTKEY EQ 02)', 'STOKSTAK ')

New Built-In Fields for IMS Option

Built-In Fields for IMS Runs		
Name	**Description**	**Data Type**
#PCB	This field contains the entire PCB as one character fieild. It does not include the variable-length feedback key appended to it.	36-byte character
#PCB_DBD_NAME	Contains the name of the IMS database accessed.	8-byte character
#PCB_LEVEL_NUM	Contains the level of the segment returned (when applicable).	2-byte character
#PCB_STATUS	Contains the status code for the last DLI statement. A value of blanks always indicates a successful operation.	8-byte character
#PCB_OPTIONS	Contains the options in the PCB.	4-byte character
#PCB_FDBK_SEGMENT	Contains the name of the segment just returned (when applicable)	8-byte character
#PCB_FDBK_KEY_LEN	Contains the length of the feedback key returned for the last request (when applicable)	4-byte binary number
#PCB_NUM_SEGS	Contains the number of sensitive segments for this PCB.	4-byte binary number

In addition to these built-in fields, there is one IMS-related built-in *function*. It is very similar to a built-in field, except that you may specify a length argument with it. The new function is:

Built-In Function for IMS Runs		
Name	**Description**	**Data Type**
#PCB_FDBK_KEY[(nn)]	Returns the feedback key from the last IMS request as a character field of length “nn”. Note that the feedback keys that IMS return are not always of the same length. The key length for a given request depends on the segment path used for that request. Be sure that your “nn” is large enough to hold the largest possible feedback key for your database. If the actual feedback key is larger than “nn”, the key value will be truncated to that length. If the actual feedback key is smaller than the “nn” value, the key valule is right-padded with blanks. When the optional “nn” argument is omitted, the feedback key is returned as a 100-byte character field.	Character

EXAMPLES

The examples that follow use a sample “Parts” IMS database that IBM provides at installation. It has this heirarchy:

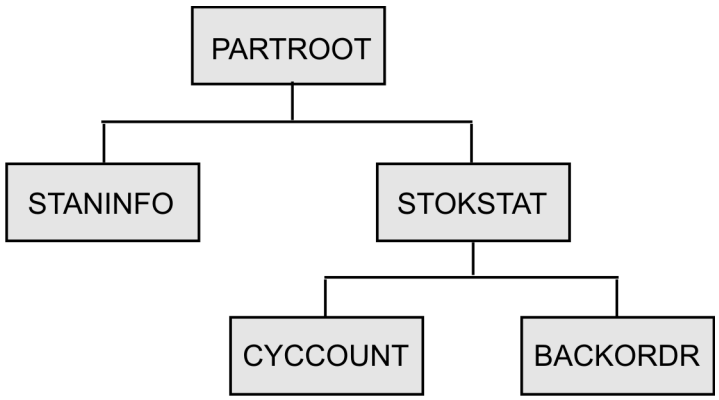


Figure 17 shows a very simple report from this IMS database. It reads all of the root segments (only) from the database. It prints a list of part numbers and descriptions from those segments.

Figure 18 (page 181) shows model code that could be used to read sequentially through *all* of the segments of the Parts database, in their heirarchical order.

These Control Statements:

```
WORKAREA
PARTROOT 60
PART-NUMBER 17 COL(1)
PART-DESC   24 COL(27)

DOUNTIL #PCB_STATUS <> '  '
   DLI PCB(1) FUNC('GN  ') IOAREA(PARTROOT) SSA('PARTROOT ')

   IF #PCB_STATUS = '  '
      PRINT PART-NUMBER PART-DESC('DESCRIPTION')
  ENDIF
ENDDO

TITLE 'LIST OF PARTS IN DI21PART DATABASE'
```

Produce this Report:

```
03/09/16  LIST OF PARTS IN DI21PART DATABASE  PAGE    1

      PART
     NUMBER        DESCRIPTION

02AN960C10         WASHER
02CK05CW181K       CAPACITOR
02CSR13G104KL      KR1J50KS
02JAN1N976B        DIODE CODE-A
02MS16995-28       SCREW
02N51P3003F000     SCREW
02RC07GF273J       RESISTOR
02106B1293P009     RESISTOR
02250236-001       CAPACITOR
02250239           TRANSISTOR
02250241-001       CONNECTOR
02250794           RESISTOR
02250796           SWITCH
02250891           SERVO VALVE
02252252-003       COUPLING
023003802          CHASSIS

(additional lines not shown)
```

Figure 17. A Report from the Root Segments of an IMS Parts Database

```
WORKAREA
WORKSEG 160
PARTROOT 60
STANINFO 85
STOKSTAT 160
CYCCOUNT 25
BACKORDR 75

DOUNTIL #PCB_STATUS = 'GE', 'GB'
   DLI PCB(1) FUNC('GN  ') IOAREA(WORKSEG)

   CASE #PCB_FDBK_SEGMENT
    WHEN 'PARTROOT'
       MOVE WORKSEG TO PARTROOT
       /* PUT LOGIC HERE THAT ONLY NEEDS PARTROOT */

    WHEN 'STANINFO'
       MOVE WORKSEG TO STANINFO
       /* PUT LOGIC HERE THAT ONLY USES PARTROOT + STANINFO*/

    WHEN 'STOKSTAT'
       MOVE WORKSEG TO STOKSTAT
       /* PUT LOGIC HERE THAT ONLY USES PARTROOT + STOKSTAT*/

    WHEN 'CYCCOUNT'
       MOVE WORKSEG TO CYCCOUNT
       /* LOGIC HERE CAN USE PARTROOT, STOKSTAT, CYCCOUNT*/

    WHEN 'BACKORDR'
       MOVE WORKSEG TO BACKORDR
       /* LOGIC HERE CAN USE PARTROOT, STOKSTAT, BACKORDR*/

   ENDCASE
ENDDO
```

Figure 18. Model Code that Sequentially Reads All Segments of an IMS Database

Appendix A. Built-In Fields

z/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 z/Writer built–in fields in detail. Unless otherwise noted, all of these fields are read-only — they may not be modified by the program.

z/Writer Built-In Fields	
Field Name	**Description**
Character Built-In Fields	
#JOBNAME	An 8-byte character field containing the jobname under which z/Writer is currently executing.
#DATE	An 8-byte character field containing the system date (when program began execution) in MM/DD/YY format.
#DAYNAME	A 9-byte character field containing the current day of the week ("MONDAY")
#TIME **#TIME12**	An 8-byte character field containing the system time (when program began execution) in AM/PM format (ex: "12:45 PM").
#TIME24	An 8-byte character field containing the system time (when program began execution) in 24–hour format (ex: "13:45:59").
Numeric Built-In Fields	
#PAGENUM **#PAGE**	The current page number of the report. This field may be modified by the user program. **Note:** This field is normally used in a TITLE statement to position the report's page number. You may also reference it in other statements (for example, an IF or a COMPUTE statement), but only after at least one REPORT, PRINT or TITLE statement has been processed (for a given report.) The #PAGE field for a report output is only created when one of those statements is encountered. **Note:** When referenced somewhere other than a TITLE statement, you will need to qualify the fieldname (#PAGE) with a report name if more than one reports are defined (for that phase). For example: `ZWOUT001.#PAGE = 1`
#RETCODE	The value of this field is returned to the z/OS operating system at the end of z/Writer's execution. z/Writer raises the value of this field when certain errors occur. This field may be modified by the user program.

z/Writer Built-In Fields (Continued)	
Field Name	**Description**
#TALLY	This field is intended for use only within code that is executed at control break time (including at the grand total pseudo control break.) That is, within the code in the paragraphs specified by the BREAKCODE parm of a BREAK statement. During break processing, #TALLY holds the number of records that make up the control group that has just ended. At grand totals time, it holds the total number of records processed for the report. When referenced in open code (not during control break processing) this field has a value of 0. **Note:** if you need a running count of the number of records that have been read, use the #COUNT built-in field associated with that file. You may reference that field at any time in your program.
For Built-In Fields Associated with Files -- see page 79	
For Built-In Fields Associated with Tables -- see page 145	
For Built-In Fields Associated with IMS PCBs -- see page 178	

Appendix B. 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 or date).

The arguments to a function will not necessarily be of the same data type as the result. The data type expected for each argument is indicated in a function's syntax. For example, "char" means that a character argument is expected. Except where otherwise indicated, an argument may be any of the following:

- a literal value
- the name of a field from any file, table or workarea
- a computational expression (which may itself involve other built–in functions)

Date arguments must:

- be a character (C) or character-numeric (N) field, or a character literal enclosed in ticks
- be either 6 or 8 bytes long
- contain a valid date in YYMMDD format (if 6 bytes long) or in YYYYMMDD format (if 8 bytes)
- 6-byte date arguments are converted internally to 8-byte dates by assigning a century based on the century cutoff value. (See page 106.)
- when a date argument is optional (enclosed in brackets), the system date will be used if the argument is not present

Note: Date results are returned as 8-byte character values, with an implicit CONTAINS(YYYYMMDD) "property". (See page page 73.) Date functions are supported for dates in the years 1601-9999.

Separate the arguments with blanks and/or commas.

The following table lists the z/Writer built–in functions. After the table, each of the functions is discussed in more detail.

z/Writer Built-In Functions	
Function	**Description**
Functions that Return a Character Value	
#AND	returns the result of AND-ing two character strings
#ASCII	returns the ASCII equivalent of an EBCDIC string
#COMPRESS	concatenates multiple fields and compresses out extra blanks
#DATE2JUL	converts a 6- or 8-byte gregorian date to a 7-byte julian date

z/Writer Built-In Functions (Continued)	
Function	**Description**
#DAY	returns the day of the week (ex: SUNDAY) for a given date
#EBCDIC	returns the EBCDIC equivalent of an ASCII string
#LCASE	returns the lower–case value of a character string
#LEAPYEAR	returns "Y" if the date argument occurs in a leap year; otherwise returns 'N'
#LEFT	returns the leftmost *n* characters of a character string
#MONTH	returns the month name (ex: JANUARY) pertaining to a given date
#OR	returns the result of OR-ing two character strings
#PARSE	returns one individual word parsed out of a character string
#REALDATE	returns "Y" if the date argument is a valid, calendar date; otherwise returns 'N'
#RIGHT	returns the rightmost *n* characters of a character string
#SUBSTR	returns a substring from a character string
#TRANSLATE	translates one set of characters within a character string to another set of characters
#UCASE	returns the upper–case value of a character string
#XOR	returns the result of XOR-ing two character strings
#YEAR	returns the 4–byte year pertaining to a given date
Functions that Return a Numeric Value	
#ABS	returns the absolute value of a number
#CHAR2NUM	converts a character value to a numeric value
#DATE2DIC	converts a date to a numeric "day in century" value
#DAYNUM	returns the day of the month (1–31) for a given date
#DOWNUM	returns a number (1-7) representing the day of the week of a given date
#INDEX	returns the column where a certain substring begins within a larger string
#INT	returns the integer portion of a number (no rounding performed)
#MAX	returns the greater of two or more values
#MIN	returns the smaller of two or more values
#MOD	returns the remainder left after division ("modulus")
#MONTHNUM	returns the month number (1–12) for a given date
#NUMWORDS	returns the number of words within a character string
#ROUND	returns the rounded value of a number
#SQRT	returns the square root of a number

z/Writer Built-In Functions (Continued)	
Function	**Description**
#YEARNUM	returns the 4–digit year for a given date
Functions that Return a 8-Byte Character Date Value	
#BEGMONTH	returns the first day of the month in which a date occurs
#BEGWEEK	returns the first day of the week in which a date occurs
#BEGYEAR	returns the first day of the year in which a date occurs
#DIC2DATE	converts a numeric “day in century” value to a date
#DMY	creates a date from three numeric parms representing day, month and year
#ENDMONTH	returns the last day of the month in which a date occurs
#ENDWEEK	returns the last day of the week in which a date occurs
#ENDYEAR	returns the last day of the year in which a date occurs
#INCDATE	increments a date by a number of days, weeks, months or years
#JUL2DATE	converts a 5- or 7-byte julian date to an 8-byte gregorian date
#MDY	creates a date from three numeric parms representing month, day and year
#YMD	creates a date from three numeric parms representing year, month and day
For Built-In Functions Associated with IMS PCBs -- see page 178	

Functions that Return a Character Value

#AND(char1,char2)

Performs the logical AND operation on the two character arguments and returns the result. (An AND operation results in a 1 bit if the corresponding bit of *both* operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right-padded with hex zeros before performing the AND operation. The size of the result is the size of the larger operand.

Example: `COMPUTE A = #AND(X'01FF',X'035E')` results in `A=X'015E'`

Here is an example of using the #AND built–in function to change a packed numeric field’s sign from the common, but non-standard, F to the standard C. For example, assume that PACKED-NUMBER is a 5-byte packed field that has an F in the zone portion of its last byte (X’000000123F’)

Example: `PACKED–NUMBER = #AND(PACKED-NUMBER,X'FFFFFFFFFC'`
results in `PACKED-NUMBER = X'000000123C'`

#ASCII(char)

Returns the ASCII equivalent of the EBCDIC character argument. The size of the value returned by this function is the size of the character argument.

```
Example:  A = #ASCII(X'F1F2F3') results in  A=X'313233'
```

Note: The three characters "123" are represented with X'F1F2F3' in EBCDIC and with X'313233' in ASCII.

#COMPRESS([n,] char [,n] ,char ...)

Concatenates any number of char arguments, compressing out all but 1 of the blanks between each argument The optional override number "n" specifies how many blanks to leave (or add) between the two char arguments (if a number other than 1 is desired). You may specify 0 if no blanks are wanted between two arguments. The size of the returned string is the sum of the sizes of all arguments, plus spacing bytes.

```
Example: COMPUTE NAME=#COMPRESS(LAST-NAME, 0, "," , FIRST-NAME)
               might result in NAME="BAKER, VIVIAN                "

Example:       COMPUTE ADDR=#COMPRESS(CITY, 0, ",", STATE ZIP-CODE)
             might result in ADDR="DALLAS, TX 75230             "
```

Note: The #COMPRESS function does not remove any *leading* blanks that might be in the character arguments.

#DATE2JUL[(date)]

Returns the 7-byte julian date (in YYYYDDD format) corresponding to the 6- or 8-byte gregorian argument date.

```
Example: COMPUTE A = #DATE2JUL('20141231')  would result in  A="2014365"
```

#DAY[(date)]

Returns the day of the week pertaining to the date argument, as a 9–byte character field.

```
Example: COMPUTE A = #DAY(HIRE-DATE)  might result in  A="TUESDAY  "
```

#EBCDIC(char)

Returns the EBCDIC equivalent of the ASCII character argument. The size of the value returned by this function is the size of the character argument.

```
Example: COMPUTE A = #EBCDIC(X'313233')  results in  A=X'F1F2F3'
```

Note: The three characters "123" are represented with X'F1F2F3' in EBCDIC and with X'313233' in ASCII.

#LCASE(char)

Returns the character argument's value after translating any of its upper–case alphabetic characters to the corresponding lower–case character. All other printable characters remain unchanged. (The effect of this function on non-printable characters is not defined.) The size of the value returned by this function is the size of the character argument.

```
Example: (Assume that DESC = "THIS IS A DESCRIPTION")
          COMPUTE A = #LCASE(DESC)  results in  A="this is a description".
```

#LEAPYEAR[(date)]

Returns 'Y' if the year portion of the argument date is a leap year. Otherwise, it returns 'N'. (The month and day portions of the argument date are not examined for validity.)

Example: `COMPUTE A=#LEAPYEAR('20070515')` results in `A='N'`

Example: `COMPUTE B=#LEAPYEAR('20080515')` results in `B='Y'`

#LEFT(char,num1)

Returns a substring of the char argument, starting with the first column, for a length of "num1" bytes. Num1 may be either a literal value or a numeric expression. When num1 is a literal value, the size of the value returned by this function is num1. When num1 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

Example: `COMPUTE A = #LEFT('ABCDEFG',4)` results in `A='ABCD'`

#MONTH[(date)]

Returns the name of the month pertaining to the date argument, as a 9–byte character field.

Example: `COMPUTE A = #MONTH(HIRE–DATE)` might result in `A="MARCH    "`

#OR(char1,char2)

Performs the logical OR operation on the two character arguments and returns the result. (An OR operation results in a 1 bit if the corresponding bit of *either (or both)* operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right–padded with hex zeros before performing the OR operation. The size of the result is the size of the larger operand.

Example: `COMPUTE A = #OR(X'8024',X'0756')` results in `A=X'8776'`

Note: you can use the #OR function to create packed numeric fields that have a sign of F (rather than the standard sign of C). For example, assume that PACKED-NUMBER has a value of X'00123C'.

Example: `COMPUTE PACKED-F = #OR(PACKED-NUMBER,X'00000F')` results in `PACKED–F = X'00123F'`

#PARSE(char,num)

Returns a single word parsed from the character argument. Internally, the character argument is first parsed into individual words, each delimited by one or more spaces. The numeric argument specifies which of the parsed words should be returned by the function. A numeric argument of 1 indicates that the first word should be returned; an argument of 2 means return the second word, etc. Negative numbers may also be used. A negative number indicates the word to return counting backwards from the last word parsed. A numeric argument of –1 means return the last word parsed; an argument of –2 means return the second to last word, etc. If the word indicated by the numeric argument doesn't exist, blanks are returned by this function. The size of the value returned by this function is the size of the whole character argument.

Note: you can use the related #NUMWORDS built–in function to find out how many words a character string contains.

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)

```
COMPUTE A = #PARSE(DESC,1)  results in   A="THIS            "
COMPUTE A = #PARSE(DESC,2)  results in   A="IS              "
```

```
COMPUTE A = #PARSE(DESC,–1) results in   A="DESCRIPTION      "
COMPUTE A = #PARSE(DESC,5) results in    A="                 "
```

Note: To parse a text using a delimiter other than blanks, try using the #TRANSLATE built-in function to first translate the desired delimiter characters into blanks. For example, you could parse an IP address (which is delimited with dots) this way:

```
Example: COMPUTE A = #PARSE(#TRANSLATE(IPADDR,"."," "),2)
```

Assume that IPADDR = "12.345.67.8". The above statement results in A = "345 "

Note that using #TRANSLATE with #PARSE may not work if the original string contains *multiple consecutive* delimiters.

#REALDATE(date)

Returns 'Y' if the date argument contains a real calendar date. Otherwise, it returns 'N'.

Example: `COMPUTE A=#REALDATE(6/30/2007)` results in A being "true"

Example: `COMPUTE B=#REALDATE(6/31/2007)` results in B being "false"

Example: `COMPUTE C=#REALDATE(2/29/2007)` results in C being "false"

Example: `COMPUTE D=#REALDATE(99/99/9999)` results in D being "false"

#RIGHT(char,num1)

Returns a substring of the char argument consisting of the last "num1" bytes. Num1 may be either a literal value or a numeric expression. When num1 is a literal value, the size of the value returned by this function is num1. When num1 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

Example: `COMPUTE A = #RIGHT('ABCDEFG',4)` results in `A='DEFG'`

#SUBSTR(char,num1,num2)

Returns a substring of the char argument, starting at column "num1" for a length of "num2" bytes. (The first byte in a string is column 1.) Num1 and num2 may be literal values or numeric expressions. When num2 is a literal value, the size of the value returned by this function is num2. When num2 is an expression, the size returned by this function is the size of the character argument (since that is the maximum possible size of the result).

Example: `COMPUTE A = #SUBSTR('ABCDEFG',2,3)` results in `A='BCD'`

#TRANSLATE(char1,char2,char3)

Returns the char1 string after translating any of its characters found in the char2 argument into the corresponding character of the char3 argument. (Translated characters in the char1 argument are *not* then re–evaluated for additional translation.) The size of the value returned by this function is the size of the char1 argument.

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)
`COMPUTE A = #TRANSLATE(DESC,"TA","XY")` would result in
`A="XHIS IS Y DESCRIPXION".`

Note: the char2 and char3 arguments must be character or hex literals.

#UCASE(char)

Returns the character argument's value after translating any of its lower–case alphabetic characters to the corresponding upper–case character. All other printable characters remain unchanged. (The effect of this function on non-printable characters is not defined.) The size of the value returned by this function is the size of the character argument.

Example: (Assume that `NAME = "Smith "`)
`COMPUTE SORT–NAME = #UCASE(NAME)` results in `SORT–NAME = "SMITH "`

#XOR(char1,char2)

Performs the logical XOR operation on the two character arguments and returns the result. (An XOR operation results in a 1 bit if the corresponding bit of either (but *not* both) operands is a 1: otherwise it results in a 0 bit.) If the two operands are not the same size, the shorter operand will be right-padded with hex zeros before performing the XOR operation. The size of the result is the size of the larger operand.

Example: `COMPUTE A = #XOR(X'5766',X'5744')` results in `A=X'0022'`

#YEAR[(date)]

Returns the year portion of the date argument as a 4–byte character field.

Example: `COMPUTE A = #YEAR(HIRE–DATE)` might result in `A="2001"`

Functions that Return a Numeric Value

#ABS(num)

Returns the absolute value of the numeric argument.

Example: `COMPUTE A = #ABS(–4)` results in `A= 4`

#CHAR2NUM(char)

Converts a string of numeric characters into a numeric value. Standard punctuation commonly used with numbers (such as ‘ $1,234.50 ‘) is allowed in the character argument. However, it must not contain any alphabetic characters (other than a final digit that is signed with X'C' or X'D', such as X'C1'). An all–blank string returns the value zero.

Example: `COMPUTE A = #MAKENUM('  125')` results in `A=125`

#DATE2DIC(date)

Converts the date argument into a numeric “day in century” value. January 1, 1900 corresponds to day 1, and December 31, 2099 is day 73,049. For dates before 1900, the day in century will be a negative value.

Example: `COMPUTE A = #DATE2DIC('20071231')` results in `A=39446`

Example: `COMPUTE A = #DATE2DIC('20080101')` results in `A=39447`

Example: (of computing the number of days between two dates):
`COMPUTE NUM–DAYS = #DATE2DIC(END–DATE) – #DATE2DIC(START–DATE)`

If END–DATE is 4/2/2007 and START–DATE is 3/28/2007, then the above example would result in NUM–DAYS = 5.

Note: You can use the #DIC2DATE function to convert a numeric day in century back into a date.

#DAYNUM[(date)]

Returns the numeric value of the day portion of the date argument.

Example: `COMPUTE A = #DAYNUM('20070331')` results in `A=31`

#DOWNUM[(date)]

Returns a number from 1 to 7 representing the day of the week of the argument date. (1 means Sunday, 2 means Monday, ... 7 means Saturday.)

Example: `COMPUTE A = #DOWNUM('20070331')` results in `A=7`

#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. (The first byte in a string is byte 1.) If char1 does not contain char2, #INDEX returns zero.

Example: `COMPUTE A = #INDEX('ABCDEF', 'CDE')` results in `A=3`

#INT(num)

Returns the integer portion of the numeric argument. The argument's decimal digits, if any, are simply dropped, regardless of the sign of the argument.

Example: `COMPUTE A = #INT(12.987)` results in `A= 12`
`COMPUTE A = #INT(–12.987)` results in `A= –12`

#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`

#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`

#MOD(num1,num2)

Returns the remainder left when the first argument is divided by the second argument.

Example: Example: `COMPUTE A = #MOD(45, 4)` results in `A= 1`
`COMPUTE A = #MOD(–45, 4)` results in `A= –1`
`COMPUTE A = #MOD(1.5, .2)` results in `A= .1`

#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.

Example: (Assume that `DESC = "THIS IS A DESCRIPTION"`)
`COMPUTE A = #NUMWORDS(DESC)` results in `A = 4.`

Note: This function may be useful when you want to assign a value to a computed field differently depending on how many, if any, words are in some other field. For example, the following example assigns the second word from the DESC field to the result. However, if the DESC field contains only 1 (or no) words, the text "*NONE*" is assigned instead:

```
Example: COMPUTE A = WHEN(#NUMWORDS(DESC) >= 2) ASSIGN(#PARSE(DESC,2))
                     ELSE                       ASSIGN("*NONE*")
```

#ROUND(num1,num2)

Returns the first numeric argument, rounded to the precision specified by the second numeric argument. Num2 is the number of decimal places that num1 should be rounded to. Rounding of negative numbers is performed as if they were positive. Num2 must be a literal integer (not an expression). The *number* of decimal digits returned by this function is the same as the number of decimal digits in the num1 argument.

Num2 can also be a **negative** number. Use this feature to round to a digit position on the *left* side of the decimal point. Use -1 to round to tens, -2 to round to hundreds, and so on.

```
Example: COMPUTE A = #ROUND(12345.678, 2) results in A= 12345.680
         COMPUTE A = #ROUND(–12345.678, 2) results in A=–12345.680
         COMPUTE A = #ROUND(12345.678, 0)  results in A= 12346.000
         COMPUTE A = #ROUND(12345.678, –2) results in  A= 12300.000
```

#SQRT(num)

Returns the square root of the numeric argument.

```
Example: COMPUTE A = #SQRT(2) results in A = 1.41
```

#YEARNUM[(date)]

Returns the 4-digit numeric value of the year portion of the date argument.[Note 1]

```
Example: COMPUTE A = #YEARNUM('20070331') results in A=2007
```

Functions that Return a Date Value

#BEGMONTH[(date)]

Returns the date of the first day of the month in which the date argument occurs.

```
Example: COMPUTE A = #BEGMONTH('20070515') results in A='20070501'
```

#BEGWEEK[(date)]

Returns the Sunday of the calendar week in which the date argument occurs.

```
Example: COMPUTE A = #BEGWEEK('20070515') results in A='20070513'
```

Note: You can also use this function to return *any* particular day of a given week (Monday, Tuesday, etc.). Just use it in combination with an #INCDATE function that adds the appropriate number of days to the result. Add 1 to get Monday, 2 to get Tuesday, and so on. The following example returns the Wednesday of the week that SALES-DATE falls within.

```
Example: WED-SALES-DATE = #INCDATE(#BEGWEEK(SALES-DATE), 3, DAYS)
```

#BEGYEAR[(date)]

Returns the first day of the year in which the date argument occurs.

```
Example: COMPUTE A = #BEGYEAR('20070515')  results in  A='20070101'
```

#DIC2DATE(num)

The numeric argument is treated as a "day in century" value. (For example, the value 1 corresponds to January 1, 1900, and 73,049 corresponds to December 31, 2099. The function returns the date corresponding to the numeric day from the start of the 20th century. Use this function to change the results of the #DATE2DIC function back into a date.

Example: `COMPUTE A = #DIC2DATE(39446)` results in `A='20071231'`

#DMY(num, num, num)

See the description under #YMD below.

#ENDMONTH[(date)]

Returns the last day of the month in which the date argument occurs.

Example: `COMPUTE A = #ENDMONTH('20070515')` results in `A='20070531'`

#ENDWEEK[(date)]

Returns the Saturday of the calendar week in which the date argument occurs.

Example: `COMPUTE A = #ENDWEEK('20070515')` results in `A='20070519'`

#ENDYEAR[(date)]

Returns the last day of the year in which the date argument occurs.

Example: `COMPUTE A = #ENDYEAR('20070515')` results in `A='20071231'`

#INCDATE([date,] number, units)

Returns the date obtained by incrementing the argument date by the given number of units. Units can be any of these keywords or abbreviations:

- DAYS, DAY, D
- WEEKS, WEEK, WKS, WK, W
- MONTHS, MONTH, MONS, MON, M
- YEARS, YEAR, YRS, YR, Y

Example: `COMPUTE A = #INCDATE('20070515', 3, WEEKS)` results in `A = '20070605'`
`COMPUTE YESTERDAY = #INCDATE( -1, DAYS)` results in `YESTERDAY` being the date before the system date.

Note: When incrementing by months or years, the day portion of the resulting date is sometimes changed to the last day of the month, in order to return a valid calendar date.

Example: `COMPUTE A = #INCDATE('20070531', 1, MONTH)` results in `A = '200705630'` (not '20070631', which is not a valid date)
`COMPUTE B = #INCDATE('20080229', 1, YEAR)` results in `B = '20090228'` (not '20090229', which is not a valid date)

#JUL2DATE[(char)]

Returns the 8-byte gregorian date corresponding to the 5- or 7-byte julian argument date.

Example: `COMPUTE A = #DATE2JUL2('14365')` would result in `A="20141231"`

#MDY(num, num, num)

See the description under #YMD below.

#YMD(num, num, num) and #MDY(num, num, num) and #DMY(num, num, num)

Returns a date value based on the three numeric arguments (representing month, day and year in the order indicated by the function name.) The resulting date is not validity-checked to see if it is an actual calendar date. (You can use the #REALDATE function to find this out.) The numeric argument representing the year can be any 1 to 4 digit number, and the month and day arguments can be any 1 or 2 digit number.

Example: `COMPUTE A = #MDY(12,31,2007)` results in `A='20071231'`
`COMPUTE B = #YMD(9999,99,99)` results in `B='99999999'`

Appendix C. Syntax of PICTURE Display Formats

A PICTURE is a special display format that describes how a numeric value should be displayed in a report. The PICTURE display format consists of the word PICTURE (or an abbreviation, such as PIC) immediately followed by text enclosed in either apostrophes or quotation marks. (Do not put a space before the apostrophe or quotation mark.) For example:

```
PICTURE'text'
PIC'text'
```

The characters making up the text give a "picture" of how the formatted result should look. The PICTURE specifies such thing as:

- the **size** of the formatted output (that is, how many characters it will occupy in a print line)
- whether **leading zeros** should be displayed or suppressed
- whether **commas** (or some other character) will be used to separate the thousands, the millions, etc.
- whether a **floating dollar sign** should appear in the result
- where the **minus sign** should appear, for negative numbers
- where (and whether) a **plus sign** should be displayed for positive numbers
- how many **decimal digits** should print
- any **literal characters** that should be included in the formatted result
- whether **automatic scaling** of the number is wanted (to allow a wide range of values to fit into a small column)

Examples of PICTUREs

z/Writer's PICTUREs are very similar to COBOL's PICTURE clause, in case you are familiar with those. 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 (Z and 9) in your PICTURE according to the size of the numbers that you will be printing.

In the table below, a sample positive value (1,234.56) and a sample negative value (-98,765.4) are used to demonstrate each PICTURE.

EXAMPLES OF PICTURES		
PICTURE	FORMATTED POSITIVE VALUE	FORMATTED NEGATIVE VALUE
PIC'999999999'	000001235	****S****
PIC'999999.9'	001234.6	****S***
PIC'999999.99'	001234.56	****S****

EXAMPLES OF PICTUREs (CONTINUED)		
PICTURE	FORMATTED POSITIVE VALUE	FORMATTED NEGATIVE VALUE
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 EURO'	1.234,56 EURO	–98.765,40 EURO
PIC'ZZZZZ9.99%'	1234.56%	–98765.40%

Note: The first several examples above resulted in size error indicators (***S***) for the negative value. That is because the PICTURE did not have a place where the minus sign could be displayed. Since leading zero suppression was *not* used, there were no leading blanks in which to place a minus sign. If your numbers will include negative values, do *not* use all 9's in your PICTURE. Add at least one leading Z or – to the PICTURE.

Below are two additional examples that illustrate special purpose PICTUREs. Notice that when literal text is used heavily, you should normally use "9" as your digit symbol. If you want to display a literal character *before* the first numeric digit (as in the telephone number example below), you *must* use "9" for all of your digit symbols
.

ADDITIONAL PICTURE EXAMPLES		
PICTURE	UNFORMATTED VALUE	FORMATTED VALUE
PIC'(999) 999–9999'	1234567890	(123) 456–7890
PIC'999–99–9999'	123456789	123–45–6789

PICTUREs can be used anywhere that a numeric display format is allowed. Following are a few examples of how PICTUREs can be used in various control statements:

```
FLD  AMOUNT P5.2 FORMAT(PIC'$$$,$$9')
PRINT EMPL-NAME  TOTAL-SALES(PIC'ZZZ,ZZZ,ZZ9.99-')
TITLE 'TELEPHONE DIRECTORY --'  TELEPHONE(PIC'(999) 999-9999')
```

How PICTUREs Work

This section explains in more detail exactly how PICTUREs are processed.

When a numeric value is being formatted according to a PICTURE, the following process takes place. The PICTURE is evaluated one character at a time, from left to right. Each character in the PICTURE is either:

- a symbol that represents one potential *digit* of the numeric value
- a *literal character* that, under certain conditions, will be moved into the result

The **character 9** in a PICTURE always represents a *digit* from the numeric value. It will be replaced by the appropriate digit of the number, *even if that digit is a leading zero*.

If you want to suppress leading zeros in your result, use one of the following characters to represent leading digits in your PICTURE: Z, $, + or –. When one of these characters appears in the PICTURE before the first 9, that character becomes the **leading zero suppression symbol** for the PICTURE. Each occurrence of that symbol will be replaced by the appropriate digit of the number *as long as that digit is not a leading zero*. If the digit is a leading zero, then a blank will appear in that position of the result.

Use the **$ character** for the leading digits in your PICTURE if you want a floating dollar sign to be placed just before the first significant digit in the result. (You can use the CURRCHAR parm in the REPORT statement to choose a different currency symbol to appear in your result. But you will still use **$** in your PICTURE.)

Use the **+ character** for the leading digits in your PICTURE if you want a floating sign to be placed just before the first significant digit in the result. A plus sign is used for positive numbers; a minus sign is used for negative numbers; no sign is used if the number is zero.

Use the **– character** for the leading digits in your PICTURE if you want a floating minus sign to be placed just before the first significant digit in the result (for negative values). Positive and zero values will have no sign character.

When the **letter Z** is used for the leading digits in your PICTURE, *and no trailing sign symbol appears in the* PICTURE, a floating minus sign is placed before the first significant digit in the result (for negative values).

Use a **+ character** as the *last byte* in your PICTURE if you want a trailing sign (either plus or minus) to be placed in that position of the result.

Use a **– character** as the *last byte* in your PICTURE if you only want a trailing minus sign to be placed in that position of the result (for negative values).

The **letter V** has a special meaning within a PICTURE. It shows where an "understood decimal point" is located. A PICTURE may contain only one V symbol. The V symbol does not take up a byte in the formatted output. (Thus, the result of PIC'99V9' would be just 3 bytes long, not 4.) If a V is present in the PICTURE, all decimal points (.) in the PICTURE are treated as literals and are not used in determining where the decimal digits appear in the result.

The **decimal point (.)** is treated specially within a PICTURE. If the PICTURE contains a V symbol, all decimal points within the PICTURE are just treated as literals. (Thus, the two decimal points in PIC'ZZZ.ZZZ.ZZ9V9' are treated as regular literals.) If no V symbol appears within the PICTURE, a single

decimal point is allowed within the PICTURE. It shows where an "explicit decimal point" is to be located in the result.

The **number sign (#)** and **the at sign (@)** are used in scaled pictures (page 200) to show where to put the abbreviation for the scale used (K, M, G, etc.).

All other characters are treated as *literals*. Literals are moved into the result just as they appear in the PICTURE, with one exception. Any literal that appears *before* the last zero suppression symbol in a PICTURE is blanked out if zero suppression is still in effect at that point. Such literals are only moved to the result if one or more non–zero digits have already been moved to the result. (Thus, the comma literals in PIC'ZZZ,ZZZ,ZZ9.99' are blanked out until after the first digit appears in the result.) Also, *trailing literals* are always moved to the result (even if no digits were moved.) Trailing literals are those that appear after all of the numeric positions in a PICTURE. They are usually currency indicators (PIC'ZZ9.99 USD') or percentage signs (PIC'ZZ9.9%').

Exception: in PICTUREs with *no* zero suppression symbols (such as PIC'(999) 999–9999'), all literals are moved to the result.

The following table summarizes the meaning of each character that can appear in a PICTURE.

Note: A PICTURE may contain symbols representing no more than 31 digits. However, the entire PICTURE text (including literal characters) can be larger than 31 characters.

.

MEANING OF SYMBOLS WITHIN A PICTURE	
SYMBOL	**MEANING**
9	Replace this character with a digit from the numeric value, even if that digit is a leading zero.
Z	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a minus sign for negative numbers (unless the PICTURE contains an explicit trailing plus or minus sign).
$	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a dollar sign. For negative numbers, a minus sign will appear just before the floating dollar sign (unless the PICTURE contains an explicit trailing plus or minus sign).
–	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain a minus sign for negative numbers.

MEANING OF SYMBOLS WITHIN A PICTURE (CONTINUED)	
SYMBOL	**MEANING**
+	*When used as the leading zero suppression symbol.*(1) Replace this character with a digit from the numeric value, with the following exception: leading zeros will appear as blanks. The position before the first non–suppressed digit will contain: a plus sign for positive numbers; a minus sign for negative numbers; a blank if the number is zero.
–	*Minus sign, as the last character in a picture.* Specifies that a minus sign should appear in that position if the number is negative. Otherwise, a blank will appear in that position.
+	*Plus sign, as the last character in a picture.* Specifies that: a plus sign should appear in that position if the number is positive; a minus sign should appear in that position if the number is negative; a blank should appear in that position if the number is zero.
V	*Understood decimal point.* This character indicates where the understood decimal point exists within a picture. However, no actual decimal point will appear there. This PICTURE symbol does not affect the size of the formatted result. When this symbol is used, any decimal points (.) in the PICTURE are treated as literals.
.	*When used as an explicit decimal point.* When a PICTURE does not contain a V, this becomes the explicit decimal point. It is displayed as is, unless "leading zero suppression" is still in effect. In that case, a blank will appear in its place.
#	Indicates that the numeric value should be scaled as necessary to fit within the PICTURE. Base-10 scaling (division by factors of 1000) is desired. The "@" symbol also indicates where to put the scale abbreviation (K, M, G, etc.).
@	Indicates that the numeric value should be scaled as necessary to fit within the PICTURE. Base-2 scaling (division by factors of 1024) is desired. The "?" symbol also indicates where to put the scale abbreviation (K, M, G, etc.).
other	Any characters other than those listed above are considered *literal characters* within a picture. These characters will appear in the formatted result just as they are, unless "leading zero suppression" is still in effect. In that case, blanks will appear in their place. Trailing literals (any literal after the last digit position) are always formatted into the result.

Notes:
(1) the first Z, $, + or – character that appears in a picture becomes the "zero suppression symbol" for that picture. Once the zero suppression symbol has been determined for a picture, the other three characters in that set are just treated as literals.

Scaling Numbers with PICTUREs

z/Writer PICTUREs also have a unique “scaling” option that you won’t find in other languages. Such PICTURES automatically scale the number being formatted. Scaling means to round the number to thousands, millions, etc. as necessary to make it fit within the PICTURE. The appropriate abbreviation (K, M, G, etc.) indicates what scale the number is shown in.

Scaled PICTUREs allow you to use less space in a report line while still showing approximate values for very large numbers. Look at these two columns of data:

```
                  SALES          SALES
                             26      26
                         48,712      49 K
                  5,862,131,092  5,862 M
```

The first column, while showing the exact value of each number, uses up 13 bytes of the report line (even more if you have to allow room for totals). The second column shows scaled values for the same numbers and only uses 7 bytes. (And the total would also fit in 7 bytes.)

Here is the PRINT statement used to format the above columns:

```
Example: PRINT SALES(13) SALES(PIC'Z,ZZ9 #')
```

The "#" in the PICTURE indicates that base-10 scaling (division by factors of 1000) is wanted for that column. (Base-10 scaling is normally used with business and financial data.) The "#" symbol also indicates just where to place the scale abbreviation (K, M, G, etc.).

Scaled PICTUREs can also include decimal digits, if you like:

```
Example:  PRINT  FILESIZE(13) FILESIZE(PIC’ZZ9.9 #’)
```

The above statements result in the following columns.

```
                  SALES          SALES
                             26     26.0
                         48,712     48.7 K
                  5,862,131,092      5.9 M
```

You can also request base-2 scaling (division by factors of 1024, or 2 to the 10th power). This type of scaling is often used with data related to computer systems. To specify base-2 scaling, use the "@" scaling symbol instead of "#". If you also add a literal "B" to the PICTURE, you will end up with the abbreviations KB, MB, GB, etc. in the column.

```
Example: PRINT SALES(13) SALES(PIC'Z,ZZ9 @B')
```

The above statements result in the following columns.

```
                 FILESIZE    FILESIZE
                       26      26  B
                   48,712      48 KB
             5,862,131,092   5,591 MB
```

Note: If the field you are scaling can contain **negative values**, *be sure* to begin the PICTURE with a minus sign, a space or an "extra" Z. If you fail to this, you won't get a size error (***S***) as with regular PICTUREs. But the negative number will have to be scaled down to a potentially misleading degree. Sometimes all the way down to 0 (on some scale.)

Take, for example, this PICTURE which has no extra byte for a minus sign: PIC'ZZ9#. The number 100,000,000 would format normally as "100M". But the number –100,000,000 appears, surprisingly (at first glance), as " 0G". z/Writer can't show "–100M" in the 4-byte PICTURE. So it has to scale the number down further to –0.1 billion. Rounding that to a whole number (to match the PICTURE) gives 0 billion. That does fit in the PICTURE (" 0G") but is not very useful and could be misleading. Using the correct PIC'–ZZ9#' would give the results you expect for both positive and negative numbers: " 100M" and "–100M".

Note: in most cases, you will want **at least 3 digit positions** in scaled PICTUREs. Otherwise, you can have a similar problem to the one described above (of having your number rounded down to meaninglessness), even when all values shown will be positive. Take for example, the following PICTURE with only 2 digits positions: PIC'Z9#'. The positive number 100,000,000 can only be shown as " 0G" in this small PICTURE.

Note: if you need to use a *literal* character # or @ in your picture, use the PICBASE2 and PICBASE10 parms in an OPTION statement (to choose different characters to assign this special meaning to.) Then the # and/or @ characters will not have special meaning within PICTUREs and can be used as literals.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z