New Features in Spectrum Writer 3.0.0

The major new features in release 3.0.0 of Spectrum Writer are:

- the ability to **create multiple reports** in a single run (page 1)
- the ability to **normalize records** as a means of working with arrays (page 6)
- the ability to **stop a run early** based on the contents of the input file (page 16)
- the ability to **upgrade the severity of I/O errors** for one or more input files (page 17)

Creating Multiple Reports in a Single Run

This section explains:

- how to create more than one report (or output file) during a single run of Spectrum Writer

One very powerful feature of Spectrum Writer is its ability to produce multiple reports during a single pass of the input file. This can greatly reduce the total amount of I/O processing (compared to producing the same reports in separate Spectrum Writer runs).

In multiple report runs, you have complete control over each individual report. For example, each report can have its own:

- criteria for including records from the input file (INCLUDEIF statement)
- sort order (SORT statement)
- control breaks (BREAK statements)
- data columns, column headings, format, etc. (COLUMNS statements)
- titles and/or footnotes (TITLE/FOOTNOTE statements)
- most OPTION statement options that affect a report's appearance

The only things that all reports in a single run are required to have in common are:

- the primary input file (INPUT statement)
- a few OPTION statement options that relate to the processing of the input file or that apply to the run as a whole.

The NEWOUT Control Statement

The NEWOUT (or NEWOUTPUT) statement is used to request an additional report.

Put the NEWOUT statement after all of the control statements used to request the first report. The NEWOUT statement tells Spectrum Writer that you have finished one report request and

are now starting to define a new output (that is, a new report or output file.) The NEWOUT statement has no parms:

```
NEWOUT:
```

After the NEWOUT statement, you are now working on a fresh report. Nothing that you specified for any earlier report(s) is still in effect, except that:

- the primary input file and all of it's fields are available
- any auxiliary input files and all of their fields are available
- all computed fields are available

After the NEWOUT statement, just put the COLUMNS, SORT, TITLE, INCLUDEIF, etc. statements that you want for the new report. The control statements following a NEWOUT statement affect only the new report. They do not affect any earlier report(s).

If needed, you can also add any additional READ statements or COMPUTE statements required for the new report. Or, if you prefer, you can place those READ and COMPUTE statements back with the control statements for the first report. Their location will not affect the results of the run.

Here is an example of producing three reports from a single pass of the SALES-FILE.

```
INPUT: SALES-FILE
TITLE: 'SALES BY REGION'
COLUMNS: REGION CUSTOMER AMOUNT TAX
SORT: REGION
BREAK: REGION

NEWOUT:
TITLE: 'SALES BY CUSTOMER'
COLUMNS: CUSTOMER REGION AMOUNT TAX  EMPL-NAME
SORT: CUSTOMER
BREAK: CUSTOMER

NEWOUT:
TITLE: 'SALES OVER $100 SORTED BY DESCENDING AMOUNT'
INCLUDEIF: AMOUNT > 100
COLUMNS: AMOUNT CUSTOMER
SORT: AMOUNT(DESC)
```

The above statements request three separate reports from the SALES-FILE. (The blank lines before the NEWOUT statements are not required. We added them only to make the request more readable.) As you can see, each report has it's own title, sort order and columns. The first report is sorted on REGION, with REGION control totals. The second report is sorted by CUSTOMER, with CUSTOMER control totals. The last report is sorted by descending AMOUNT, and has no control breaks. This report includes only those records whose AMOUNT field contains a value greater than 100.

How Many Reports Can I Request?

Each NEWOUT statement results in one additional report. Spectrum Writer does not impose a limit on the number of NEWOUT statements that you can specify. You can produce as many reports in a single run as your system resources will allow.

Each report in a run does require additional system resources (in particular, storage.) When a job needs more resources than are available, an ABEND can occur. For suggestions on maximizing storage resources, see "MVS Storage Considerations" (page 3) or "VSE Storage Considerations" (page 5).

Operating System Considerations (MVS)

Output File DD Statements

When requesting multiple reports, the execution JCL must have one DD statement for each report produced. Following are the default names of the DD statements that Spectrum Writer writes to:

- SWOUTPUT (for the first report)
- SWOUT002 (for the second report)
- SWOUT003 (for the third report)
- and so on.

Add the appropriate DD statement(s) to your JCL, depending on how many reports you will be creating.

> **Note:** If desired, you can override the DD name used by Spectrum Writer for an output. You can also specify, for each output, the type of file to write (QSAM or VSAM) and, for new datasets, the LRECL to use. Specify your overrides with the OUTDDN, OUTTYPE and/or OUTLRECL options (in an OPTION statement.) The overrides will apply to the report that is currently being defined.

Sort Work File DD Statements

When producing multiple reports, you may need to add DD statements to your JCL for additional sort work files. Spectrum Writer starts a separate Sort subtask for each (sorted) report in a run. When a sort cannot be performed entirely in memory, the Sort program uses temporary sort work files. These sort work files may not be shared -- each sort must have its own separate work files. At some shops, sort work files are dynamically allocated by the Sort program. If your shop uses such dynamic allocation, you do not need to add any additional DDs to your JCL. Otherwise, you should add new sort work DD statements for each additional report. The sort work file DDs must be named as followed:

- SORTWK01, SORTWK02, SORTWK03, ... (for the first report)
- SRT2WK01, SRT2WK02, SRT2WK03, ... (for the second report)
- SRT3WK01, SRT3WK02, SRT3WK03, ... (for the third report)
- ...
- SR10WK01, SR10WK02, SR10WK03, ... (for the tenth report)
- SR11WK01, SR11WK02, SR11WK03, ... (for the eleventh report)
- and so on

MVS Storage Considerations

Each additional report in a run requires additional storage. Mostly this storage is needed for the Sort program. If your job fails due to lack of storage, consider the following:

- increase the value of the REGION= parm in the JOB or EXEC statement.

- reduce the amount of storage used by each Sort. By default, Spectrum Writer allows 256K of storage for each Sort. You can change this default with the SORTSIZE option (in an OPTIONS statement):

```
OPTION: SORTSIZE(128)
```

  The above statement causes the Sort program (for the current report) to use only 128K of storage. Note that the SORTSIZE option applies only to the output currently being defined. Thus to reduce the amount of storage used in all sorts, you should add a SORTSIZE option statement to the control statements for each report.

Operating System Considerations (VSE)

Output Attributes

By default, Spectrum Writer writes all reports to printer-type logical units. Following are the default logical units that Spectrum Writer writes to:

- SYS011 (for the first report)
- SYS012 (for the second report)
- SYS013 (for the third report)
- and so on.

If you wish to write a report to a different logical unit, or to a SAM or VSAM file, use the OUTATTR option (in an OPTION statement.)

The OUTATTR option describes the output attributes for the report currently being defined. For example, to write your 3 reports to DLBLs named BYREG, BYCUST and BYAMT, you could use these control statements:

```
INPUT: SALES-FILE
OPTION: OUTATTR(SAM,'BYREG',100,1000)
TITLE: 'SALES BY REGION'
COLUMNS: REGION CUSTOMER AMOUNT TAX
SORT: REGION
BREAK: REGION

NEWOUT:
OPTION: OUTATTR(SAM,'BYCUST,150,1500)
TITLE: 'SALES BY CUSTOMER'
COLUMNS: CUSTOMER REGION AMOUNT TAX  EMPL-NAME
SORT: CUSTOMER
BREAK: CUSTOMER

NEWOUT:
OPTION: OUTATTR(SAM,'BYAMT',80,800)
TITLE: 'SALES SORTED BY DESCENDING AMOUNT'
COLUMNS: AMOUNT CUSTOMER
SORT: AMOUNT(D)
```

Sort Work File DLBL Statements

When producing multiple reports, you may need to add DLBL (and EXTENT) statements to your JCL for additional sort work files. Spectrum Writer starts a separate Sort subtask for each (sorted) report in a run. When a sort cannot be performed entirely in memory, the Sort program must use temporary sort work files. These sort work files may not be shared among sorts -- each sort must have its own separate work files. At some shops, sort work

files are dynamically allocated by the Sort program. If your shop uses such dynamic allocation, you do not need to add any additional DLBLs to your JCL. Otherwise, you should add new sort work DLBL statements for each additional report. The sort work file DLBLs must be named as followed:

- SORTWK1, SORTWK2, SORTWK3, ... (for the first report)
- SRT2WK1, SRT2WK2, SRT2WK3, ... (for the second report)
- SRT3WK1, SRT3WK2, SRT3WK3, ... (for the third report)
- ...
- SR10WK1, SR10WK2, SR10WK3, ... (for the tenth report)
- SR11WK1, SR11WK2, SR11WK3, ... (for the eleventh report)
- and so on

Note: many Sort programs assume, by default, that a single sort work file is available in the execution JCL. To explicitly tell the Sort program how many sort work files are provided for in the JCL, use the SORTWORKNUM option (in a Spectrum Writer OPTIONS statement. Spectrum Writer will then pass this information to the Sort program.) For example:

```
OPTION: SORTWORKNUM(3)
```

The above statement tells Spectrum Writer that 3 sort work files are defined in the JCL for the report being defined. Note that the SORTWORKNUM option applies only to the report currently being defined. Thus, you may want to include a SORTWORKNUM option among the control statements for *each* report.

Note: Even in shops where the sort work files are dynamically allocated, the SORTWORKNUM option may still be required in order to trigger the dynamic allocation.

VSE Storage Considerations

Each additional report in a run requires additional storage. Mostly this storage is used by the Sort program. If your job fails due to lack of storage, consider the following:

- increase the amount of reserved storage using the EXEC statement's SIZE parm. The storage used by the Sort program comes from this part of the partition. Thus, if your run produces 3 reports, each one using the default 256K for its sort, then 768K will be required just for the Sort programs. Round this up for other users of storage:

  ```
  // EXEC SPECTWTR,SIZE=(SPECTWTR,900K)
  ```

- reduce the amount of storage used by each Sort, By default, Spectrum Writer allows 256K for each Sort. You can change this default with the SORTSIZE option (in an OPTIONS statement):

  ```
  OPTION: SORTSIZE(128)
  ```

  The above statement causes the Sort program (for the current report) to use only 128K of storage. Note that the SORTSIZE option applies only to the output currently being defined. Thus to reduce the amount of storage used in all sorts, you should add a SORTSIZE option statement to the control statements for each report.

- try running the job in a larger partition

Working With Arrays

Programmers of low level languages process arrays using index variables, program loops, etc. As a high level, non-procedural report writer, Spectrum Writer does not have the procedural elements (i.e., "go to" or looping instructions) that are needed with such methods. Therefore, other methods must be used to process arrays with Spectrum Writer.

The following sections discuss methods that can be useful when working with records that contain arrays. These methods are:

- normalizing the input records (see below)
- printing multiple report lines per input record in conjunction with the SKIPZERODETAIL option (see page 156 in the full manual.)

Using Normalization to Process Arrays

Often, the best way to process an array with Spectrum Writer is to "normalize" the array-containing records. During normalization, Spectrum Writer turns each *physical* input record into one or more *logical* records. Each logical record corresponds to one "occurrence" of the array in the physical record. These logical records are all identical to the physical record, *except* for the part of the record that contains the first occurrence of the array. That part of the record is overlaid, successively, by the second, third, fourth, etc. occurrence of the array.

In effect, Spectrum Writer loops through the array for you, building logical records by moving, one at a time, each occurrence of the array into the first position. The result is that instead of a single record containing multiple occurrences of an array, we end up with multiple records that each contain a single occurrence of the array in a fixed location.

Figure 1 (page 7) shows an example of a normalized file. As you can see in the Cobol layout, the SALES-HISTORY record contains an array named SALE-ARRAY. The array holds information for up to 6 sales. The NUM-SLOTS field tells how many occurrences of the array are actually used in any particular record. You can see that the normalized file contains from 1 to 6 logical records for each physical record in the original file. The normalized file contains exactly the same sales information as the physical file. The difference is that in the normalized file, the information for all sales appears in one location -- the first occurrence of the array (where the SALE-DATE and SALE-AMT fields are located). Also, the empty occurrences of the array are eliminated from the normalized file (that is, no logical records are created for them.)

Normalized files are quite easy to process with Spectrum Writer. You simply ignore the array (beyond the first occurrence). In the normalized file, there is only one occurrence of relevant data in each record, always in the same location.

Cobol Definition of SALES-HISTORY File

```
01    SALES-HISTORY-REC
      05 NAME                   PIC X(10).
      05 CITY                   PIC X(10).
      05 NUM-SLOTS              PIC 9.
      05 SALE-ARRAY  OCCURS 6 TIMES.
         10 SALE-DATE           PIC 9(6).
         10 SALE-AMT            PIC 9(5)V9(2).
      05 RECORD-STATUS          PIC X(1).
```

Physical SALES-HISTORY File (Sales Data is in 6 Locations)

		sale	sale	sale	sale	sale	sale
BAKER	BOSTON	2921201004239893010400912250000000000000000000000000000000000000000000000000000000000A
CHAVEZ	MIAMI	1930125018890100000000000000000000000000000000000000000000000000000000000000000000000B
JEFFERSON	CHICAGO	2930120006675593012300442340000000000000000000000000000000000000000000000000000000000B
JOHNSON	DALLAS	5921230010081093010200554759301100075065930111002998093011900301620000000000000A
JONES	ATLANTA	6921229007110592123000192569301080109023930110005247593011300789129301160120030A
MORRISON	NEW YORK	3930102005220093010400919449301060140246000000000000000000000000000000000000000B
SHARP	PORTLAND	1930131006001900000000000000000000000000000000000000000000000000000000000000000A
SMITH	ST LOUIS	4930119003342393012100708109301240100056930128002007293013100941990000000000000B

File Definition and INPUT Statements to Normalize the SALES-HISTORY File

```
FILE:  SALES-HISTORY DDNAME(SALEHIST) LRECL(100)
FIELD: NAME          LENGTH(10)
FIELD: CITY          LENGTH(10)
FIELD: NUM-SLOTS     LENGTH(1)     TYPE(NUM)
FIELD: SALE-ARRAY    LENGTH(13)
FIELD: SALE-DATE     COLUMN(*-13)  TYPE(YYMMDD)
FIELD: SALE-AMT      LENGTH(7)     TYPE(NUM)     DEC(2)
FIELD: RECORD-STATUS COLUMN(SALE-ARRAY + 78) LENGTH(1)
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS)
```

Normalized (Logical) SALES-HISTORY File (Sales Data is in 1 Location)

		sale
BAKER	BOSTON	2921201004239893010400912250000000000000000000000000000000000000000000000000000000000A
BAKER	BOSTON	2930104009122593010400912250000000000000000000000000000000000000000000000000000000000A
CHAVEZ	MIAMI	1930125018890100000000000000000000000000000000000000000000000000000000000000000000000B
JEFFERSON	CHICAGO	2930120006675593012300442340000000000000000000000000000000000000000000000000000000000B
JEFFERSON	CHICAGO	2930123004423493012300442340000000000000000000000000000000000000000000000000000000000B
JOHNSON	DALLAS	5921230010081093010200554759301100075065930111002998093011900301620000000000000A
JOHNSON	DALLAS	5930102005547593010200554759301100075065930111002998093011900301620000000000000A
JOHNSON	DALLAS	5930110007506593010200554759301100075065930111002998093011900301620000000000000A
JOHNSON	DALLAS	5930111002998093010200554759301100075065930111002998093011900301620000000000000A
JOHNSON	DALLAS	5930119003016293010200554759301100075065930111002998093011900301620000000000000A
JONES	ATLANTA	6921229007110592123000192569301080109023930110005247593011300789129301160120030A
JONES	ATLANTA	6921230001925692123000192569301080109023930110005247593011300789129301160120030A
JONES	ATLANTA	6930108010902392123000192569301080109023930110005247593011300789129301160120030A
JONES	ATLANTA	6930110005247592123000192569301080109023930110005247593011300789129301160120030A
JONES	ATLANTA	6930113007891292123000192569301080109023930110005247593011300789129301160120030A
JONES	ATLANTA	6930116012003092123000192569301080109023930110005247593011300789129301160120030A
MORRISON	NEW YORK	3930102005220093010400919449301060140246000000000000000000000000000000000000000B
MORRISON	NEW YORK	3930104009194493010400919449301060140246000000000000000000000000000000000000000B
MORRISON	NEW YORK	3930106014024693010400919449301060140246000000000000000000000000000000000000000B
SHARP	PORTLAND	1930131006001900000000000000000000000000000000000000000000000000000000000000000A
SMITH	ST LOUIS	4930119003342393012100708109301240100056930128002007293013100941990000000000000B
SMITH	ST LOUIS	4930121007081093012100708109301240100056930128002007293013100941990000000000000B
SMITH	ST LOUIS	4930124010005693012100708109301240100056930128002007293013100941990000000000000B
SMITH	ST LOUIS	4930131009419993012100708109301240100056930128002007293013100941990000000000000B

Figure 1. A Normalized File

For example, assume that we want a report that simply lists all of the sales over $100 in the SALES-HISTORY file. If we used the physical file, we would have to examine up to 6 different amount fields in each record. We would also need to check the NUM-SLOTS field to see which amount fields were actually used in a given record. This would require 6 COMPUTE statements. Then, we would need 6 COLUMNS statements to potentially print each of the 6 amount fields, plus an option to suppress any zero lines, and so on. (The basics of this alternative approach are described in the section beginning on page 156 in the full manual.)

On the other hand, if we let Spectrum Writer normalize the file, we can easily produce the report using just the SALE-DATE and SALE-AMT fields, like this:

```
INCLUDE IF: SALE-AMT > 100
COLUMNS:   NAME  SALE-DATE  SALE-AMT
```

Remember that every sale amount in the array of the original file now exists in the SALE-AMT field of some logical record. We simply include the records where that amount is over $100.

Figure 2 (page 9) shows a report that uses the above statements.

> **Note:** a normalized file exists only as a temporary, logical file. During the run, the records are built, processed by Spectrum Writer, and then discarded. They are not actually written to a physical file.

The next section explains exactly how to use the NORMALIZE parm to have Spectrum Writer normalize an input file.

The NORMALIZE Parm

To normalize an array in an input file, Spectrum Writer must know 3 things about that array:

- what column the array **starts in**
- the **total size** of each occurrence of the array
- **how many** occurrences there are

Provide this information by adding a NORMALIZE parm to the INPUT statement. The syntax of the NORMALIZE parm is:

```
NORMALIZE(normalize-field, occurs-expression, ...)
```

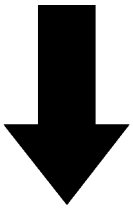
The **normalize field** must be a field that defines the *entire first occurrence* of the array that you want to normalize. In **Figure 1** (page 7), the normalize field is SALE-ARRAY. SALE-ARRAY is a 13-byte character field that includes both the SALE-DATE and SALE-AMT fields. Thus, it defines the entire first occurrence of the array.

Note that in that example we could *not* use SALE-DATE as the normalize field. Doing so would yield incorrect results because it does not define the entire first occurrence of the array. It only defines the first 6 bytes of the first occurrence.

Why is the normalize field's length important? Because it tells Spectrum Writer how many bytes to move each time it builds a new logical record. It also tells Spectrum Writer where to start taking the new bytes from (namely, from the first byte following the normalize field.)

These Control Statements:

```
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS
TITLE: "SALES OVER $100"
INCLUDEIF: SALE-AMT > 100
COLUMNS: NAME  SALE-DATE  SALE-AMT
```



Produce this Report:

SALES OVER $100

NAME	SALE DATE	SALE AMT
BAKER	12/01/92	423.98
BAKER	01/04/93	912.25
CHAVEZ	01/25/93	1,889.01
JEFFERSON	01/20/93	667.55
JEFFERSON	01/23/93	442.34
JOHNSON	12/30/92	1,008.10
JOHNSON	01/02/93	554.75
JOHNSON	01/10/93	750.65
JOHNSON	01/11/93	299.80
JOHNSON	01/19/93	301.62
JONES	12/29/92	711.05
JONES	12/30/92	192.56
JONES	01/08/93	1,090.23
JONES	01/10/93	524.75
JONES	01/13/93	789.12
JONES	01/16/93	1,200.30
MORRISON	01/02/93	522.00
MORRISON	01/04/93	919.44
MORRISON	01/06/93	1,402.46
SHARP	01/31/93	600.19
SMITH	01/19/93	334.23
SMITH	01/21/93	708.10
SMITH	01/24/93	1,000.56
SMITH	01/28/93	200.72

*** GRAND TOTAL (24 ITEMS) 17,445.76

Figure 2. A report that uses normalization to process an array

The **occurs expression** in the NORMALIZE parm tells Spectrum Writer how many occurrences of the array to process. It can be a constant numeric literal, the name of a numeric field, a numeric COMPUTE field, or any other valid numerical expression.

Let's look at an example of the NORMALIZE parm in an INPUT statement:

```
INPUT: SALES-HISTORY
       NORMALIZE(SALE-ARRAY, NUM-SLOTS)
```

The above statement tells Spectrum Writer to normalize the input records from the SALES-HISTORY file. The first occurrence of the array being normalized is defined by SALE-ARRAY. The number of occurrences to be used from the array is contained in the NUM-SLOTS field.

As records are read from the SALES-HISTORY file, here is what happens.

First, the unchanged physical record is processed by Spectrum Writer as usual. (That is, the INCLUDEIF tests are performed on it, and, if included, its contents are formatted into a line of the report.) This physical record is considered the first logical record.

Next, a second logical record is created by moving the 13 bytes immediately following the SALE-ARRAY field into the SALE-ARRAY area of the record. Now, SALE-DATE contains the date from the second occurrence of the array, and SALE-AMT contains the amount from the second occurrence of the array. This logical record is then processed just as if it had been read directly from the input file. (The INCLUDEIF tests are performed on it, and, if included, its contents are formatted into a line of the report.)

Then, a third logical record is created by moving the *next* 13 bytes into the SALE-ARRAY area. Now, SALE-DATE and SALE-AMT contain the date and the amount from the third occurrence of the array. This third logical record is then processed just as if it had been read directly from the input file, and so on

The number of logical records produced for each physical record is determined by the value of the NUM-SLOTS field (in this example). If NUM-SLOTS is 1 (or less), only the original, physical record is processed. If NUM-SLOTS contains 2, then 2 logical records are processed (the original physical record, plus one additional record.). If NUM-SLOTS contains 3, then 3 logical records are processed, and so on.

> **Note:** you can also put the NORMALIZE parm directly in the FILE statement that defines a file. This is convenient when you know that you will *always* want to normalize a certain file. That way, you don't need to include the NORMALIZE parm in the INPUT statement every time you produce a report from that file. (You can still prevent normalization for individual runs by adding the NONORMALIZE parm to the INPUT statement.)

File Definition Tips for Records with Arrays

The FIELD statements that define an array will be different, depending on whether or not you will normalize that array. This section explains the differences.

Consider the SALES-HISTORY file, which contains an array with information for up to 6 sales. For runs where we normalize that file, we would use the file definition statements shown in **Figure 1** (page 7). But for runs where we do not normalize the file, we might prefer to use the file definition statements shown on page 157 in the full manual.

If you are *not* normalizing a record, and you want to access data from an array, you must define each occurrence of the array as a separate field. That is because the only way to

access each occurrence of the array is to refer to it specifically by its unique fieldname. Thus, on page 157 each occurrence of the array in the SALES-HISTORY file is defined individually (SALE-DATE-1, SALE-AMT-1, SALE-DATE-2, SALE-AMT-2, etc.) This file definition is for use in reports where the SALES-HISTORY file is *not* normalized.

However, if you *are* using normalization to process an array, it is not necessary to define all of the occurrences of the array. It is sufficient to define just the fields in the first occurrence of the array. (Plus, you may need to define one additional field that includes the entire first occurrence of the array, for use in the NORMALIZATION parm.) You do not need to define the other occurrences of the array. That is because, after normalization, all array data will be located in the first occurrence of the array (of some logical record). Thus, if we know that the SALES-HISTORY file will always be normalized, we can define it as we did in **Figure 1** (page 7). It is not necessary to define SALE-DATE-2, SALE-AMT-2, SALE-DATE-3 and so on.

Remember that if the array contains more than one field in each occurrence, you must also define a "high-level" field that defines the entire first occurrence of the array. This field will be needed in the NORMALIZATION parm. In the SALES-HISTORY file, we defined SALE-ARRAY as a high level field which includes both SALE-DATE and SALE-AMT. Define the high-level field as a character field whose length is the length of the entire first occurrence of the array. That is, its length will be the sum of the lengths of each individual field.

> **Caution:** Cobol sometimes adds what are called "slack bytes" at the end of each array occurrence. This is done in order to align the next occurrence on a halfword or fullword boundary. This may happen if a field in your Cobol record layout uses the SYNCHRONIZED parm. Any such slack bytes must be included in the length of the high-level field you define.

After defining the high-level field, you can use a COLUMN(*-nnn) parm on the next FIELD statement to "back up" again to the beginning of the array. You can then start "redefining" the lower level fields in that same portion of the record.

You may want to define your array so that it can be used either with or without normalization. In that case, define all of the occurrences of the array, as well as one high-level field for the entire first occurrence.

Normalizing Nested Arrays

Some records contain "arrays within arrays." Such arrays are also called "nested arrays." Spectrum Writer is able to normalize records containing any number of nested arrays.

Specify one NORMALIZE parm for each array level. The first NORMALIZE parm defines the outermost array. The next NORMALIZE parm defines the next deeper array, and so on.

For example, consider this Cobol record layout:

```
01 RECORD.
     05 EMPL-NAME          PIC X(20).
     05 SALE-ARRAY OCCURS 10 TIMES.
        10 SALE-DATE       PIC 9(6).
        10 SALE-CUSTOMER   PIC X(10).
        10 SALE-PRODUCT-CODE OCCURS 5 TIMES PIC X(3).
     05 RECORD-STATUS      PIC X(1).
```

The above record contains a nested array. The outer array (SALE-ARRAY) contains an inner array (SALE-PRODUCT-CODE). We could define this record to Spectrum Writer in the following way:

```
FIELD: EMPL-NAME           LENGTH(20)
FIELD: SALE-ARRAY          LENGTH(31)
FIELD: SALE-DATE     COLUMN(*-31) TYPE(YYMMDD)
FIELD: SALE-CUSTOMER       LENGTH(10)
FIELD: SALE-PRODUCT-CODE   LENGTH(3)
*
FIELD: RECORD-STATUS COLUMN(SALE-ARRAY + 310) LENGTH(1)
```

Since we will be normalizing this file, it is sufficient to define only the first occurrence of each array. (That is because, once normalized, the relevant data in each logical record will be in that first occurrence location).

As you can see, we first defined a field that includes the whole, first 31-byte occurrence of the outer array (SALE-ARRAY). We then backed up 31 bytes in order to define the individual fields within that occurrence. First we defined the 6-byte SALE-DATE field, then the 10-byte SALE-CUSTOMER field. The last field within the 31-byte SALE-ARRAY field is the SALE-PRODUCT-CODE array (a total of 15 bytes). Again, it was only necessary to define the first occurrence of that array. And, since the SALE-PRODUCT-CODE array does not have multiple fields per occurrence, it was not necessary to define a higher-level field to define its entire first occurrence. The 3-byte SALE-PRODUCT-CODE field itself defines the entire first occurrence of that inner array.

After defining the first occurrence of the arrays, we defined the RECORD-STATUS field. We use the COLUMN(SALE-ARRAY + 310) parm to locate this field in the correct column. This COLUMN parm tells Spectrum Writer to locate the RECORD-STATUS field 310 bytes after the start of the SALE-ARRAY field. (Ten occurrences of the 31-byte SALE-ARRAY field is a total of 310 bytes.)

After defining our file as above, we can normalize it like this:

```
INPUT: OUR-FILE
       NORMALIZE(SALE-ARRAY, 10)
       NORMALIZE(SALE-PRODUCT-CODE, 5)
```

As Spectrum Writer normalizes the input records, it first "loops through" the most deeply nested array (the one specified in the last NORMALIZE parm). That is the SALE-PRODUCT-CODE array in this example. Thus, the first logical record, as always, is the unchanged physical record. The second logical record will have the second product code in SALE-PRODUCT-CODE. The third logical record will have the third product code in SALE-PRODUCT-CODE, and so on for the fourth and fifth logical records.

Then, having fully normalized the inner array for the first occurrence of the outer array, Spectrum Writer begins normalizing the outer array (SALE-ARRAY). Thus, for the sixth logical record, Spectrum Writer moves the 31 bytes following the SALE-ARRAY field into the SALE-ARRAY location. That is, it moves the second occurrence of the outer array into the first occurrence's location. This 31-bytes includes the second occurrence of the SALE-DATE and SALE-CUSTOMER fields. It also includes the entire 15-byte SALE-PRODUCT-CODE array from the second occurrence of SALE-ARRAY. At this point, the SALE-PRODUCT-CODE field contains the first occurrence of that array (within the second occurrence of the outer array.) Using Cobol notation, we could say it contains SALE-PRODUCT-CODE (2, 1). After processing this logical record, Spectrum Writer continues to fully normalize the inner array. Thus, for the next logical records, Spectrum Writer moves the 2nd, 3rd, 4th and 5th occurrences of the inner array to SALE-PRODUCT-CODE.

After that, Spectrum Writer moves the third occurrence of the outer array (SALE-ARRAY) into the SALE-ARRAY field, and so on.

Each time one inner level array has been fully normalized, the next higher array level is adjusted and all lower levels are normalized all over again.

Spectrum Writer continues normalization in this manner until the last occurrence of the inner array has been normalized for the last occurrence of the outer array. Thus, each physical record in this example results in 50 logical input records (the 5 occurrences of the inner array times the 10 occurrences of the outer array.)

Normalizing Multiple, Non-Nested Arrays

Some records contain multiple arrays that are *not* nested. That is, there may be 2 or more independent arrays in a record. For example, consider this Cobol record layout:

```
01 RECORD.
     05 EMPL-NAME                        PIC X(20).
     05 SALE-DATE      OCCURS 10 TIMES   PIC 9(6).
     05 HIRE-DATE                        PIC 9(6).
     05 SALE-CUSTOMER  OCCURS 10 TIMES   PIC X(10).
     05 RECORD-STATUS                    PIC X(1).
```

In this record layout, both SALE-DATE and SALE-CUSTOMER are arrays. But they are not nested. We could define this record to Spectrum Writer in the following way:

```
FIELD: EMPL-NAME        LENGTH(20)
FIELD: SALE-DATE        TYPE(YYMMDD)
FIELD: HIRE-DATE        COLUMN(*+54)  TYPE(YYMMDD)
FIELD: SALE-CUSTOMER    LENGTH(10)
FIELD: RECORD-STATUS    COLUMN(*+90)  LENGTH(1)
```

Note again that when normalizing a file, we only need to define the first occurrence of each array. Thus we defined just the first SALE-DATE field at the beginning of that array. Since this is an array of just a single field, we did not need to define a separate "high level" field. The SALE-DATE field itself defines the entire first occurrence of the array.

We used the COLUMN(*+54) parm on the HIRE-DATE field to skip over the other 9 undefined occurrences of the 6-byte sales date field. (We could also have specified COLUMN(SALE-DATE + 60). Either method will properly locate the HIRE-DATE field.)

Similarly, we defined only the first occurrence of the SALE-CUSTOMER array.

We used the COLUMN(*+90) parm on the RECORD-STATUS field to skip over the other 9 undefined occurrences of the 10-byte customer field.

Now that we have defined the file, how do we normalize it? There are two different ways to normalize records that contain non-nested arrays. The method you choose will depend on the logical relationship between the data in the two arrays.

Often, the data in the two arrays will have a one-to-one relationship. That is, the first date in the SALE-DATE array corresponds to the first customer in the SALE-CUSTOMER array. The second date in the SALE-DATE array corresponds to the second customer in the SALE-CUSTOMER array, and so on.

If this is the case, you want to normalize the two arrays in parallel. You can think of it as "stepping through" both arrays in sync. To normalize two or more arrays in parallel, specify all of the arrays in a *single* NORMALIZE parm:

```
INPUT: OUR-FILE
       NORMALIZE(SALE-DATE, 10, SALE-CUSTOMER, 10)
```

When performing this normalization, the first logical record, as always, will be the unchanged physical record. (The first date is in SALE-DATE and the first customer is in SALE-CUSTOMER.) The second logical record will have the second date in SALE-DATE and the second customer in SALE-CUSTOMER. The third logical record will have the third date in SALE-DATE and the third customer in SALE-CUSTOMER, and so on. If you normalize the above file in this manner, each physical record will result in 10 logical records.

On the other hand, you may have two separate arrays in a record whose data is *not* related in the manner described above. In other words, *all* occurrences of the first array may apply to *all* occurrences of the second array. In that case, you would normalize them *as if* they were nested (even though they are not physically nested). That will cause one logical record to be created for every possible combination of items from the two arrays. To normalize in this manner, use two separate NORMALIZE parms (just as for true nested arrays):

```
INPUT: OUR-FILE
       NORMALIZE(SALE-DATE, 10)
       NORMALIZE(SALE-CUSTOMER, 10)
```

When performing this normalization, the first logical record is, as always, the unchanged physical record. (The first date is in SALE-DATE and the first customer is in SALE-CUSTOMER.) The second logical record will retain the first date in SALE-DATE, but move the second customer to SALE-CUSTOMER. The third logical record again retains the first date in SALE-DATE and now has the third customer in SALE-CUSTOMER. Thus, the first ten logical records all have the first date in SALE-DATE, while the SALE-CUSTOMER array is normalized.

Next (for the eleventh logical record), the second date is moved to SALE-DATE and SALE-CUSTOMER is re-initialized to contain the first customer. The next logical record has the second date and the second customer. The next one has the second date and the third customer, and so on.

When normalized in this way, each physical record results in 100 logical records. (Ten occurrences of the first array times the ten occurrences of the second array.)

You can specify the two NORMALIZE parms in either order (since the arrays are not physically nested.) The only difference will be the order in which Spectrum Writer builds the logical records. It always normalizes the last NORMALIZE parm first.

Normalizing only Certain Records

Some files contain more than one type of record. For example, a file may contain a combination of header records and detail records. Perhaps the detail records contain an array that must be normalized, while the header records do not contain that array. Or, the header records might contain an entirely different array in a different location.

For such files, you need **conditional normalization**. Spectrum Writer provides the NORMWHEN parm to perform conditional normalization.

When a NORMALIZE parm is present in an INPUT or READ statement, the default is for Spectrum Writer to normalize *all* of the records from that file. You can, however, put a NORMWHEN parm ahead of the NORMALIZE parm(s). In that case, the normalization is done only on records where the condition specified in the NORMWHEN parm is true. For example:

```
INPUT: BATCH-FILE
       NORMWHEN(RECORD-TYPE = 'HDR')
         NORMALIZE(STATUS-ARRAY, 5)
       NORMWHEN(RECORD-TYPE = 'DET')
         NORMALIZE(CUSTOMER-ARRAY, 8)
```

The above statements tell Spectrum Writer to normalize the STATUS-ARRAY only for those physical records where the RECORD-TYPE field contains 'HDR'. And the CUSTOMER-ARRAY will be normalized only for those physical records where the RECORD-TYPE field contains 'DET'. Records with any other value in the RECORD-TYPE field will not be normalized at all. (That is, only the physical records themselves will be processed.)

Each NORMWHEN parm governs the NORMALIZE parm(s) that follow it (until the next NORMWHEN parm, if any). Spectrum Writer first tests the condition in the first NORMWHEN parm. If true, it performs the complete normalization specified in the following NORMALIZE parm(s). After that normalization, Spectrum Writer then tests the condition in the next NORMWHEN parm. If true, Spectrum Writer then performs the normalization specified by the following NORMALIZE parm(s), and so on.

If the conditions in multiple NORMWHEN parms are true, *each* of the corresponding normalizations will be performed on that record. When a record is normalized multiple times, the unchanged physical record is processed only one time (not one time per normalization).

Any valid conditional expression is allowed within the NORMWHEN parm.

If any NORMALIZE parms precede the first NORMWHEN parm, their normalization will be performed on every record.

Normalizing an Auxiliary Input File

If the records read from an auxiliary input file contain an array, you may want to normalize those records as well. Do that by adding the necessary NORMWHEN and/or NORMALIZE parms to your READ statement.

Remember that, by default, a READ statement only returns a single record (the first record whose key matches the READKEY value.) Therefore to successfully normalize an auxiliary input file, you must also specify the MULTI parm in the READ statement. The MULTI parm tells Spectrum Writer to use *all* of the records from the file whose key matches the READKEY value. The MULTI parm allows all of the logical records created during the normalization process to be used in the report.

Normalization Errors

If Spectrum Writer detects erroneous normalization information in a record, it does not normalize the field in question for that record. (If normalization was requested for more than one field, Spectrum Writer will still try to perform the other normalization(s).) Examples of normalization errors are:

- an error occurs while trying to compute the "occurs" value. For example, a numeric field involved in the computation might contain a non-numeric value.
- the occurs expression results in a value that is negative or zero.
- the occurs expression results in a value that is too big. That is, the last occurrence of the array would be beyond the end of the record area.

When Spectrum Writer encounters any of these normalization errors, it prints a message in the control listing, along with a dump of the record in question. By default, only the first ten such normalization errors are printed. You can use the MAXNORMDUMP option (in an OPTIONS statement) to print more (or fewer) such messages. The syntax of the MAXNORMDUMP option is:

```
OPTIONS: MAXNORMDUMP(nnnnn/10)
```

By default, normalization error messages are treated as informational messages only. When a normalization error occurs, Spectrum Writer processes the physical record, and then skips the normalization in question for that record. If you want normalization errors to be treated as more serious errors, use the ONNORMERROR option (in an OPTIONS statement). The syntax of the ONNORMERROR option is:

```
ONNORMERROR(DEFAULT/WARNING/ERROR/STOP)
```

For example, to stop the entire run if a normalization error occurs, specify:

```
 OPTION: ONNORMERROR(STOP)
```

How to Stop Reading the Input File Early

A new **STOPWHEN parm** is allowed in either the INPUT statement or in an OPTION statement. The STOPWHEN parm tells Spectrum Writer that it can stop reading the primary input file when a certain condition is met. The syntax of the STOPWHEN parm is:

```
STOPWHEN(conditional-expression)
```

For example:

```
INPUT: SALES-FILE
       STOPWHEN(EMPL-NUM > '039')
```

The above statement tells Spectrum Writer to stop reading the primary input file when it encounters a record whose EMPL-NUM field is greater than ’039’. When Spectrum Writer reads a record whose EMPL-NUM is greater than ’039’, it ignores that record and acts as if it has hit EOF on that file. No more records are read from the file. A STOPWHEN parm in the INPUT statement stops the input file processing for *all* reports in a run.

When requesting more than one report in a run, each report can have a different STOPWHEN condition, or none at all. Specify the STOPWHEN parm for an individual report in an OPTIONS statement within that report’s definition statements:

```
OPTION: STOPWHEN(EMPL-NUM > '036')
```

The above statement tells Spectrum Writer to stop reading the primary input file, *for the current report only*, when it encounters a record whose EMPL-NUM field is greater than '036'. When Spectrum Writer reads a record whose EMPL-NUM is greater than '036', it ignores that record and acts as if it has hit EOF on that file for the current report. However, if other reports are still active, Spectrum Writer continues to read from the input file for those reports.

Upgrading the Severity of I/O Errors

A new **ONIOERROR parm** is supported in the INPUT, READ, and OPTION statements. Use this parm to upgrade the severity of I/O errors. (By default, Spectrum Writer generally treats an I/O error on the primary input file as an "Error", while an I/O error on an auxiliary input file is treated as a "Warning".)

When the ONIOERROR parm is specified in an INPUT or READ statement, it applies only to I/O errors for that particular input file. When this parm is specified in an OPTIONS statement, it applies to I/O errors for all input files used in the run. The format of the ONIOERROR parm is:

```
ONIOERROR(DEFAULT/ERROR/STOP)
```

For example:

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

The above statement tells Spectrum Writer to stop the run with an error if an I/O error occurs while trying to process the EMPL-FILE.

Note: missing records are not considered I/O errors.