New Features in Spectrum Writer 3.0.1

The major new features in release 3.0.1 of Spectrum Writer are:

- you can now specify a special **step completion code for "empty" runs** (runs where no records were included) (page 1)
- a new PICTURE feature **automatically scales numbers** (to be "in thousands," "in millions," etc.) for the best fit within a column (page 2)
- you can now specify a **record name in the COLUMNS** statement to easily output all of the fields in that record (page 3)
- a number of **new built-in functions** are now supported in the COMPUTE statement, including several powerful date- and time-manipulation functions (page 6).

Step Completion Code for Empty Runs

Sometimes it is important to know when Spectrum Writer did not write any records to the report or output file. Now you can specify a special completion code for Spectrum Writer to use for such "empty" runs. Use the EMPTYCC(nn) option in an OPTIONS statement:

```
OPTION: EMPTYCC(12)
```

The above statement tells Spectrum Writer to set the step completion code to 12 if no input records pass the INCLUDEIF conditions for the report. If one or more records are included, the standard completion code will be used (normally 0, unless an error was encountered).

Alternatively, you can specify a completion code to use when records *are included* in a run.

```
OPTION: NOTEMPTYCC(16)
```

The above statement tells Spectrum Writer to set the step completion code to 16 if any records are included in the run. If no records are included, the standard completion code will be used. This option is useful for "exception reports," where you are looking for error situations that should not normally occur. When an exceptional situation *is* found, you want the completion code to call attention to that fact.

> **Note:** these options can not be used to *lower* Spectrum Writer's standard completion code (which may be 4 or 8 if warning or error messages were issued).
>
> **Note:** these options are "by output" options. That is, in a run that produces multiple outputs, the option applies only to the output that is being defined when the OPTION statement is encountered.

"Scaled Numeric" Display Format

Two new scaling symbols in PICTURE display formats allow you to format a column of numbers in a "scaled" format. That is, numbers are automatically scaled down to be "in thousands," "in millions," etc. as needed. This allows 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 need to allow for totals). The second column shows scaled values for the same numbers and only uses 7 bytes. (The total also requires only 7 bytes.)

Here is how Spectrum Writer scales numbers. When a number is too big to fit in a regular (non-scaled) column, a size error indicator appears in the report (***S***). But for scaled columns, when the full number does not fit in the column, Spectrum Writer will choose the best scaling factor (thousands, millions, billions etc.) to make it fit within the PICTURE pattern specified. It then adds a scale abbreviation (K, M, G, etc.) to indicate what scale the number is shown in.

Spectrum Writer supports both base-10 and base-2 scaling, as explained below.

Scaling Syntax in PICTURES

Here is the COLUMNS statement used to format the above columns:

```
OPTION:  SCALEPIC
COLUMNS: SALES(13) SALES(PIC'Z,ZZ9 @')
```

The "@" in the PICTURE indicates that base-10 scaling (division by factors of 1000) is wanted for the column. (Base-10 scaling is normally used with business and financial data.) The "@" symbol also shows where the scale abbreviation (K, M, G etc.) should be placed.

Scaled PICTURES can also include decimal digits, if you like:

```
OPTION:  SCALEPIC
COLUMNS: FILESIZE(13) FILESIZE(PIC'ZZ9.9 @')
```

The above statements result in the following columns.

```
   FILESIZE    FILESIZE
            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 KB, MB, GB, etc. in the column.

```
OPTION:  SCALEPIC
COLUMNS: FILESIZE(13) FILESIZE(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: in order to use "@" and "?" as special scaling symbols in a PICTURE, you must have specified the **SCALEPIC option** in an (earlier) OPTION statement. Otherwise, "@" and "?" will be treated as in prior versions of Spectrum Writer -- as bytes of literal text.

Note: if the field you are scaling can have a **negative value**, *be sure* that the PICTURE includes a leading byte for a minus sign. (That is, begin your PICTURE with a leading minus sign, a space or an extra "Z".) Otherwise, negative numbers will have to be scaled down further than you want (sometimes all the way down to 0)!

Take, for example, this PICTURE which has no extra byte for a minus sign: PIC'ZZ9@'. The number 100,000,000 would format normally as "100M". But the number -100,000,000 appears, surprisingly, as " 0G". Spectrum Writer can't show "-100M" in the 4-byte PICTURE. So it has to scale the number down further to -0.1 billion. Rounding that to a whole number gives 0 billion, which is positive and does fit in the PICTURE (" 0G"). But it's not what you want. Using PIC'-ZZ9@' would give the results you expect for both positive and negative numbers: " 100M" and "-100M".

Note: Spectrum Writer sometimes adds **additional leading digit positions** to user's PICTURES (whether scaled or not). It does that when a long column heading or a column width parm makes a column wider than the width of the PICTURE itself. Usually, the ability to show extra digits is desirable. If, however, you don't want any extra leading digits added to your scaled PICTURE, left-pad your PICTURE with blanks to make it the same size as the column.

For example, if you want to see exactly PIC'Z,ZZ9@' in a column that will be 10 bytes long (because of a long column heading), change the PICTURE to PIC' Z,ZZ9@'.

Including All Fields in the COLUMNS Statement

How can you get all of the fields from an input record into the COLUMNS statement without having to type each field name individually? Just put the record name in the COLUMNS statement. (An input file's record name is, by default, the same as the file name.)

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

The three statements above would reformat the entire contents of the SALES-FILE into a comma-delimited "PC" file.

You can list more than one record name in the COLUMNS statement, if your report uses multiple input files. For example:

```
INPUT: SALES-FILE
READ:  EMPL-FILE READKEY(EMPL-NUM)
COLUMNS: SALES-FILE EMPL-FILE
```

The above statements cause all of the fields from the SALES-FILE record to be output, followed by all of the fields from the EMPL-FILE record.

You can also mix record names and individual field names in the COLUMNS statement:

```
INPUT: SALES-FILE
READ:  EMPL-FILE READKEY(EMPL-NUM)
COLUMNS: LAST-NAME FIRST-NAME SALES-FILE
```

The above statements would create columns for the LAST-NAME and FIRST-NAME fields (from the EMPL-FILE), followed by all of the fields in the SALES-FILE.

COLUMNS Statement Record Name Options

There are a number of options you can use when you specify a record name in the COLUMNS statement. Use these options to:

- specify how overlapping fields should be handled
- specify individual fields to exclude from the output
- specify what order the columns (fields) should appear in

Here is the full syntax of the record name option of the COLUMNS statement:

```
COLUMNS: record-name[( [exclude-field1 exclude-field2 ...]
                       [OUTER/INNER] [BYDEF/BYNAME/BYCOL] [LIST/NOLIST] )] ...
```

Excluding Duplicate Data from Overlapping Fields

Specifying a record name in the COLUMNS statement is a quick way to get all of the data from an input record into your output file. But in most cases there will be some fields that you don't really need or want in your output.

One common example of this are fields that overlap with other fields. For example, consider this definition of a date field:

```
FIELD: SALES-DATE TYPE(YYMMDD)
FIELD: SALES-YY   LEN(2) COLUMN(SALES-DATE)
FIELD: SALES-MM   LEN(2)
FIELD: SALES-DD   LEN(2)
```

The SALES-DATE field defines the whole 6-byte date field in the record. Then, the next three fields *redefine* the individual YY, MM and DD components of the same field.

By default, Spectrum Writer writes *all* of the fields defined for a file to the output. That means it will write all four of the above fields, even though it is a duplication of the same data.

If that is not what you want, specify either the OUTER or INNER parm.The **OUTER parm** tells Spectrum Writer to exclude the outer field(s) when an overlap occurs.

```
COLUMNS: SALES-FILE(OUTER)
```

The above statement would result in SALES-YY, SALES-MM and SALES-DD being written to the output file, but not SALES-DATE.

The **INNER parm** does just the opposite. It excludes the inner fields when an overlap is detected. It would result in only SALES-DATE being written to the output file.

Excluding Individual Fields

There are times when you can not get the exact results you want with either the INNER or OUTER parm. This often happens when there are multiple levels of overlapping fields, or partially overlapping fields. In such cases, you can also name individual fields as **"exclude fields."** Any field name specified in the parms for a record name will *not* be included in the output. You can specify as many exclude fields as you like, in any order. You may also specify exclude fields in conjunction with either the INNER or OUTER parm, if you like.

You may also want to exclude some less important fields from an output in order to make a smaller output record:

```
COLUMNS: SALES-FILE(BACKUP-EMPL-NUM  COMMISSION-RATE  TIME-ON-PHONE)
```

The above statement would write out all fields from the SALES-FILE except for the three fields named in the parms as exclude fields.

> **Note:** if you have actual fields named INNER, OUTER or any of the other parm names, Spectrum Writer assumes you are naming the *field* by that name (as an exclude field), rather than naming the parm. If you have fields with the same name as a parm, you can specify that you mean the *parm* by preceding it with a pound sign (#). For example:
>
> ```
> COLUMNS: SALES-FILE(#INNER #BYNAME)
> ```

Specifying the Field Order in the Output Record

By default, the fields appear in the output record in the order they were defined in (the default BYDEF option). Specify the BYNAME parm if you want the fields to appear in alphabetical order. Or specify BYCOL to put them in column order (that is, the order in which they occur in the input record).

```
COLUMNS: SALES-FILE(BYCOL)
```

Listing the Fields in the Control Listing

By default, Spectrum Writer lists the names of all of the fields it is automatically including in the output (the default LIST option). This might be a long list for some files. If you prefer to suppress the list of field names in the control listing, use the NOLIST parm:

```
COLUMNS: SALES-FILE(NOLIST)
```

Including All COMPUTE Fields

COMPUTE fields that are part of a file definition (that is, that are kept in the copy library along with the FIELD statements) are generally included with the rest of the fields when a record name is specified in the COLUMNS statement.

But there are some COMPUTE fields that are not considered part of any file's definition. Example of such COMPUTE fields are those which are computed without using any fields as operands (that is, they use only literals) and those which are defined out of sequence for an earlier file (while a different file is the "current" file being defined).

Spectrum Writer has a special "record name" called #COMPUTES that includes all COMPUTE fields that are not part of a file definition. Use this special record name to output all COMPUTE fields that are not part of any file's definition. The syntax is:

```
COLUMNS: #COMPUTES[( [exclude-field1 ...] [BYDEF/BYNAME] [LIST/NOLIST] )] ...
```

Note that the INNER, OUTER and BYCOL parms do not apply to this special record name.

New Built-In Functions

A number of new built-in functions are supported in this release, including several that allow powerful manipulation of dates and times. Following is a list of the new functions.

Functions that Return a Numeric Value

#HOURNUM[(time)]

Returns the numeric value of the hours portion of the time argument. Note 1

EXAMPLE: `COMPUTE: A = #HOURNUM(12:30:59)` results in `A=12`

#MINUTENUM[(time)]

Returns the numeric value of the minutes portion of the time argument. Note 1

EXAMPLE: `COMPUTE: A = #MINUTENUM(12:30:59)` results in `A=30`

#SECONDNUM[(time)]

Returns the numeric value of the seconds portion of the time argument. Note 1

EXAMPLE: `COMPUTE: A = #SECONDNUM(12:30:59)` results in `A=59`

#DOWNUM[(date)]

Returns a number representing the day of the week of the argument date. (1 means Sunday, 2 means Monday, ... 7 means Saturday.) Notes 2, 3

EXAMPLE: `COMPUTE: A = #DOWNUM(1/1/2007)` results in `A=2`

Functions that Return a Date Value

#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, WEEKS, MONTHS, YEARS (D, W, M, Y). Notes 2, 3

EXAMPLES: `COMPUTE: A = #INCDATE(1/1/2007, 3, WEEKS)` results in `A = 1/22/2007`
`COMPUTE: YESTERDAY = #INCDATE( -1, DAYS)` results in `YESTERDAY` being the date before the system date.

Note: when incrementing by months or years, the day portion of the resulting date must sometimes be changed to the last day of the month, in order to return a valid calendar date.

`COMPUTE: A = #INCDATE(5/31/2008, 1, MONTH)`
would result in `A = 6/30/2008` (not 6/31/2008 which is not a valid date).
`COMPUTE: B = #INCDATE(2/29/2004, 1, YEAR)`
would result in `B = 2/28/2005` (not 2/29/2005 which is not a valid date).

#INCDATETIME([date,] [time,] number, units) or #INCDATETIME([date,] [time,] time)

Returns the date obtained by incrementing the date and time arguments by the given number of units, or by a time value. Units can be any of these keywords or abbreviations: SECONDS, MINUTES, HOURS (S, M, H). Notes 1, 2, 3

EXAMPLES

`COMPUTE: A= #INCDATETIME(1/1/2008, 23:45:00, 12, MINUTES)` results in `A = 1/1/2008`
`COMPUTE: B= #INCDATETIME(1/1/2008, 23:45:00, 16, MINUTES)` results in `A = 1/2/2008`

Note: this function is often used in conjunction with #INCTIME. Together, they let you add a time interval to a starting date and time and get the resulting date and time. For example, to compute an "expiration" date and time that is 12 hours after SALES-DATE and SALES-TIME, you could use the following:

```
COMPUTE: EXPIRE-DATE = #INCDATETIME(SALES-DATE, SALES-TIME, 12, HOURS)
COMPUTE: EXPIRE-TIME = #INCTIME(SALES-TIME, 12, HOURS)
```

Note: Switches to or from Daylight Saving Time are not taken into account by this function.

#BEGYEAR[(date)]

Returns the first day of the year in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #BEGYEAR(5/15/2008)` results in `A=1/1/2008`

#BEGMONTH[(date)]

Returns the first day of the month in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #BEGMONTH(5/15/2008)` results in `A=5/1/2008`

#BEGWEEK[(date)]

Returns the Sunday of the calendar week in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #BEGWEEK(5/15/2008)` results in `A=5/11/2008`

#ENDYEAR[(date)]

Returns the last day of the year in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #ENDYEAR(5/15/2008)` results in `A=12/31/2008`

#ENDMONTH[(date)]

Returns the last day of the month in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #ENDMONTH(5/15/2008)` results in `A=5/31/2008`

#ENDWEEK[(date)]

Returns the Saturday of the calendar week in which the date argument occurs. Notes 2, 3

EXAMPLE: `COMPUTE: A = #ENDWEEK(5/15/2008)` results in `A=5/17/2008`

#YMD(num, num, num)
#MDY(num, num, num)
#DMY(num, num, num)

Returns a date value based on the three numeric arguments (representing month, day and year in the order indicated by the function name.) The resulting date is not validity-checked to see if it is an actual calendar date. (You can use the #REALDATE function to find out.) The numeric argument representing the year can be any 1 to 4 digit number, and the other two numeric arguments can be any 1 or 2 digit number.

EXAMPLES: `COMPUTE: A = #MDY(12,31,2009)` results in `A=12/31/2009`
`COMPUTE: B = #YMD(9999,99,99)` results in `B=99/99/9999`

Functions that Return a Time Value

#INCTIME([time,] number, units) or
#INCTIME([time,] time)

Returns the time of day obtained by incrementing the time argument by the given number of units, or by another time value. Units can be any of these keywords or abbreviations: SECONDS, MINUTES, HOURS (S, M, H). The result will always be a proper time of day (that is, in the range 00:00:00 to 23:59:59). Note 1

EXAMPLES: `COMPUTE: A=#INCTIME(23:00:00, 2, HOURS)` results in `A = 01:00:00`
`COMPUTE: B=#INCTIME(23:00:00, 05:12:34)` results in `B = 04:12:34`

Note: see also the related #INCDURATION built-in function.

Note: this function is often used in conjunction with #INCDATETIME. Together, they let you add a time to a starting date and time and get the resulting date and time. For example, to compute an "expiration" date and time that is 12 hours after SALES-DATE and SALES-TIME, you could use the following:

```
COMPUTE: EXPIRE-DATE = #INCDATETIME(SALES-DATE, SALES-TIME, 12, HOURS)
COMPUTE: EXPIRE-TIME = #INCTIME(SALES-TIME, 12, HOURS)
```

Note: Switches to or from Daylight Saving Time are not taken into account by this function.

#INCDURATION([time,] number, units) or
#INCDURATION([time,] time)

Returns the time duration (not necessarily a time of day) obtained by incrementing the time argument by the given number of units, or by another time value. Units can be any of these keywords or abbreviations: SECONDS, MINUTES, HOURS (S, M, H). The result is treated as a time duration (interval) and is not converted to a proper time of day. Note 1

EXAMPLES: `COMPUTE: A=#INCDURATION(23:00:00, 2, HOURS)` results in `A = 25:00:00`
`COMPUTE: B=#INCDURATION(23:00:00, 05:12:34)` results in `B = 28:12:34`

Note: see the related #INCTIME built-in function.

Functions that Return a Boolean Value

#ERROR(field)

Returns "true" if the argument field is in error. Otherwise, it returns "false."

Fields in error appear in a report with error indicators (like ***I***, ***Z***, ***V***, etc.). Examples of fields in error are fields containing invalid packed data and compute fields where a divide-by-zero or an overflow occurred. Missing fields are not in error.

EXAMPLES: `COMPUTE: A=#ERROR(AMOUNT)` results in `A` being "true," if the AMOUNT field in the input record does not contain a valid numeric value; otherwise A will be "false."

```
COMPUTE: X = 3 / 0
COMPUTE: B=#ERROR(X)
```
results in `B` being "true"

#ISNUM(char)

Returns "true" if the character argument contains only numeric characters (optionally preceded by leading blanks.) Otherwise, it returns "false." All-blank fields and fields with decimal points or commas, etc. return "false."

Note: a character field that returns "true" for this function can be converted to a numeric field with the #MAKENUM function.

EXAMPLES:
```
COMPUTE: A=#ISNUM(' 123')   results in A being "true"
COMPUTE: B=#ISNUM('123.45') results in B being "false"
COMPUTE: C=#ISNUM('JONES')  results in C being "false"
```

#LEAPYEAR[(date)]

Returns "true" if the year portion of the argument date is a leap year. Otherwise, it returns "false." (The month and day portions of the argument date are not examined for validity.) Note 2

EXAMPLES:
```
COMPUTE: A=#LEAPYEAR(5/15/2008) results in A being "true"
COMPUTE: B=#LEAPYEAR(5/15/2009) results in B being "false"
```

#MISSING(field)

Returns "true" if the argument field is missing. Otherwise, it returns "false."

Fields in the primary input file (specified by the INPUT statement) are never missing. Fields in records read from an auxiliary input file (specified in a READ statement) are missing when no record is found that matches a particular "read key." Fields from DB2 tables used as auxiliary inputs are missing when no row meets the conditions specified in the WHERE parm.

EXAMPLE:
```
INPUT: SALES-FILE
READ:  EMPL-FILE  READKEY(EMPL-NUM)
COMPUTE: A=#MISSING(FIRST-NAME)
```
since FIRST-NAME is a field in the EMPL-FILE, this results in A being "true" if there is no record in the EMPL-FILE whose key matches the EMPL-NUM in the current SALES_FILE record. If such a record does exist, it results in A being "false."

#REALDATE(date)

Returns "true" if the date argument contains a real calendar date. Otherwise, it returns "false."

EXAMPLES:
```
COMPUTE: A=#REALDATE(6/30/2008)
COMPUTE: B=#REALDATE(6/31/2008)
COMPUTE: C=#REALDATE(2/29/2007)
COMPUTE: D=#REALDATE(99/99/9999)
```
results in A being "true"
results in B being "false"
results in C being "false"
results in D being "false"

Function Notes:

1. If the time argument is omitted, the system time is used.
2. If the date argument is omitted, the system date is used.
3. If the date argument to this function is not a valid calendar date, the function returns an "invalid data" error (***I***).