Numeric Data In SAS®: Guidelines for Storage and Display

Paul Gorrell, Social & Scientific Systems, Inc., Silver Spring, MD

ABSTRACT

Understanding how SAS stores and displays numeric data is essential for both accurate computations and effective, useful, reports and tables. SAS stores the values of all numeric variables in *floating-point* representation. This paper begins with a brief, practical, overview of floating point representation and how it relates to programming questions regarding length, precision, and efficient use of disk space. I will discuss situations where numeric length should not be reduced, even if the range of integer values would appear to permit it. For saving disk space, I'll show the advantages of the SAS V8 COMPRESS=BINARY option.

How SAS stores numbers also affects the use of comparison operators, and I will discuss the use of numeric functions such as ROUND, FLOOR and CEIL to achieve program goals. The use of formats for the effective display of numeric data will also be discussed. In particular I will explain the use of the '*w*' (width) and '*d*' (decimal) values of formats such as COMMAw.d, DOLLARw.d and PERCENTw.d. The goal of this paper is to give the SAS programmer a clear conception of how SAS stores and displays numbers—and to therefore increase the programmer's ability to achieve precise program and project objectives.

INTRODUCTION

This paper looks at some important properties of two different, but related, aspects of numeric data in SAS. The first is how numeric data are *stored* (i.e. how a number is represented in the computer). The second is how numeric data are *displayed* (i.e. how a number appears on a screen or piece of paper).

Different systems use different methods to store numbers. SAS uses floating point representation and, by default, stores numeric values using eight bytes. This paper will begin with a brief, practical, sketch of floating point representation and some of its underlying concepts. The goal of this sketch is not to exhaustively explain how to translate base 10 numbers into floating point representation, but rather to give you a more-concrete sense of how the SAS system interprets a statement such as:

```
(1) LENGTH NVAR1 3 ;
```

One reason for not delving into the minutiae of floating point representation is that it is rarely of practical value in the day-to-day work of SAS programming and making decisions concerning the LENGTH of numeric data. It is better to have a small set of guiding principles for making decisions in a timely manner. Of course it is always good to know where to go to get detailed information if you need it, and the RESOURCES section at the end of this paper contains two useful references from SAS Institute (they include additional references).

SAS displays numeric data using either a default FORMAT or one that has been specified by the programmer. This paper will discuss the SAS formats COMMAw.d, DOLLARw.d and PERCENTw.d. Although not as complex a topic as floating point representation, not knowing the proper use of these formats can lead to lots of frustration in generating reports and tables.

HOW SAS STORES NUMBERS

There's quite a bit of detail in this section. In general, there would seem to be little reason to pay attention to this level of detail in making day-to-day programming decisions. So why go to the trouble of devoting time to it? For two reasons: (i) so that you are in a position to evaluate the general claims made here in terms of your particular situation; (ii) you may, at some point, encounter a programming problem that requires a detailed understanding of how SAS stores numbers—and how this differs from how SAS displays numbers.

FLOATING POINT REPRESENTATION

Most papers on floating-point representation and numeric precision only tell part of the story. The part of the story I'm going to tell focuses on giving you enough information to understand, or, at least give you a jumpstart in understanding, more-detailed discussions of these topics. I also hope it allows you to evaluate the guidelines that I give at the end of this section.

The basic unit of storage is the *bit* (binary digit). As the term *binary* suggests, there are two possible values: 0 and 1. A sequence of 8 bits is called a *byte*.

SAS stores numeric data using 64 bits (eight bytes). The eight bytes are divided among 3 different types of information: the *sign*, the *exponent*, and the *mantissa*. An important fourth type of information is the *base*, i.e. the number raised to a power. These are terms for the parts of a number in scientific notation. Floating point representation is just one form of scientific notation. In this system, each number is represented as a value (the mantissa) between 0 and 1 raised to a power of 2 (in a binary, base 2, system). The term *decimal* in 'decimal point' assumes a base 10 system, so the term '*radix* point' is often used when the base is not 10. The radix point for a number is moved (i.e. it *floats* in the picturesque speech of computer science) to the left until the number is a fraction between 0 and 1.

If we use the more-familiar base 10 system, the number 234 would be represented as $.234 \times 10^3$. Placing the decimal point to the left of the number is called 'normalizing the value.' This normalization yields a value between 0 and 1. In addition to the mantissa (.234) and the exponent (3), a full representation of 234 would include the sign (negative or positive).

Here's the byte layout for a 64-bit number in the IEEE system used by Windows (where S = sign; E = exponent; and M = mantissa):

```
(2)    SEEEEEEE EEEEMMMM MMMMMMMM MMMMMMMM
         byte 1   byte 2   byte 3   byte 4
       MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
         byte 5   byte 6   byte 7   byte 8
```

That is, there's 1 bit for the sign, 11 bits for the exponent, and 52 bits for the mantissa. The number of exponent bits determines the magnitude of the numbers that can be represented. The number of mantissa bits determine the precision. For example, an IBM mainframe system will use 56 bits for the mantissa, allowing for greater precision than you get on a PC. In the following discussion I will focus on the mantissa since that is the part of the representation affected by the LENGTH statement.

For 52-bit systems, the functional equivalent of 53 bits is achieved by assuming an 'implied' bit. That is, since the only possible non-zero digit is 1, we can just assume an initial mantissa bit with a value of 1. We'll see that the existence of this implied bit solves an apparent puzzle when we look at numbers in binary where all the mantissa bits have values of 0.

When you use a LENGTH statement such as (1), you are telling the SAS system to only use the first 3 bytes to store the number, i.e. bytes 1-3 in (2). You save 5 bytes per file record this way, but you are potentially losing precision because you are sacrificing 40 mantissa bits. Note that you cannot specify length in terms of bits, or specify which bytes to use. If LENGTH is 3 then you only have the first 3 bytes in (2).

INTEGERS

Below is a table showing (what is often referred to as) the largest integer that can be represented accurately for a given LENGTH specification. The numbers in the table will be different for IBM and VAX computers (consult the *SAS Companion* for your operating system).

LENGTH IN BYTES	LARGEST INTEGER (PC/UNIX)
2	NOT ALLOWED
3	8,192
4	2,097,152
5	536,870,912
6	137,438,953,472
7	35,184,372,088,832
8	9,007,199,254,740,992

Now let's look at the 64-bit representation of 8,192 and 8,193 to see why 8,192 is the 'largest integer' that can be accurately represented. You can create 64-bit output by using the BINARY64. format (The spaces between bytes were added later).

```
(3) The 64-bit representation of 8,192:

       01000000 11000000 00000000 00000000
       00000000 00000000 00000000 00000000
```

```
(4) The 64-bit representation of 8,193:

       01000000 11000000 00000000 10000000
       00000000 00000000 00000000 00000000
```

The first thing to notice is that the difference between 8,192 and 8,193 is in byte 4. If you are limited to the first 3 bytes then this difference is eliminated and, as far as the computer is concerned, the numbers are identical (because bytes 1-3 are identical).

We can show this with the SAS program in (5).

```
(5) DATA ONE ;
       LENGTH VAR1 VAR2 3 ;
       VAR1 = 8192 ;
       VAR2 = 8193 ;
    RUN;

    DATA TWO;
       SET ONE;
       PUT VAR1=22.16 ;
       PUT VAR1=BINARY64. ;
       PUT VAR2=22.16 ;
       PUT VAR2=BINARY64. ;
    RUN;
```

Here's the LOG output you'd get:

```
(6) VAR1= 8192.0000000000000000

VAR1= 01000000 11000000 00000000 00000000
      00000000 00000000 00000000 00000000

    VAR2= 8192.0000000000000000

VAR2= 01000000 11000000 00000000 00000000
      00000000 00000000 00000000 00000000
```

There are a couple of things going on here we need to unpack. First, VAR2 was initially assigned a value of 8,193, but in data set TWO it's 8,192. The reason is that the LENGTH specification of 3 has removed the distinguishing information in byte 4. When SAS stored this variable it only used three bytes.

Second, and this is a point we'll come back to later, in the DATA step (i.e. the program data vector [PDV]) SAS uses the full 8-byte representation of numbers (Note that, for character variables, the LENGTH statement applies both to the PDV and the output data set). If a variable was stored as LENGTH 3, then bytes 4 through 8 are 'filled in' with zeros. This is what has happened with VAR1 and VAR2. When data set ONE is stored, VAR1 and VAR2 are stored with 3 bytes. When they are read for the DATA step that creates TWO, they are expanded to eight bytes, with the last 5 bytes all zeros.

For VAR1 (8,192) this does not result in any loss of precision because the last 5 bytes are all zeros in the original 8-byte representation. Note that there are no mantissa bits with a value of 1. This is because 8,192 is a power of 2 (2^{13}) and so only the implied bit needs a value of 1.

For VAR2 (8,193) there *is* a loss of precision when stored as LENGTH 3 because the 4th bit contains information. This information is lost when the data set is stored. In this case the filling in of bytes 4-8 with zeros results in a value equivalent to 8,192—so the original value of VAR1 (8,193) has been changed to 8,192.

Let's expand on the DATA step in (5) to illustrate a couple of points.

```
(7) DATA ONE ;
       LENGTH VAR1 VAR2 VAR3 3 ;
       VAR1 = 8192 ;
       VAR2 = 8193 ;
       VAR3 = 16384 ;
       PUT VAR1=22.16 ;
       PUT VAR1=BINARY64. ;
       PUT VAR2=22.16 ;
       PUT VAR2=BINARY64. ;
       PUT VAR3=22.16 ;
       PUT VAR3=BINARY64. ;

     RUN;

VAR1= 8192.0000000000000000

VAR1= 01000000 11000000 00000000 00000000
      00000000 00000000 00000000 00000000

VAR2= 8193.0000000000000000

VAR2= 01000000 11000000 00000000 10000000
      00000000 00000000 00000000 00000000

VAR3= 16384.0000000000000000

VAR3= 01000000 11010000 00000000 00000000
      00000000 00000000 00000000 00000000


    DATA TWO;
       SET ONE;
       PUT VAR1=22.16 ;
       PUT VAR1=BINARY64. ;
       PUT VAR2=22.16 ;
       PUT VAR2=BINARY64. ;
       PUT VAR3=22.16 ;
       PUT VAR3=BINARY64. ;
    RUN;

VAR1= 8192.0000000000000000

VAR1= 01000000 11000000 00000000 00000000
      00000000 00000000 00000000 00000000

VAR2= 8192.0000000000000000

VAR2= 01000000 11000000 00000000 00000000
      00000000 00000000 00000000 00000000

VAR3= 16384.0000000000000000

VAR3= 01000000 11010000 00000000 00000000
      00000000 00000000 00000000 00000000
```

The first thing to notice is that the LENGTH statement in the first DATA step has no effect on the length of the variables created in the DATA step. The LENGTH statement only takes effect when the variables are stored.

The second thing to notice is that (unlike with VAR2: 8,193), there is no loss of precision when VAR3 (16,384) is stored. This is because in the full, 8-byte, representation of 16,384, bytes 4-8 contain only zeros. In fact, because 16,384 is a power of 2 (2^{14}), only the implied mantissa bit has information.

So it's not actually true that 8,192 is the largest integer that can be accurately represented with LENGTH 3 (on a PC or UNIX system). What is true is that 8,192 is the largest integer *of a continuous range of integers* that can be accurately represented with 3 bytes. The same is true for the other integers in the right-hand column in the table. These 'largest integer' tables are still handy references for quick decisions, but it's important to know that exceptions exist.

FRACTIONS

It's also important to remember that these tables apply only to *integers*. The situation is somewhat different when we look at fractions. Let's compare 1/10 and 1/2.

```
(8) DATA ONE;
       VAR1 = 1 ;
       VAR2 = 0 ;
       DO X = 1 TO 10 ;
              VAR2 + 0.1 ;
       END;
       VAR3 = 0 ;
       DO X = 1 TO 2 ;
              VAR3 + 0.5 ;
       END;
       IF VAR1 = VAR2
              THEN PUT VAR1 'EQ ' VAR2 ;
       ELSE PUT VAR1 'NE ' VAR2 ;
       IF VAR1 = VAR3
              THEN PUT VAR1 'EQ ' VAR3 ;
       ELSE PUT VAR1 'NE ' VAR3 ;
    RUN;
```

If you run this DATA step, the LOG output will be:

```
(9)

       1 NE 1          [for 0.1]

       1 EQ 1          [for 0.5]
```

The first LOG message looks like a clear contradiction. How can the result be that 1 does not equal 1? The answer, of course, is that the stored value of VAR2 differs from its display form. Here's the 8-byte representation of VAR1 and VAR2:

```
(10)

VAR1= 00111111 11110000 00000000 00000000
      00000000 00000000 00000000 00000000

VAR2= 00111111 11101111 11111111 11111111
      11111111 11111111 11111111 11111111
```

Once we see the 8-byte representation, it's clear that VAR1 and VAR2 have different stored values. The output of the comparison also shows that SAS is comparing stored values and not display values. The reason that VAR2 does not equal 1 is because 0.1 cannot be represented precisely in a binary system (though it's straightforward in a decimal system) and this imprecision iterates with each addition in the DO loop.

Now let's compare VAR1 with VAR3:

```
(11)

VAR1= 00111111 11110000 00000000 00000000
      00000000 00000000 00000000 00000000

VAR3= 00111111 11110000 00000000 00000000
      00000000 00000000 00000000 00000000
```

Here we see that adding 0.5 to 0.5 results in a stored value that is equal to 1. Why here and not with 0.1? Notice that, in contrast to VAR2, bytes 3-8 are all zeros. In fact, in binary representation, 0.5 looks a lot like 8,192 and 16,384. This is because these numbers are all powers of 2: $0.5 = 2^{-1}$.

GUIDELINES FOR STORAGE

It's clear that the magnitude of a number is an imperfect predictor of whether or not it can be accurately represented with a reduced number of bytes. The numbers 1/2 and 16,384 can be accurately represented with 3 bytes, but 8,193 cannot.

Let's consider a numeric variable HOSPITAL_DAYS that indicates how many days in a year a person was in the hospital. The largest possible value is 365 in non-leap years. But let's assume that half-days (but no other fractions) are possible, so values such as 189.5 are allowed. This is a variable that, despite the non-integer values, is a candidate for LENGTH 3.

Recognizing candidates for reduced LENGTH is an important part of the decision process, and knowing a bit (no pun intended) about how SAS stores numbers is essential for this. The larger question concerns other factors that are involved in making such a decision.

Here's an example that illustrates how you often need to consider a larger context when making, what appear to be, narrow, technical decisions. Assume that you have a SAS data set with a variable for annual out-of-pocket dental expenditures. The values are all integers and the largest value for this variable on the file is 8,192. Should you set LENGTH 3?

Let's say you do. Fast forward a year or so and the task is to input this file, increase dental expenditures by 13% (to account for inflation), and round the result to the nearest integer. Here's a DATA step that shows the potential problem.

```
(12) DATA ONE;
       LENGTH DENTOOP1 3 ;
       DENTOOP1 = 8192;
       DENTOOP2 = 8192;
     RUN;

     DATA TWO;
       SET ONE;
       DENTOOP1 = ROUND((DENTOOP1*(1.13));
       DENTOOP2 = ROUND((DENTOOP2*(1.13));
       PUT DENTOOP1= ;
       PUT DENTOOP2= ;
     RUN;

     DENTOOP1=9256;
     DENTOOP2=9257;
```

In data set ONE the variable DENTOOP1 has a LENGTH of 3 whereas DENTOOP2 has a LENGTH of 8 (the default). After the 13% increase and rounding in the second DATA step, we see that the modified values depend on the LENGTH specification. What you don't want is a phone call from the client asking why, when a simple check with a hand calculator is used, they get a value of 9257 (i.e. 9256.96 rounded) but your program output is 9256. Finding out why could ruin your whole afternoon.

Of course the problem could have been avoided if the programmer updating dental expenditures had simply (i) run a PROC CONTENTS and noticed that the LENGTH is 3 bytes; (ii) known the interger values for which accuracy is preserved with this LENGTH (iii) run a PROC and noticed that there are values that, if increased by 13%, would exceed 8,192; (iv) realized that variable attributes like LENGTH are inherited by one data set from another; (v) known how to stop the default attribute inheritance for this variable; and (vi) placed a new LENGTH statement in the DATA step that updated the variable's values.

If the steps listed in the last paragraph are second nature to you, everyone you work with, and everyone you will ever hire, then this would favor a decision to reduce numeric length based simply on the values in the data set.

I would argue, however, for a more conservative approach. This approach would first distinguish between *actual* and *permissible* values. By actual values I mean the set of values that happen to be true for a variable on a particular data set. Given reasonable QC, the set of actual values should be a subset of the set of permissible values.

Let's take the dental expenditure variable as an example. On the data set I discussed earlier, the maximum value was 8,192. But this just happened to be true. It could have been different. We can contrast this with a numeric variable such as BIRTH_MONTH, which might have a permissible range of 1-12 and be restricted to integers. For various reasons it is possible that, on a particular data set, not all of these are actual values. But the point is that, if LENGTH 3 is set for this variable, it's hard to imagine a situation where values for this variable would ever create imprecision issues. Of course the variable might have a value of 99 or whatever to indicate "UNKNOWN", but, unlike the dental expenditure variable discussed earlier, the permissible range is well defined and clearly within the reduced LENGTH specification.

For the HOSPITAL_DAYS variable, given the permissible range, and the fact that the only permissible non-integer is 0.5, I would decide to store this variable as LENGTH 3.

One reason to be tempted by a less-conservative approach is the potential saving in storage space. In the next section I will discuss an alternative way of reducing the storage space required for numeric variables. It is important to remember that the only reason to reduce LENGTH with numeric variables is to save disk space. There is no increase in computational efficiency. All numeric variables are expanded to 8 bytes for computations performed in DATA and PROC steps.

COMPRESS= BINARY

In SAS Version 8 one of the options for compressing SAS data sets is COMPRESS= BINARY. This is a compression method that is recommended for numeric data. It is efficient in two ways: (i) it can dramatically reduce storage requirements; (ii) it can decrease the time it takes to read in the data set. You can use this option as follows:

```
(13) DATA TWO (COMPRESS= BINARY);
       SET ONE;
       ....
     RUN;
```

Here's one example of the type of reduction you will find in storage requirements. The 1997 Medical Expenditure Panel Survey [MEPS] public use file HC-020 ("The 1997 Full-Year Consolidated Data File") has 1,215 variables and 34,551 observations. Observation length is 6,544. Almost 99% of the variables are numeric (1,201). Of these, 621 are LENGTH 3 and 580 are LENGTH 8.

Uncompressed the file uses 275 MB of disk space. Compressed with the COMPRESS= BINARY option it uses 43 MB. Also, the number of data set pages is reduced from 17,285 to 2,718. This latter reduction, in part, allows for faster I/O times because the SAS system will usually transfer a complete page in a single I/O operation. So, the fewer pages per data set, the fewer I/O operations required.

For many data sets this creates a win-win situation: you save both storage space and the time it takes to read in a file. I have run numerous tests on a variety of data sets and the following generalization holds:

```
(14) If the COMPRESS= BINARY option reduces
     file size, then SAS takes less time to
     read in the file.
```

SAS gives you informative LOG messages about the results of compression, for example:

```
(15)

NOTE: There were 34551 observations read
      from the data set HC020.P20V4X.
NOTE: The data set DTEST.PUF20 has 34551
      observations and 1215 variables.
NOTE: Compressing data set DTEST.PUF20
      decreased size by 84.28 percent.
      Compressed is 2718 pages; un-
      compressed would require 17285 pages.
```

Obviously COMPRESS= BINARY is an option with lots of advantages. More CPU resource will be required, but this cost is usually more than offset by the benefits.

In some SAS documentation you will see statements such as the following.

```
(16)  In Release 8.2, when a request is
      made to compress a file, SAS
      determines if the compressed file
      will be larger than an uncompressed
      file.  If so, SAS creates an
      uncompressed file and issues a
      warning message that compression is
      not enabled.
```

In a subsequent SAS Note (SN-005687), this general statement is corrected because there are cases where compression will be enabled despite it increasing data set size. As a test of this you can compress a data set with one variable, a character variable of LENGTH 13, and 100,000 observations. The LOG note will let you know that compressing the data set increased file size.

The general rule of thumb is that the effectiveness of compression increases with observation length. The overhead associated with a SAS-compressed data set is 12 bytes per observation, so this must be taken into account when making decisions about compression.

One final advantage of using SAS-internal compression: there is no need for an explicit utility or command to uncompress the data set. SAS will automatically uncompress the file.

The general guidelines for storing SAS numeric variables are given in (17).

```
(17)  a. Distinguish between permissible
         and actual values for a variable
         in a data set.
      b. Use the default LENGTH
         specification of 8 bytes unless
         all the permissible values for a
         variable can be accurately
         represented in fewer bytes.
      c. Use the COMPRESS= BINARY option
         whenever it reduces data set size.
```

These are guidelines, not rules, and (as always) you need to take into account any specific properties of your data set or project that would argue for a different approach.

COMPARISON OPERATORS

In the section on fractions, we have already seen that comparing two numbers can lead to counter-intuitive results if we do not consider the effects of numeric imprecision. In (8), because 1/10 cannot be accurately represented in a base 2 system, we appeared to have LOG output stating that 1 does not equal 1. Obviously this apparent contradiction only existed due to the numeric appearance of the variables. The actual comparison was between the stored representations.

Let's modify (8) as follows:

```
(18) DATA ONE;
       VAR1 = 1 ;
       VAR2 = 0 ;
       DO X = 1 TO 10 ;
              VAR2 + 0.1 ;
       END;
       VAR3 = 0 ;
       DO X = 1 TO 2 ;
              VAR3 + 0.5 ;
       END;
       IF VAR1 = ROUND(VAR2)
              THEN PUT VAR1 'EQ ' VAR2 ;
       ELSE PUT VAR1 'NE ' VAR2 ;
       IF VAR1 = VAR3
              THEN PUT VAR1 'EQ ' VAR3 ;
       ELSE PUT VAR1 'NE ' VAR3 ;
   RUN;
```

If you run this DATA step, the LOG output will be:

```
(19)

       1 EQ 1          [for 0.1]

       1 EQ 1          [for 0.5]
```

The round function corrects for the imprecision by rounding to the nearest integer. In (18) the rounding is within the comparison, the value of VAR2 is unchanged, i.e. exactly what you see in (10). But you may want to actually change the stored value of the variable. If so, then the DATA step in (20) is what you want.

```
(20) DATA ONE;
       VAR1 = 1 ;
       VAR2 = 0 ;
       DO X = 1 TO 10 ;
              VAR2= ROUND((VAR2+0.1),.1) ;
       END;
       VAR3 = 0 ;
       DO X = 1 TO 2 ;
              VAR3 + 0.5 ;
       END;
       IF VAR1 = VAR2
              THEN PUT VAR1 'EQ ' VAR2 ;
       ELSE PUT VAR1 'NE ' VAR2 ;
       IF VAR1 = VAR3
              THEN PUT VAR1 'EQ ' VAR3 ;
       ELSE PUT VAR1 'NE ' VAR3 ;
     RUN;
```

The LOG output of this DATA step will be the same as for (18), but for a different reason. Here the ROUND function is part of the assignment statement and the stored values of VAR1 and VAR2 are equal, as shown in (21).

```
(21)

VAR1= 00111111 11110000 00000000 00000000
      00000000 00000000 00000000 00000000
VAR2= 00111111 11110000 00000000 00000000
      00000000 00000000 00000000 00000000
```

SAS gives you the choice of rounding for the comparison or rounding the actual value of the variable. With the ROUND function you can specify a *round-off-unit* as a second argument that allows you to specify the decimal level for rounding. For example, ROUND(N,.1) rounds to the nearest tenth. If you omit the second argument, as in (18), then SAS rounds to the nearest integer, i.e. it assumes a second argument of 1.

A reasonable guideline is to store the more-precise (unrounded) values for a variable and round for specific purposes. That way if you ever need the more-precise values, you have them. Of course, as is often true in programming, the trick is being able to distinguish between what you see and what you have.

There are situations where the ROUND function itself may be affected by numeric imprecision For a detailed discussion of this issue see TS-230 from SAS Institute.

The CEIL and FLOOR functions may also be useful in specific situations. Both functions return integer values, i.e. unlike ROUND, you cannot specify a second argument indicating the decimal level.

The CEIL function rounds *up* to the nearest integer, i.e. it returns the smallest integer that is greater than or equal to the argument. The FLOOR function rounds *down* to the nearest integer, i.e. it returns the largest integer that is less than or equal to the argument.

NUMERIC DISPLAY FORMATS

In this section I'm going to discuss three common numeric formats: COMMA*w.d*, DOLLAR*w.d* and PERCENT*w.d*. Using these formats is not complicated, but avoiding the frustrating message "At least one W.D format was too small for the number to be printed..." does require knowing, not only your data, but also the particular requirements of the values for *w* and *d*.

In general, *w* is a number that specifies the total width of the field. For the COMMA*w.d* format, the value of *w* must include the total number of commas the highest value of the variable will need. The number 1,000,000,000 requires a field width of 13. If you want to add 2 decimal places you need to specify COMMA16.2 for 1,000,000,000.00. That is, you need to add 3 to the field width, 1 for the decimal point and 2 for the decimal places. You also need to consider the minus sign with negative numbers.

For the DOLLAR*w.d* format, you always need to add 1 to the *w* value for the dollar sign. The DOLLAR*w.d.* format automatically includes commas, so you need to add them into the field width.

For both the COMMA*w.d* and DOLLAR*w.d* formats, the value of *d* must be either 0 or 2 (omitting any specification for *d* is equivalent to specifying 0). Whether you specify 0 or 2 for the *d* value, SAS will round the number if necessary. For example 12.44 will be displayed as 12 if *d* is 0, but 12.99 will be displayed as 13.

A common guideline for displaying numeric data is to display a level of detail that neither obscures relevant information nor displays irrelevant or misleading information. For example, for a particular report it might be necessary to display dollar figures to the penny, for another report it might be sufficient to omit the cents by specifying 0 for the *d* value. But if you are omitting information, even if it is irrelevant to the specific table, it's important to document (e.g. in a footnote) how the displayed values have been rounded. For example, it may be useful for someone reading a report to know that values of 0.49 will appear as 0 because decimal values were omitted.

The PERCENT*w.d.* format operates in generally the same way as the other two, but with a few differences. First, negative values are displayed within parentheses, so you need to add 2 to the width field for these (*even if none of the values you are displaying are negative!*). Of course, you also have to add 1 for the percent sign. The other difference is that the *d* value may be either 0, 1 or 2. Again, the value of *d* will affect the value of *w*.

CONCLUSION

It is important to understand that the display properties of numbers in SAS differ from the stored properties. Being aware of these differences and how they affect computations and reports is an essential part of good SAS programming.

Knowing the basics of how SAS stores numbers leads to more informed decisions regarding LENGTH specifications for numeric variables. The default specification in SAS is also the maximum specification, and this is exactly right. The costs and benefits of reduced numeric length should be carefully considered.

One guideline for storing numbers in SAS data sets is to only reduce length if the set of *permissible* values can be accurately represented in the number of bytes specified. Reducing numeric length does not affect DATA or PROC steps, so the only reason for doing so is to save disk space. For many SAS data sets, considerable saving of disk space can be achieved with the COMPRESS= BINARY option.

Even with a full 8-byte representation, not all numbers can be accurately stored. The ROUND function may be used to correct for this either by rounding the stored values of numeric variables, or by rounding for the specific purpose of a comparison within a DATA step.

SAS provides a variety of formats for displaying numeric data, three of them discussed in this paper. SAS also allows users the option of creating their own formats. SAS formats are extremely useful and extremely powerful. They have the potential to reveal interesting patterns in the data, but also to obscure information. Again, this points to the importance of distinguishing between the appearance of a number on a screen or piece of paper, and how it is represented in the computer.

REFERENCES

"Numeric Precision 101," available from:
http://ftp.sas.com/techsup/download/technote/ts654.pdf.

"Dealing With Numeric Representation Error in SAS Applications," available from:
http://ftp.sas.com/techsup/download/technote/ts230.html

ACKNOWLEDGMENTS

I would like to thank all my SAS-programming colleagues at Social & Scientific Systems, Inc. SSS has a great community of SAS programmers and I've learned a lot being part of it. Our own internal SAS users listserv (SAS_USERS-L) is a great way for us to pose questions and share programming tips that are specific to SSS projects.

CONTACT INFORMATION

Paul Gorrell
Social & Scientific Systems, Inc.
8757 Georgia Avenue, 12th Floor
Silver Spring, MD 20910

Email: pgorrell@s-3.com
Telephone: 301-628-3237 (Office)
301-628-3000 (Main)
FAX: 301-628-3201