Numeric Variable Storage Pattern

Sreekanth Middela Srinivas Vanam Rahul Baddula

Percept Pharma Services, Bridgewater, NJ

ABSTRACT

This paper presents the Storage pattern of Numeric Variables within the SAS system, which helps in understanding the underlying hardware implementation of floating point representation in both Mainframe and Non-Mainframe Systems.

INTRODUCTION

There are 2 types of variables in a SAS dataset namely: Character and Numeric.

Character Variables:

Character Variables are stored as 1 character per byte. Character variables are stored in either ASCII or EBCDIC standards. The length of a Character Variable can range from 1 to 32767.

Example:

The string "Hello World" consists of 11 characters (including space) which require 11 bytes for storing in the SAS system.

Numeric Variables:

Numeric Variables are stored as multiple digits per byte. SAS stores all numeric variables using DOUBLE PRECISION Floating-point representation which is a form of Scientific notation, where values are represented as numbers between 0 and 1 times a power of 10. Scientific notation takes the form of [**m * b ^ e**], where

"m" is the mantissa, which is a number lying between 0 and 1,

"b" is the base,

"e" is the exponent.

Example: The number 1234 can be represented in Scientific notation as [**.1234 * 10 ^ 4**] where,

".1234" is the mantissa,

"10" is the base,

"4" is the exponent.

FLOATING POINT REPRESENTATION

- Floating-point representation is a form of Scientific notation, except that the base is either 2(Non-Mainframes) or 16(Mainframes) instead of 10.
- The default length of a Numeric Variable is 8, which can range from 2 - 8 in Mainframes and 3 - 8 in Non-Mainframes.

DIFFERENCE BETWEEN MAINFRAMES AND NON-MAINFRAMES

	Mainframe Systems	Non-Mainframe Systems
Default Length	8	8
Minimum Length	2	3
Maximum Length	8	8
Base	16	2
No. of Sign Bits	1(Left most bit)	1(Left most bit)
No. of Exponent Bits	7 bits	11 bits
No. of Mantissa Bits	56	52

In the following sections, we clearly describe the storage in Mainframes and Non-mainframes.

FLOATING-POINT REPRESENTATION IN MAINFRAMES

In Mainframe systems, Out of the 64 bits, the left most bit is considered as Sign bit, and the next 7 bits are considered as Characteristic (Exponent) bits and the following 56 bits are considered as Mantissa bits.

SEEEEEEE	MMMMMMMM	MMMMMMMM	...	...	...	...	MMMMMMMM
(1, 2-8)	(9-16)	(17-24)		…			(57-64)

The value of Sign bit is 0 if the number is positive and 1 if it is negative. The Characteristic (7 bits) is the signed exponent which is obtained by adding the bias to the actual exponent. The bias is an offset used to enable both positive and negative exponents with the bias representing 0. For the mainframes, bias is 64. If the bias is not used, then an additional sign bit must be allocated for the exponent.

Example:

Actual Exponent = 2, Characteristic = bias + exponent = 64 + 2 = 66.

Actual Exponent = -3, Characteristic = bias + exponent = 64 - 3 = 61.

There is an implied radix point before the left most bit of the mantissa, therefore mantissa is always less than 1. The radix point can be thought of as a generic form of decimal point.

Example:

In the decimal number 3▪25, ▪ Is a decimal point.

In its binary equivalent 11▪01(3▪25 in decimal), the ▪ is a radix point. The conversion of decimal 3.25 into binary 11.01 is discussed in the [**Example:** Number [3.25]] section later in this paper.

The exponent has a base associated with it, which is 16(hexadecimal) in mainframes and 2(binary) in non-mainframes. The exponent is used to determine how many times the base should be multiplied by the mantissa. So the generic form of Scientific notation m * b ^ e takes m * 16 ^ e in Mainframe systems and m * 2 ^ e in Non-Mainframe systems.

Each bit in the Mantissa represents a fraction whose numerator is 1 and denominator is a power of 2. The left most bit in the Mantissa represents 1/2 and the next bit represents (1/2)^2 and so on until the last bit of mantissa. In other words, the mantissa is the sum of series of fractions such as 1/2, 1/4, 1/8 and so on.

Example: Let us consider an example on how the number 100 is stored in the Mainframe systems.

- Consider the number 100 and convert it into hexadecimal form which results in 64.
- **Example: 64(hexadecimal) = 6*16^1 + 4*16^0 = 6*16 + 4*1 = 96 + 4 = 100(decimal).**
- Then normalize it to have the radix point on the left of the number, which results in ▪64 * 16 ^ 2. (Since the radix point is moved by 2 places on the left, the mantissa becomes as ▪64 and base is 16(mainframes) and 2 will be the exponent).
- Now the Sign bit will be 0 since 100 is a positive number.
- Then the Characteristics bits are calculated as the sum of bias and exponent, which is 64 + 2 = 66. So 66 when represented in binary comes as [100 0010].
- Then the mantissa is calculated as the binary form of hexadecimal 64, which will be, [0110 0100].
- The left over bits for mantissa are padded with zeros.
- So the number 100 is stored in as,

Sign bit(1)	0
Characteristic bits(7)	100 0010
Mantissa bits(56)	01100010 {00000000}*6

Programmatically checking value of 100 gives the following output:

```
data _null_ ;
    x = 100 ;
    put "The binary equivalent of " x " is " x binary64. ;
run;
```

The following output will be printed In the Log:

```
The binary equivalent of 100 is
0100001001100010000000000000000000000000000000000000000000000000
NOTE: DATA statement used (Total process time):
    real time           0.15 seconds
    cpu time            0.01 seconds
```

We can also track the decimal equivalent from the binary sequence as follows:

- From the available 64 bits, consider the left most bit as Sign bit, which is 0, that means the resultant number is a positive number.
- Then the next 7 bits are considered as Characteristic bits, which is 66. Subtract the bias from this number to get the exponent, which is 66-64 = 2.
- As discussed previously, Mantissa is the sum of series of numbers 1/2, 1/4, 1/8 and so on. For this binary sequence mantissa comes as, [(1/2)^2 + (1/2)^3 + (1/2)^6] * 16 ^ 2, because we have a binary 1 at 2nd, 3rd and 6th places in mantissa.
- The result of above expression comes as 100.

FLOATING POINT REPRESENTATION IN NON-MAINFRAMES

Non-mainframes are very similar to mainframes except the following differences,

- There are 11 bits allocated for Characteristic instead of 7, which reduces the number of mantissa bits to 52.
- The base is 2 instead of 16.
- The bias is 1023 instead of 64.
- There is a single hidden bit which is not present in mainframes.

SEEEEEEE	EEEE MMMM	MMMMMMMM	...	...	...	...	MMMMMMMM
(1, 2-8)	(9-12 13-16)	(17-24)		…			(57-64)

Example:

Actual Exponent = 2, Characteristic = bias + exponent = 1023 + 2 = 1025.

Actual Exponent = -3, Characteristic = bias + exponent = 1023 - 3 = 1020.

Example: Let us consider an example on how the number 100 is stored in the Mainframe systems.

- Consider the number 100 and convert it into binary form which results in 1100100.
- Then normalize it to have the radix point on the left of the number, which results in .1100100 * 2 ^ 7. (Since the radix point is moved by 7 places on the left, the mantissa becomes as .1100100 and base is 2(non-mainframes) and 6 will be the exponent (since we subtract the hidden bit from 7)).
- Now the Sign bit will be 0 since 100 is a positive number.
- Then the Characteristics bits are calculated as the sum of bias and exponent, which is 1023 + 6 = 1029. So 1029 when represented in binary comes as [100 0000 0101].
- Now the mantissa becomes [1100 100 ...].
- The left over bits for mantissa are padded with zeros.
- So the number 100 is stored as,

Sign bit(1)	0
Characteristic bits(11)	100 0000 0101
Mantissa bits(56)	11001000 {00000000}*6

Programmatically checking value of 100 gives the following output:

```
data _null_ ;
    x = 100 ;
    put "The binary equivalent of " x " is " x binary64. ;
run;
```

The following output will be printed In the Log:

```
The binary equivalent of 100 is
0100001001100010000000000000000000000000000000000000000000000000
NOTE: DATA statement used (Total process time):
    real time           0.15 seconds
    cpu time            0.01 seconds
```

We can also track the decimal equivalent from the binary sequence as follows:

- From the available 64 bits, consider the left most bit as Sign bit, which is 0, that means the resultant number is a positive number.
- Then the next 11 bits are considered as Characteristic bits, which is 1029. Subtract the bias from this number to get the exponent, which is 1029-1023 = 6. Since there will be a hidden bit, the exponent becomes 6+1=7.
- Then consider the Mantissa bits from bit 13 in the above sequence which comes as 11001000 00000000 and so on. Since the exponent is 7, keep a radix point after 7 binary digits, which comes as [1100100.000000000]. When this sequence is converted into decimal we get the number 100.0 which is simply 100.

Example: Number [58].

Convert 58 to binary and that would result in 111010

- Then normalize it to have the radix point on the left of the number, which results in .110010 * 2 ^ 6. (Since the radix point is moved by 6 places on the left, the mantissa becomes as

  .110010 and base is 2(non-mainframes) and 5 will be the exponent (since we subtract the hidden bit from 6)).
- Now the Sign bit will be 0 since 58 is a positive number.
- Then the Characteristics bits are calculated as the sum of bias and exponent, which is 1023 + 5 = 1028. So 1028 when represented in binary comes as [10000000100].
- Now the mantissa becomes [1100 10. ...].
- The left over bits for mantissa are padded with zeros.
- So the number 58 is stored in as,

Sign bit(1)	0
Characteristic bits(11)	100 0000 0101
Mantissa bits(56)	110010 00 {00000000}*6

Programmatically checking value of 58 gives the following output:

```
data _null_ ;
    x = 58 ;
    put "The binary equivalent of " x " is " x binary64. ;
run;
```

The following output will be printed In the Log:

```
The binary equivalent of 58 is
0100000001001101000000000000000000000000000000000000000000000000
NOTE: DATA statement used (Total process time):
    real time           0.15 seconds
    cpu time            0.01 seconds
```

Example: Number [-58].

Everything will be same as the number 58 with the only difference as sign bit will be 1.

Example: Number [3▪25].

Conversion of decimal 3.25 into binary: The number 3 in binary is 11. The number 0.25 in binary can be obtained as follows:

0▪25 * 2 → 0▪5 – 0 (Remainder is considered 0 and fraction part is used in next step).

0▪5 * 2 → 1▪0 – 1 (Remainder is considered 1 and since the fraction part is 0, calculation no longer continues).

Hence when the bits are read top-down ▪01 becomes the binary equivalent of 0▪25. So, the binary equivalent of 3▪25 is 11▪01.

- Then normalize it to have the radix point on the left of the number, which results in ▪1101 * 2 ^ 2. (Since the radix point is moved by 2 places on the left, the mantissa becomes as ▪1101 and base is 2(non-mainframes) and 1 will be the exponent (since we subtract the hidden bit from 2)).
- Now the Sign bit will be 0 since 3.25 is a positive number.
- Then the Characteristics bits are calculated as the sum of bias and exponent, which is 1023 + 1 = 1024. So 1024 when represented in binary comes as [10000000000].
- Now the mantissa becomes [1101 ...].
- The left over bits for mantissa are padded with zeros.
- So the number 3.25 is stored as,

Sign bit(1)	0
Characteristic bits(11)	100 0000 0000
Mantissa bits(56)	1101 {00000000}*6

Programmatically checking value of 3.25 gives the following output:

```
data _null_ ;
    x = 3.25 ;
    put "The binary equivalent of " x " is " x binary64. ;
run;
```

The following output will be printed In the Log:

```
The binary equivalent of 3.25 is
0100000000001010000000000000000000000000000000000000000000000000
NOTE: DATA statement used (Total process time):
    real time           0.15 seconds
    cpu time            0.01 seconds
```

Example: Number [1.2].

Conversion of decimal 1.2 into binary: The number 1 in binary is 1. The number 0.2 in binary can be obtained as follows:

0.2 * 2 → 0.4 – 0

0.4 * 2 → 0.8 – 0

0.8 * 2 → 1.6 --1 (Remainder is considered 1 and fraction part is used in next step).

0.6 * 2 → 1.2 – 1 (Remainder is considered 1 and fraction part is used in next step).

0.2 * 2 → 0.4 .This sequence is repeated infinite number of times.

Hence when the bits are read top-down .001100110011… becomes the binary equivalent of 0.2. So, the binary equivalent of 1.2 is 1.0011 0011….

- Then normalize it to have the radix point on the left of the number, which results in .100110011 * 2 ^ 1. (Since the radix point is moved by 1 place on the left, the mantissa becomes as .10011 and base is 2(non-mainframes) and 0 will be the exponent (since we subtract the hidden bit from 1)).
- Now the Sign bit will be 0 since 1.2 is a positive number.
- Then the Characteristics bits are calculated as the sum of bias and exponent, which is 1023 + 0 = 1023. So 1023 when represented in binary comes as [01111111111].
- Now the mantissa becomes [10011 ...].

- The left over bits for mantissa are padded with 0011 sequence.
- So the number 1.2 is stored as,

Sign bit(1)	0
Characteristic bits(11)	011 1111 1111
Mantissa bits(56)	10011 0011 …

Note that when we try to convert back to decimal there might be a significant approximation needed since there is a repetitive sequence.

Programmatically checking value of 100 gives the following output:

```
data _null_ ;
    x = 1.2 ;
    put "The binary equivalent of " x " is " x binary64. ;
run;
```

The following output will be printed In the Log:

```
The binary equivalent of 1.2 is
0011111111110011001100110011001100110011001100110011001100110011
NOTE: DATA statement used (Total process time):
    real time           0.15 seconds
    cpu time            0.01 seconds
```

TRUNC() FUNCTION

This function truncates a numeric value to a specified number of bytes.

Syntax: TRUNC(number,length)

The TRUNC function truncates a full-length number (stored as double) to a smaller number of bytes, as specified in length and pads the truncated bytes with 0s. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them.

If the TRUNC function returns the same number with the length specified then that means the specified number of bytes in length is sufficient to store the number.

Example:

TRUNC(3,8) = 3 : 8 is sufficient length for 3.

TRUNC(8193,3) = 8192 : 3 is not sufficient length for 8193.

REFERENCES

1. "***Numeric Precision in SAS Software***" from SAS website. [http://support.sas.com/documentation/cdl/en/lrcon/62955/HTML/default/viewer.htm#a000695157.htm].

2. "***Numeric Length in SAS: A Case Study in Decision Making***" by Paul Gorrell.

ACKNOWLEDGEMENTS

- Percept Pharma Services (www.perceptservices.com)
  1031, US Highway 22W, Suite 304,
  Bridgewater, NJ. 08807

CONTACT INFORMATION

Your comments and suggestions are valued and encouraged. Contact the authors at:

Sreekanth Middela (sreekanth.middela@perceptservices.com)

Srinivas Vanam (srinivas.vanam@gmail.com)

Rahul Baddula (rahul.baddula@gmail.com)

TRADEMARK INFORMATION