CHAPTER

10

SAS Variables

Definitions

variables
are containers that you create within a program to store and use character and numeric values. Variables have *attributes*, such as *name* and *type*, that enable you to identify them and that define how they can be used.

character variables
are variables of type *character* that contain alphabetic characters, numeric digits 0 through 9, and other special characters.

numeric variables
are variables of type *numeric* that are stored as floating-point numbers, including dates and times.

numeric precision
refers to the degree of accuracy with which numeric variables are stored in your operating environment.

Variable Attributes

A SAS variable has the attributes that are listed in the following table:

Table 10.1 Variable Attributes

Variable Attribute	Possible Values	Default Value
Name	Any valid SAS name. See Chapter 3, "Rules for Words and Names," on page 15.	None
Type [1]	Numeric, character	Numeric
Length [1]	2 to 8 bytes [2] 1 to 32,767 bytes for character	8 bytes for numeric, character
Format	See Chapter 5, "Formats," on page 27.	BEST12. for numeric, $*w*. for character
Informat	See Chapter 7, "Informats," on page 65.	*w.d* for numeric, $*w*.for character
Label	Up to 256 characters	None
Position in observation	1- *n*	NA
Index type	NONE, SIMPLE, COMPOSITE, or BOTH.	NA

1 If not explicitly defined, a variable's type and length are implicitly defined by its first occurrence in a DATA step.
2 The minimum length is 2 bytes in some operating environments, 3 in others. See the SAS documentation for your operating environment.

You can use the CONTENTS procedure, or the functions that are named in the following definitions, to obtain information about a variable's attributes:

Name
: identifies a variable. A variable name must conform to SAS naming rules. A SAS name can be up to 32 characters long. The first character must be a letter (A, B, C, . . . , Z) or underscore (_). Subsequent characters can be letters, digits (0 to 9), or underscores. Note that blanks are not allowed. Mixed case variables are allowed. See Chapter 3, "Rules for Words and Names," on page 15 for more details on mixed case variables.

  The names _N_, _ERROR_, _FILE_, _INFILE_, _MSG_, _IORC_, and _CMD_ are reserved for the variables that are generated automatically for a DATA step. Note that SAS products use variable names that start and end with an underscore; it is recommended that you do not use names that start and end with an underscore in your own applications. See "Automatic Variables" on page 107 for more information.

  To determine the value of this attribute, use the VNAME or VARNAME function.

  Note: The rules for variable names that are described in this section apply when the VALIDVARNAME= system option is set to VALIDVARNAME= V7, the default setting. Other rules apply when this option is set differently. See Chapter 3, "Rules for Words and Names," on page 15 for more information. △

Type
: identifies a variable as numeric or character. Within a DATA step, a variable is assumed to be numeric unless character is indicated. Numeric values represent numbers, can be read in a variety of ways, and are stored in floating-point format. Character values can contain letters, numbers, and special characters and can be from 1 to 32,767 characters long.

  To determine the value of this attribute, use the VTYPE or VARTYPE function.

Length
: refers to the number of bytes used to store each of the variable's values in a SAS data set. You can use a LENGTH statement to set the length of both numeric and character variables. Variable lengths specified in a LENGTH statement affect the length of numeric variables only in the output data set; during processing, all numeric variables have a length of 8. Lengths of character variables specified in a LENGTH statement affect both the length during processing and the length in the output data set.

  In an INPUT statement, you can assign a length other than the default to character variables. You can also assign a length to a variable in the ATTRIB statement. A variable that appears for the first time on the left side of an assignment statement has the same length as the expression on the right side of the assignment statement.

  To determine the value of this attribute, use the VLENGTH or VARLEN function.

Format
: refers to the instructions that SAS uses when printing variable values. If no format is specified, the default format is BEST12. for a numeric variable, and $*w*. for a character variable. You can assign SAS formats to a variable in the FORMAT or ATTRIB statement. You can use the FORMAT procedure to create your own format for a variable.

  To determine the value of this attribute, use the VFORMAT or VARFMT function.

Informat
: refers to the instructions that SAS uses when reading data values. If no informat is specified, the default informat is *w.d* for a numeric variable, and $*w*. for a character variable. You can assign SAS informats to a variable in the INFORMAT

or ATTRIB statement. You can use the FORMAT procedure to create your own informat for a variable.

To determine the value of this attribute, use the VINFORMAT or VARINFMT function.

Label
refers to a descriptive label up to 256 characters long. A variable label, which can be printed by some SAS procedures, is useful in report writing. You can assign a label to a variable with a LABEL or ATTRIB statement.

To determine the value of this attribute, use the VLABEL or VARLABEL function.

Position in observation
is determined by the order in which the variables are defined in the DATA step. You can find the position of a variable in the observations of a SAS data set by using the CONTENTS procedure. This attribute is generally not important within the DATA step except in variable lists, such as the following:

```
var rent--phone;
```

See “SAS Variable Lists” on page 108 for more information.

The positions of variables in a SAS data set affect the order in which they appear in the output of SAS procedures, unless you control the order within your program, for example, with a VAR statement.

To determine the value of this attribute, use the VARNUM function.

Index type
indicates whether the variable is part of an index for the data set. See “SAS Indexes” on page 433 for more information.

To determine the value of this attribute, use the OUT= option with the CONTENTS procedure to create an output data set. The IDXUSAGE variable in the output data set contains one of the following values for each variable:

Table 10.2 Index Type Attribute Values

Value	Definition
NONE	The variable is not indexed.
SIMPLE	The variable is part of a simple index.
COMPOSITE	The variable is part of one or more composite indexes.
BOTH	The variable is part of both simple and composite indexes.

Creating Variables

Ways to Create Variables

You can create variables in a DATA step in the following ways:

- by using an assignment statement
- by reading data with the INPUT statement in a DATA step
- by specifying a new variable in a FORMAT or INFORMAT statement
- by specifying a new variable in a LENGTH statement

□ by specifying a new variable in an ATTRIB statement.

Note: You can also create variables with the FGET function. See *SAS Language Reference: Dictionary* for more information. △

Using an Assignment Statement

In a DATA step, you can create a new variable and assign it a value by using it for the first time on the left side of an assignment statement. SAS determines the length of a variable from its first occurrence in the DATA step. The new variable gets the same type and length as the expression on the right side of the assignment statement.

When the type and length of a variable are not explicitly set, SAS gives the variable a default type and length as shown in the examples in the following table.

Table 10.3 Resulting Variable Types and Lengths Produced When Not Explicitly Set

Expression	Example	Resulting Type of X	Resulting Length of X	Explanation
Numeric variable	`length a 4;` `x=a;`	Numeric variable	8	Default numeric length (8 bytes unless otherwise specified)
Character variable	`length a $ 4;` `x=a;`	Character variable	4	Length of source variable
Character literal	`x='ABC';` `x='ABCDE';`	Character variable	3	Length of first literal encountered

Concatenation of variables	`length a $ 4`
	`b $ 6`
	`c $ 2;`
	`x=a||b||c;`
		Character variable	12	Sum of the lengths of all variables

Concatenation of variables and literal	`length a $ 4;`
	`x=a||'CAT';`
	`x=a||'CATNIP';`
		Character variable	7	Sum of the lengths of variables and literals encountered in first assignment statement

If a variable appears for the first time on the right side of an assignment statement, SAS assumes that it is a numeric variable and that its value is missing. If no later statement gives it a value, SAS prints a note in the log that the variable is uninitialized.

Note: A RETAIN statement initializes a variable and can assign it an initial value, even if the RETAIN statement appears after the assignment statement. △

Reading Data with the INPUT Statement in a DATA Step

When you read raw data in SAS by using an INPUT statement, you define variables based on positions in the raw data. You can use one of the following methods with the INPUT statement to provide information to SAS about how the raw data is organized:

- column input
- list input (simple or modified)
- formatted input
- named input.

See *SAS Language Reference: Dictionary* for more information about using each method.

The following example uses simple list input to create a SAS data set named GEMS and defines four variables based on the data provided:

```
data gems;
   input Name $ Color $ Carats Owner $;
   datalines;
emerald green 1 smith
sapphire blue 2 johnson
ruby red 1 clark
;
```

Specifying a New Variable in a FORMAT or an INFORMAT Statement

You can create a variable and specify its format or informat with a FORMAT or an INFORMAT statement. For example, the following FORMAT statement creates a variable named Sale_Price with a format of 6.2 in a new data set named SALES:

```
data sales;
   Sale_Price=49.99;
   format Sale_Price 6.2;
run;
```

SAS creates a numeric variable with the name Sale_Price and a length of 8.

See *SAS Language Reference: Dictionary* for more information about using the FORMAT and INFORMAT statements.

Specifying a New Variable in a LENGTH Statement

You can use the LENGTH statement to create a variable and set the length of the variable, as in the following example:

```
data sales;
   length Salesperson $20;
run;
```

For character variables, you must allow for the longest possible value in the first statement that uses the variable, because you cannot change the length with a subsequent LENGTH statement within the same DATA step. The maximum length of any character variable in the SAS System is 32,767 bytes. For numeric variables, you can change the length of the variable by using a subsequent LENGTH statement.

When SAS assigns a value to a character variable, it pads the value with blanks or truncates the value on the right side, if necessary, to make it match the length of the target variable. Consider the following statements:

```
length address1 address2 address3 $ 200;
address3=address1||address2;
```

Because the length of ADDRESS3 is 200 bytes, only the first 200 bytes of the concatenation (the value of ADDRESS1) are assigned to ADDRESS3. You might be able to avoid this problem by using the TRIM function to remove trailing blanks from ADDRESS1 before performing the concatenation, as follows:

```
address3=trim(address1)||address2;
```

See *SAS Language Reference: Dictionary* for more information about using the LENGTH statement.

Specifying a New Variable in an ATTRIB Statement

The ATTRIB statement enables you to specify one or more of the following variable attributes for an existing variable:

- FORMAT=
- INFORMAT=
- LABEL=
- LENGTH=.

If the variable does not already exist, one or more of the FORMAT=, INFORMAT=, and LENGTH= attributes can be used to create a new variable. For example, the following DATA step creates a variable named Flavor in a data set named LOLLIPOPS:

```
data lollipops;
   Flavor="Cherry";
   attrib Flavor format=$10.;
run;
```

Note: You cannot create a new variable by using a LABEL statement or the ATTRIB statement's LABEL= attribute by itself; labels can only be applied to existing variables. △

See *SAS Language Reference: Dictionary* for more information about using the ATTRIB statement.

Using the IN= Data Set Option

The IN= data set option creates a special boolean variable that indicates whether the data set contributed data to the current observation. The variable has a value of 1 when true, and a value of 0 when false. You can use IN= on the SET, MERGE, and UPDATE statements in a DATA step.

The following example shows a merge of the OLD and NEW data sets where the IN= option is used to create a variable named X that indicates whether the NEW data set contributed data to the observation:

```
data master missing;
   merge old new(in=x);
   by id;
   if x=0 then output missing;
   else output master;
run;
```

Variable Type Conversions

If you define a numeric variable and assign the result of a character expression to it, SAS tries to convert the character result of the expression to a numeric value and to execute the statement. If the conversion is not possible, SAS prints a note to the log,

assigns the numeric variable a value of missing, and sets the automatic variable _ERROR_ to 1. For a listing of the rules by which SAS automatically converts character variables to numeric variables and vice-versa, see "Automatic Numeric-Character Conversion" on page 136.

If you define a character variable and assign the result of a numeric expression to it, SAS tries to convert the numeric result of the expression to a character value using the BEST*w*. format, where *w* is the width of the character variable and has a maximum value of 32. SAS then tries to execute the statement. If the character variable you use is not long enough to contain a character representation of the number, SAS prints a note to the log and assigns the character variable asterisks. If the value is too small, SAS provides no error message and assigns the character variable the character zero (0).

Output 10.1 Automatic Variable Type Conversions (partial SAS log)

```
4
5          data _null_;
6            x= 3626885;
7            length y $ 4;
8            y=x;
9            put y;

36E5
NOTE: Numeric values have been converted to character
      values at the places given by:
      (Number of times) at (Line):(Column).
      1 at 8:5

10         data _null_;
11           xl= 3626885;
12           length yl $ 1;
13           yl=xl;
14           xs=0.000005;
15           length ys $ 1;
16           ys=xs;
17           put yl= ys=;
18         run;

NOTE: Invalid character data, XL=3626885.00 ,
      at line 13 column 6.
YL=* YS=0
XL=3626885 YL=* XS=5E-6 YS=0 _ERROR_=1 _N_=1
NOTE: Numeric values have been converted
      to character values at the places
      given by: (Number of times) at
      (Line):(Column).
      1 at 13:6
      1 at 16:6
```

In the first DATA step of the example, SAS is able to fit the value of Y into a 4-byte field by representing its value in scientific notation. In the second DATA step, SAS cannot fit the value of YL into a 1-byte field and displays an asterisk (*) instead.

Aligning Variable Values

In SAS, numeric variables are automatically aligned. You can further control their alignment by using a format.

However, when you assign a character value in an assignment statement, SAS stores the value as it appears in the statement and does not perform any alignment. Output

10.2 on page 107 illustrates the character value alignment produced by the following program:

```
data aircode;
   input city $1-13;
   length airport $ 10;
   if city='San Francisco' then airport='SFO';
      else if city='Honolulu' then airport='HNL';
      else if city='New York' then airport='JFK or EWR';
      else if city='Miami' then airport='   MIA    ';
   datalines;
San Francisco
Honolulu
New York
Miami
;

proc print data=aircode;
run;
```

This example produces the following output:

Output 10.2 Output from the PRINT Procedure

```
                    The SAS System

            OBS    CITY              AIRPORT

             1     San Francisco     SFO
             2     Honolulu          HNL
             3     New York          JFK or EWR
             4     Miami                MIA
```

Automatic Variables

Automatic variables are created automatically by the DATA step or by DATA step statements. These variables are added to the program data vector but are not output to the data set being created. The values of automatic variables are retained from one iteration of the DATA step to the next, rather than set to missing.

Automatic variables that are created by specific statements are documented with those statements. For examples, see the BY statement, the MODIFY statement, and the WINDOW statement in *SAS Language Reference: Dictionary*.

Two automatic variables are created by every DATA step: _N_ and _ERROR_.

_N_
is initially set to 1. Each time the DATA step loops past the DATA statement, the variable _N_ is incremented by 1. The value of _N_ represents the number of times the DATA step has iterated.

ERROR
is 0 by default but is set to 1 whenever an error is encountered, such as an input data error, a conversion error, or a math error, as in division by 0 or a floating

point overflow. You can use the value of this variable to help locate errors in data records and to print an error message to the SAS log.

For example, either of the two following statements writes to the SAS log, during each iteration of the DATA step, the contents of an input record in which an input error is encountered:

```
if _error_=1 then put _infile_;

if _error_ then put _infile_;
```

SAS Variable Lists

Definition

A SAS *variable list* is an abbreviated method of referring to a list of variable names. SAS allows you to use the following variable lists:

- numbered range lists
- name range lists
- name prefix lists
- special SAS name lists.

With the exception of the numbered range list, you refer to the variables in a variable list in the same order that SAS uses to keep track of the variables. SAS keeps track of active variables in the order that the compiler encounters them within a DATA step, whether they are read from existing data sets, an external file, or created in the step. In a numbered range list, you can refer to variables that were created in any order, provided that their names have the same prefix.

You can use variable lists in many SAS statements and data set options, including those that define variables. However, they are especially useful *after* you define all of the variables in your SAS program because they provide a quick way to reference existing groups of data.

Note: Only the numbered range list is allowed in the RENAME= option. △

Numbered Range Lists

Numbered range lists require you to have a series of variables with the same name, except for the last character or characters, which are consecutive numbers. For example, the following two lists refer to the same variables:

```
x1,x2,x3,...,xn

x1-xn
```

In a numbered range list, you can begin with any number and end with any number as long as you do not violate the rules for user-supplied variable names and the numbers are consecutive.

For example, suppose you decide to give some of your numeric variables sequential names, as in VAR1, VAR2, and so on. Then, you can write an INPUT statement as follows:

```
input idnum name $ var1-var3;
```

Note that the character variable NAME is not included in the abbreviated list.

Name Range Lists

Name range lists rely on the position of variables in the program data vector, as shown in the following table:

Table 10.4 Name Range Lists

This variable list …	includes …
x--a	all variables ordered as they are in the program data vector, from X to A inclusive.
x-numeric-a	all numeric variables from X to A inclusive.
x-character-a	all character variables from X to A inclusive.

For example, consider the following INPUT statement:

```
input idnum name $ weight pulse chins;
```

In later statements you can use these variable lists:

```
/* keeps only the numeric variables idnum, weight, and pulse */

keep idnum-numeric-pulse;


/* keeps the consecutive variables name, weight, and pulse */

keep name--pulse;
```

Name Prefix Lists

Some SAS functions and statements allow you to use a name prefix list to refer to all variables that begin with a specified character string:

```
sum(of SALES:)
```

tells SAS to calculate the sum of all the variables that begin with “SALES,” such as SALES_JAN, SALES_FEB, and SALES_MAR.

Special SAS Name Lists

Special SAS name lists include

NUMERIC
: specifies all numeric variables that are already defined in the current DATA step.

CHARACTER
: specifies all character variables that are currently defined in the current DATA step.

ALL
: specifies all variables that are currently defined in the current DATA step.

Dropping, Keeping, and Renaming Variables

Using Statements or Data Set Options

The DROP, KEEP, and RENAME statements or the DROP=, KEEP=, and RENAME= data set options control which variables are processed or output during the DATA step. You can use one or a combination of these statements and data set options to achieve the results you want. The action taken by SAS depends largely on whether you

- use a statement or data set option or both
- specify the data set options on an input or an output data set.

The following table summarizes the general differences between the DROP, KEEP, and RENAME statements and the DROP=, KEEP=, and RENAME= data set options.

Table 10.5 Statements versus Data Set Options for Dropping, Keeping, and Renaming Variables

Statements ...	Data Set Options ...
apply to output data sets only.	apply to output or input data sets.
affect all output data sets.	affect individual data sets.
can be used in DATA steps only.	can be used in DATA steps and PROC steps.
can appear anywhere in DATA steps.	must immediately follow the name of each data set to which they apply.

Using the Input or Output Data Set

You must also consider whether you want to drop, keep, or rename the variable before it is read into the program data vector or as it is written to the new SAS data set. If you use the DROP, KEEP, or RENAME statement, the action always occurs as the variables are written to the output data set. With SAS data set options, where you use the option determines when the action occurs. If the option is used on an input data set, the variable is dropped, kept, or renamed before it is read into the program data vector. If used on an output data set, the data set option is applied as the variable is written to the new SAS data set. (In the DATA step, an input data set is one that is specified in a SET, MERGE, or UPDATE statement. An output data set is one that is specified in the DATA statement.) Consider the following facts when you make your decision:

- If variables are not written to the output data set and they do not require any processing, using an input data set option to exclude them from the DATA step is more efficient.
- If you want to rename a variable before processing it in a DATA step, you must use the RENAME= data set option in the input data set.
- If the action applies to output data sets, you can use either a statement or a data set option in the output data set.

The following table summarizes the action of data set options and statements when they are specified for input and output data sets. The last column of the table tells whether the variable is available for processing in the DATA step. If you want to rename the variable, use the information in the last column.

Table 10.6 Status of Variables and Variable Names When Dropping, Keeping, and Renaming Variables

Where Specified	Data Set Option or Statement	Purpose	Status of Variable or Variable Name
Input data set	DROP= KEEP=	includes or excludes variables from processing	if excluded, variables are not available for use in DATA step
	RENAME=	changes name of variable before processing	use new name in program statements and output data set options; use old name in other input data set options
Output data set	DROP, KEEP	specifies which variables are written to all output data sets	all variables available for processing
	RENAME	changes name of variables in all output data sets	use old name in program statements; use new name in output data set options
	DROP= KEEP=	specifies which variables are written to individual output data sets	all variables are available for processing
	RENAME=	changes name of variables in individual output data sets	use old name in program statements and other output data set options

Order of Application

If your program requires that you use more than one data set option or a combination of data set options and statements, it is helpful to know that SAS drops, keeps, and renames variables in the following order:

- First, options on input data sets are evaluated left to right within SET, MERGE, and UPDATE statements. DROP= and KEEP= options are applied before the RENAME= option.
- Next, DROP and KEEP statements are applied, followed by the RENAME statement.
- Finally, options on output data sets are evaluated left to right within the DATA statement. DROP= and KEEP= options are applied before the RENAME= option.

Examples of Dropping, Keeping, and Renaming Variables

The following examples show specific ways to handle dropping, keeping, and renaming variables:

- This example uses the DROP= and RENAME= data set options and the INPUT function to convert the variable POPRANK from character to numeric. The name POPRANK is changed to TEMPVAR before processing so that a new variable POPRANK can be written to the output data set. Note that the variable TEMPVAR is dropped from the output data set and that the new name TEMPVAR is used in the program statements.

```
data newstate(drop=tempvar);
   length poprank 8;
   set state(rename=(poprank=tempvar));
   poprank=input(tempvar,8.);
run;
```

- This example uses the DROP statement and the DROP= data set option to control the output of variables to two new SAS data sets. The DROP statement applies to both data sets, CORN and BEAN. You must use the RENAME= data set option to rename the output variables BEANWT and CORNWT in each data set.

```
data corn(rename=(cornwt=yield) drop=beanwt)
     bean(rename=(beanwt=yield) drop=cornwt);
   set harvest;
   if crop='corn' then output corn;
   else if crop='bean' then output bean;
   drop crop;
run;
```

- This example shows how to use data set options in the DATA statement and the RENAME statement together. Note that the new name QTRTOT is used in the DROP= data set option.

```
data qtr1 qtr2 ytd(drop=qtrtot);
   set ytdsales;
   if qtr=1 then output qtr1;
   else if qtr=2 then output qtr2;
   else output ytd;
   rename total=qtrtot;
run;
```

Numeric Precision

Floating-Point Representation

To store numbers of large magnitude and to perform computations that require many digits of precision to the right of the decimal point, SAS stores all numeric values using *floating-point*, or real binary, representation. Floating-point representation is an implementation of what is generally known as scientific notation, in which values are represented as numbers between 0 and 1 times a power of 10. The following is an example of a number in scientific notation:

$$.1234 \times 10^{4}$$

Numbers in scientific notation are comprised of the following parts:

- The *base* is the number raised to a power; in this example, the base is 10.
- The *mantissa* is the number multiplied by the base; in this example, the mantissa is .1234.
- The *exponent* is the power to which the base is raised; in this example, the exponent is 4.

Floating-point representation is a form of scientific notation, except that on most operating systems the base is not 10, but is either 2 or 16. The following table summarizes various representations of floating-point numbers that are stored in 8 bytes.

Table 10.7 Summary of Floating-Point Numbers Stored in 8 Bytes

Representation	Base	Exponent Bits	Maximum Mantissa Bits
IBM mainframe	16	7	56
OpenVMS VAX	2	8	56
IEEE	2	11	52

SAS allows for truncated floating-point numbers via the LENGTH statement, which reduces the number of mantissa bits. For more information on the effects of truncated lengths, see "Storing Numbers with Less Precision" on page 117.

In most situations, the way that SAS stores numeric values does not affect you as a user. However, floating-point representation can account for anomalies you might notice in SAS program behavior. The following sections identify the types of problems that can occur in various operating environments and how you can anticipate and avoid them.

Floating-Point Representation on IBM Mainframes

Floating-point representations are not necessarily related to a single operating system. IBM mainframe operating environments (OS/390 and CMS) all use the same representation made up of 8 bytes as follows:

```
SEEEEEEE MMMMMMMM MMMMMMMM MMMMMMMM
byte 1   byte 2   byte 3   byte 4

MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
byte 5   byte 6   byte 7   byte 8
```

This representation corresponds to bytes of data with each character being 1 bit, as follows:

- The S in byte 1 is the *sign bit* of the number. A value of 0 in the sign bit is used to represent positive numbers.
- The seven E characters in byte 1 represent a binary integer known as the *characteristic*. The characteristic represents a signed exponent and is obtained by adding the bias to the actual exponent. The *bias* is an offset used to allow for both negative and positive exponents with the bias representing 0. If a bias is not used, an additional sign bit for the exponent must be allocated. For example, if a system employs a bias of 64, a characteristic with the value 66 represents an exponent of +2, while a characteristic of 61 represents an exponent of -3.
- The remaining M characters in bytes 2 through 8 represent the bits of the mantissa. There is an implied *radix point* before the leftmost bit of the mantissa; therefore, the mantissa is always less than 1. The term radix point is used instead of decimal point because decimal point implies that you are working with decimal (base 10) numbers, which might not be the case. The radix point can be thought of as the generic form of decimal point.

The exponent has a base associated with it. Do not confuse this with the base in which the exponent is represented; the exponent is always represented in binary, but the

exponent is used to determine how many times the base should be multiplied by the mantissa. In the case of the IBM mainframes, the exponent's base is 16. For other machines, it is commonly either 2 or 16.

Each bit in the mantissa represents a fraction whose numerator is 1 and whose denominator is a power of 2. For example, the leftmost bit in byte 2 represents $\left(\frac{1}{2}\right)^1$, the next bit represents $\left(\frac{1}{2}\right)^2$, and so on. In other words, the mantissa is the sum ofa series of fractions such as $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, and so on. Therefore, for any floating-point number to be represented exactly, you must be able to express it as the previously mentioned sum. For example, 100 is represented as the following expression:

$$\left(\frac{1}{4}+\frac{1}{8}+\frac{1}{64}\right)\times 16^{2}$$

To illustrate how the above expression is obtained, two examples follow. The first example is in base 10. The value 100 is represented as follows:

```
100.
```

The period in this number is the radix point. The mantissa must be less than 1; therefore, you normalize this value by shifting the radix point three places to the right, which produces the following value:

$$.100$$

Because the radix point is shifted three places to the right, 3 is the exponent:

$$.100\times 10^{3}=100$$

The second example is in base 16. In hexadecimal notation, 100 (base 10) is written as follows:

$$64.$$

Shifting the radix point two places to the right produces the following value:

$$.64$$

Shifting the radix point also produces an exponent of 2, as in:

$$.64\times 16^{2}$$

The binary value of this number is **.01100100**, which can be represented in the following expression:

$$\left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^3 + \left(\frac{1}{2}\right)^6 = \frac{1}{4} + \frac{1}{8} + \frac{1}{64}$$

In this example, the exponent is 2. To represent the exponent, you add the bias of 64 to the exponent. The hexadecimal representation of the resulting value, 66, is 42. The binary representation is as follows:

```
01000010 01100100 00000000 00000000
00000000 00000000 00000000 00000000
```

Floating Point Representation on OpenVMS

On OpenVMS, SAS stores numeric values in the D-floating format, which has the following scheme:

```
MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
byte 8   byte 7   byte 6   byte 5

MMMMMMMM MMMMMMMM SEEEEEEE EMMMMMMM
byte 4   byte 3   byte 2   byte 1
```

In D-floating format, the exponent is 8 bits instead of 7, but uses base 2 instead of base 16 and a bias of 128, which means the magnitude of the D-floating format is not as great as the magnitude of the IBM representation. The mantissa of the D-floating format is, physically, 55 bits. However, all floating-point values under OpenVMS are normalized, which means it is guaranteed that the high-order bit will always be 1. Because of this guarantee, there is no need to physically represent the high-order bit in the mantissa; therefore, the high-order bit is hidden.

For example, the decimal value 100 represented in binary is as follows:

```
01100100.
```

This value can be normalized by shifting the radix point as follows:

```
0.1100100
```

Because the radix was shifted to the left seven places, the exponent, 7 plus the bias of 128, is 135. Represented in binary, the number is as follows:

```
10000111
```

To represent the mantissa, subtract the hidden bit from the fraction field:

```
.100100
```

You can combine the sign (0), the exponent, and the mantissa to produce the D-floating format:

```
MMMMMMMM MMMMMMMM MMMMMMMM MMMMMMMM
00000000 00000000 00000000 00000000

MMMMMMMM MMMMMMMM SEEEEEEE EMMMMMMM
00000000 00000000 01000011 11001000
```

Floating-Point Representation Using the IEEE Standard

The Institute of Electrical and Electronic Engineers (IEEE) representation is used by many operating systems, including OS/2, Windows, and UNIX. The IEEE

representation uses an 11-bit exponent with a base of 2 and bias of 1023, which means that it has much greater magnitude than the IBM mainframe representation, but at the expense of 3 bits less in the mantissa. Note that the OS/2 operating system stores the floating-point numbers in the opposite order of most of the other operating systems listed. For example, the value of 1 represented by the IEEE standard is as follows:

```
3F F0 00 00 00 00 00 00
(most operating systems)

00 00 00 00 00 00 F0 3F
(OS/2)
```

Precision Versus Magnitude

As discussed in previous sections, floating-point representation allows for numbers of very large magnitude (numbers such as 2 to the 30th power) and high degrees of precision (many digits to the right of the decimal place). However, operating systems differ on how much precision and how much magnitude they allow.

In “Floating-Point Representation” on page 112, you can see that the number of exponent bits and mantissa bits varies. The more bits that are reserved for the mantissa, the more precise the number; the more bits that are reserved for the exponent, the greater the magnitude the number can have.

Whether precision or magnitude is more important depends on the characteristics of your data. For example, if you are working with physics applications, very large numbers may be needed, and magnitude is probably more important. However, if you are working with banking applications, where every digit is important but the number of digits is not great, then precision is more important. Most often, SAS applications need a moderate amount of both precision and magnitude, which is sufficiently provided by floating-point representation.

Computational Considerations of Fractions

Regardless of how much precision is available, there is still the problem that some numbers cannot be represented exactly. In the decimal number system, the fraction 1/3 cannot be represented exactly in decimal notation. Likewise, most decimal fractions (for example, .1) cannot be represented exactly in base 2 or base 16 numbering systems. This is the principle reason for difficulty in storing fractional numbers in floating-point representation.

Consider the IBM mainframe representation of .1:

```
40 19 99 99 99 99 99 99
```

Notice the trailing 9 digit, similar to the trailing 3 digit in the attempted decimal representation of 1/3 (.3333 …). This lack of precision is aggravated by arithmetic operations. Consider what would happen if you added the decimal representation of 1/3 several times. When you add .33333 … to .99999 … , the theoretical answer is 1.33333 … 2, but in practice, this answer is not possible. The sums become imprecise as the values continue.

Likewise, the same process happens when the following DATA step is executed:

```
data _null_;
   do i=-1 to 1 by .1;
      if i=0 then put 'AT ZERO';
   end;
run;
```

The AT ZERO message in the DATA step is never printed because the accumulation of the imprecise number introduces enough error that the exact value of 0 is never encountered. The number is close, but never exactly 0. This problem is easily resolved by explicitly rounding with each iteration, as the following statements illustrate:

```
data _null_;
   i=-1;
   do while(i<=1);
      i=round(i+.1,.001);
      if i=0 then put 'AT ZERO';
   end;
run;
```

Numeric Comparison Considerations

As discussed in "Computational Considerations of Fractions" on page 116, imprecision can cause problems with computations. Imprecision can also cause problems with comparisons. Consider the following example in which the PUT statement is not executed:

```
data _null_;
   x=1/3;
   if x=.33333 then put 'MATCH';
run;
```

However, if you add the ROUND function, as in the following example, the PUT statement is executed:

```
data _null_;
 x=1/3;
 if round(x,.00001)=.33333 then put 'MATCH';
run;
```

In general, if you are doing comparisons with fractional values, it is good practice to use the ROUND function.

Storing Numbers with Less Precision

As discussed in "Floating-Point Representation" on page 112, the SAS System allows for numeric values to be stored on disk with less than full precision. Use the LENGTH statement to dictate the number of bytes that are used to store the floating-point number. Use the LENGTH statement carefully to avoid significant data loss.

For example, the IBM mainframe representation uses 8 bytes for full precision, but you can store as few as 2 bytes on disk. The value 1 is represented as 41 10 00 00 00 00 00 00 in 8 bytes. In 2 bytes, it would be truncated to 41 10. You still have the full range of magnitude because the exponent remains intact; there are simply fewer digits involved. A decrease in the number of digits means either fewer digits to the right of the decimal place or fewer digits to the left of the decimal place before trailing zeroes must be used.

For example, consider the number 1234567890, which would be .1234567890 to the 10th power of 10 (in base 10). If you have only five digits of precision, the number becomes 123460000 (rounding up). Note that this is the case regardless of the power of 10 that is used (.12346, 12.346, .0000012346, and so on).

The only reason to truncate length by using the LENGTH statement is to save disk space. All values are expanded to full size to perform computations in DATA and PROC

steps. In addition, you must be careful in your choice of lengths, as the previous discussion shows.

Consider a length of 2 bytes on an IBM mainframe system. This value allows for 1 byte to store the exponent and sign, and 1 byte for the mantissa. The largest value that can be stored in 1 byte is 255. Therefore, if the exponent is 0 (meaning 16 to the 0th power, or 1 multiplied by the mantissa), then the largest integer that can be stored with complete certainty is 255. However, some larger integers can be stored because they are multiples of 16. For example, consider the 8-byte representation of the numbers 256 to 272 in the following table:

Table 10.8 Representation of the Numbers 256 to 272 in Eight Bytes

Value	Sign/Exp	Mantissa 1	Mantissa 2-7	Considerations
256	43	10	000000000000	trailing zeros; multiple of 16
257	43	10	100000000000	extra byte needed
258	43	10	200000000000	
259	43	10	300000000000	
	.			
	.			
	.			
271	43	10	F00000000000	
272	43	11	000000000000	trailing zeros; multiple of 16

The numbers from 257 to 271 cannot be stored exactly in the first 2 bytes; a third byte is needed to store the number precisely. As a result, the following code produces misleading results:

```
data temp;
   length x 2;
   x=257;
   y1=x+1;
run;

data _null_;
   set temp;
   if x=257 then put 'FOUND';
   y2=x+1;
run;
```

The PUT statement is never executed because the value of X is actually 256 (the value 257 truncated to 2 bytes). Recall that 256 is stored in 2 bytes as 4310, but 257 is also stored in 2 bytes as 4310, with the third byte of 10 truncated.

You receive no warning that the value of 257 is truncated in the first DATA step. Note, however, that Y1 has the value 258 because the values of X are kept in full, 8-byte floating-point representation in the program data vector. The value is only truncated when stored in a SAS data set. Y2 has the value 257, because X is truncated before the number is read into the program data vector.

CAUTION:
Do not use the LENGTH statement if your variable values are not integers. Fractional numbers lose precision if truncated. Also, use the LENGTH statement to truncate values only when disk space is limited. Refer to the length table in the SAS documentation for your operating environment for maximum values. △

Truncating Numbers and Making Comparisons

The TRUNC function truncates a number to a requested length and then expands the number back to full length. The truncation and subsequent expansion duplicate the effect of storing numbers in less than full length and then reading them. For example, if the variable

```
x=1/3;
```

is stored with a length of 3, then the following comparison is not true:

```
if x=1/3 then ...;
```

However, adding the TRUNC function makes the comparison true, as in the following:

```
if x=trunc(1/3,3) then ...;
```

Determining How Many Bytes Are Needed to Store a Number Accurately

To determine the minimum number of bytes needed to store a value accurately, you can use the TRUNC function. For example, the following program finds the minimum length of bytes (MINLEN) needed for numbers stored in a native SAS data set named NUMBERS. The data set NUMBERS contains the variable VALUE. VALUE contains a range of numbers, in this example, from 269 to 272:

```
data numbers;
   input value;
   datalines;
269
270
271
272
;

data temp;
   set numbers;
   x=value;
   do L=8 to 1 by -1;
      if x NE trunc(x,L) then
      do;
         minlen=L+1;
         output;
         return;
      end;
   end;
run;

proc print noobs;
   var value minlen;
```

```
run;
```

The following output shows the results from this code.

Output 10.3 Using the TRUNC Function

```
                        The SAS System

                        VALUE    MINLEN

                         269         3
                         270         3
                         271         3
                         272         2
```

Note that the minimum length required for the value 271 is greater than the minimum required for the value 272. This fact illustrates that it is possible for the largest number in a range of numbers to require fewer bytes of storage than a smaller number. If precision is needed for all numbers in a range, you should obtain the minimum length for all the numbers, not just the largest one.

Double-Precision Versus Single-Precision Floating-Point Numbers

You might have data created by an external program that you want to read into a SAS data set. If the data is in floating-point representation, you can use the RB*w.d* informat to read in the data. However, there are exceptions.

The RB*w.d* informat might truncate double-precision floating-point numbers if the *w* value is less than the size of the double-precision floating-point number (8 on all the operating systems discussed in this section). Therefore, the RB8. informat corresponds to a full 8-byte floating point. The RB4. informat corresponds to an 8-byte floating point truncated to 4 bytes, exactly the same as a LENGTH 4 in the DATA step.

An 8-byte floating point that is truncated to 4 bytes might not be the same as *float* in a C program. In the C language, an 8-byte floating-point number is called a *double*. In FORTRAN, it is a REAL*8. In IBM's PL/I, it is a FLOAT BINARY(53). A 4-byte floating-point number is called a *float* in the C language, REAL*4 in FORTRAN, and FLOAT BINARY(21) in IBM's PL/I.

On the IBM mainframes and OpenVMS VAX, a single-precision floating-point number is exactly the same as a double-precision number truncated to 4 bytes. On operating systems that use the IEEE standard, this is not the case; a single-precision floating-point number uses a different number of bits for its exponent and uses a different bias, so that reading in values using the RB4. informat does not produce the expected results.

Transferring Data between Operating Systems

The problems of precision and magnitude when you use floating-point numbers are not confined to a single operating system. Additional problems can arise when you move from one operating system to another, unless you use caution. This section discusses factors to consider when you are transporting data sets with very large or very small numeric values by using the UPLOAD and DOWNLOAD procedures, the CPORT and CIMPORT procedures, or transport engines.

Table 10.7 on page 113 shows the maximum number of digits of the base, exponent, and mantissa. Because there are differences in the maximum values that can be stored

in different operating environments, there might be problems in transferring your floating-point data from one machine to another.

Consider, for example, transporting data between an IBM mainframe and a PC. The IBM mainframe has a range limit of approximately .54E–78 to .72E76 (and their negative equivalents and 0) for its floating-point numbers. Other machines, such as the PC, have wider limits (the PC has an upper limit of approximately 1E308). Therefore, if you are transferring numbers in the magnitude of 1E100 from a PC to a mainframe, you lose that magnitude. During data transfer, the number is set to the minimum or maximum allowable on that operating system, so 1E100 on a PC is converted to a value that is approximately .72E76 on an IBM mainframe.

CAUTION:

Transfer of data between machines can affect numeric precision. If you are transferring data from an IBM mainframe to a PC, notice that the number of bits for the mantissa is 4 less than that for an IBM mainframe, which means you lose 4 bits when moving to a PC. This precision and magnitude difference is a factor when moving from one operating environment to any other where the floating-point representation is different. △

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS Language Reference: Concepts*, Cary, NC: SAS Institute Inc., 1999. 554 pages.

SAS Language Reference: Concepts

ISBN 1–58025–441–1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, November 1999