
Copyright © 2019, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government’s rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414
August 2020

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.

Other brand and product names are trademarks of their respective companies.

9.4-P3:lepg
# Contents

*About This Book* ................................................................. xi

## PART 1  Introduction  1

### Chapter 1  /  *The SAS System*  .................................................. 3
- About the SAS System .......................................................... 3
- Definition of Base SAS Software ........................................... 4
- Components of Base SAS Software ........................................ 4
- Base SAS Language Definitions ........................................... 5
- Anatomy of a SAS Program ............................................... 10
- Ways to Submit SAS Programs ........................................... 12
- Customizing Your SAS Session ........................................... 13
- Scope of This Book ...................................................... 14
- Operating Environment Information .................................... 14
- Using This Book ............................................................. 14

### Chapter 2  /  *SAS Processing* .................................................. 17
- Definition of SAS Processing ............................................... 17
- Types of Input to a SAS Program ........................................ 18
- The DATA Step ................................................................ 19
- The PROC Step .................................................................. 21
- SAS Processing Restrictions for Servers in a Locked-Down State .... 21

### Chapter 3  /  *DATA Step Processing* ............................................ 25
- Why Use a DATA Step? ........................................................ 25
- About Creating a SAS Data Set with a DATA Step .................... 26
- Reading Raw Data: Examples .............................................. 27
- Reading Data from SAS Data Sets ......................................... 32
- Generating Data from Programming Statements ....................... 33
- DATA Step Processing Time ............................................... 34
- Stored Compiled DATA Step Programs .................................. 36

## PART 2  Syntax  37

### Chapter 4  /  *Words and Names* .................................................. 39
- SAS Words ......................................................................... 40
- SAS Names ........................................................................ 42
- SAS Name Literals ............................................................. 46
- Variable Names .................................................................. 48
- Member Names .................................................................... 51
- Data Set Names ............................................................... 53
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Variables</td>
</tr>
<tr>
<td>6</td>
<td>Data Types</td>
</tr>
<tr>
<td>7</td>
<td>SAS Expressions</td>
</tr>
<tr>
<td>8</td>
<td>SAS Constants</td>
</tr>
<tr>
<td>9</td>
<td>Operators</td>
</tr>
<tr>
<td>10</td>
<td>Dates and Times</td>
</tr>
<tr>
<td>11</td>
<td>Component Objects</td>
</tr>
</tbody>
</table>

**Chapter 5 / Variables**
- Definition of SAS Variables: 71
- Ways to Create Variables: 71
- Creating a New Variable in a Formatted INPUT Statement: 73
- Manage Variables: 74
- Variable Attributes: 81
- Data Types: 85
- Variable Type Conversions: 88
- Automatic Variables: 90
- SAS Variable Lists: 91
- Missing Variable Values: 94
- Numeric Precision: 98
- Examples: Create and Modify SAS Variables: 111
- Examples: Control Output of Variables: 119
- Examples: Reorder and Align Variables: 122
- Examples: Convert Variable Types: 129
- Examples: Use Automatic Variables: 135
- Examples: Use Variable Lists: 138
- Examples: Manage Missing Variable Values: 142
- Examples: Manage Problems Related to Precision: 146
- Examples: Encrypt Variable Values: 159

**Chapter 6 / Data Types**
- Data Types in SAS: 165

**Chapter 7 / SAS Expressions**
- SAS Expressions: 167
- SAS Constants in Expressions: 168
- SAS Variables in Expressions: 168
- SAS Functions in Expressions: 169
- SAS Operators in Expressions: 169

**Chapter 8 / SAS Constants**
- SAS Constants in Expressions: 171
- Examples: Expressions and Constants: 177

**Chapter 9 / Operators**
- SAS Operators: 189
- Summary of Ways to Use Operators: 201
- Examples: Operators: 203

**Chapter 10 / Dates and Times**
- Dates, Times, and Intervals: 213

**Chapter 11 / Component Objects**
- DATA Step Component Objects: 215
# PART 4  Manipulating Data  361

## Chapter 19 / Grouping Data
- Definitions for BY-Group Processing ........................................... 363
- Syntax ......................................................................................... 363
- Understanding BY Groups ......................................................... 364
- Invoking BY-Group Processing .................................................. 364
- Determining Whether the Data Requires Preprocessing for BY-Group Processing ................................................. 365
- Preprocessing Input Data for BY-Group Processing ................. 366
- FIRST. and LAST. DATA Step Variables .................................... 367
- Processing BY-Groups in the DATA Step ................................. 367

## Chapter 20 / WHERE Statement Processing
- Definition of WHERE-Expression Processing ............................... 381
- Where to Use a WHERE Expression ........................................... 381
- Syntax of WHERE Expression .................................................... 382
- Combining Expressions By Using Logical Operators ................ 393
- Improving Performance of WHERE Processing ......................... 393
- Processing a Segment of Data That Is Conditionally Selected ...... 395
- Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement .................................................. 395

## Chapter 21 / Combining Data
- Overview of Combining Data .................................................... 401
- Comparing Methods ................................................................ 402
- Examples: Prepare Data ............................................................ 415
- Examples: Concatenate Data ..................................................... 423
- Examples: Interleave Data .......................................................... 438
- Examples: Match-Merge Data ................................................. 446
- Examples: Merge Data One-to-One ......................................... 453
- Examples: Combine Data One-to-One ..................................... 470
- Example: Merge Data Using a Hash Table ................................. 478
- Examples: Update Data ............................................................. 480
- Example: Modify Data ............................................................... 484

## Chapter 22 / Using Indexes
- Indexes in SAS ....................................................................... 497

## Chapter 23 / Using Arrays
- Definitions for Array Processing ............................................. 499
- Rules for Referencing Arrays ................................................... 499
- A Conceptual View of Arrays .................................................... 500
- Syntax for Defining and Referencing an Array ......................... 501
- Processing Simple Arrays ......................................................... 502
- Variations on Basic Array Processing ..................................... 507
- Multidimensional Arrays: Creating and Processing ................ 509
- Specifying Array Bounds .......................................................... 511
- Examples .................................................................................. 513

## Chapter 24 / Debugging Errors
- Types of Errors in SAS ............................................................. 519
- Error Processing in SAS ........................................................... 519
- Error Checking When Using Indexes to Randomly Access or Update Data ................................................. 529
- Examples .................................................................................. 547
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 30</td>
<td><strong>Repairing SAS Files</strong></td>
<td>633</td>
</tr>
<tr>
<td></td>
<td>Repairing Damage to SAS Data Sets and Catalogs</td>
<td>633</td>
</tr>
<tr>
<td></td>
<td>Examples</td>
<td>637</td>
</tr>
<tr>
<td>Chapter 31</td>
<td><strong>Compressing SAS Data Sets</strong></td>
<td>645</td>
</tr>
<tr>
<td></td>
<td>Compression in SAS</td>
<td>645</td>
</tr>
<tr>
<td>Chapter 32</td>
<td><strong>Moving SAS Files</strong></td>
<td>647</td>
</tr>
<tr>
<td></td>
<td>Moving Files between Operating Environments</td>
<td>647</td>
</tr>
<tr>
<td>Chapter 33</td>
<td><strong>Cross-Environment Data Access</strong></td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>Definitions for Cross-Environment Data Access (CEDA)</td>
<td>649</td>
</tr>
<tr>
<td></td>
<td>Advantages of CEDA</td>
<td>650</td>
</tr>
<tr>
<td></td>
<td>SAS File Processing with CEDA</td>
<td>650</td>
</tr>
<tr>
<td></td>
<td>Alternatives to Using CEDA</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>Examples: CEDA</td>
<td>656</td>
</tr>
<tr>
<td>Chapter 34</td>
<td><strong>Managing SAS Catalogs</strong></td>
<td>667</td>
</tr>
<tr>
<td></td>
<td>Definition of a SAS Catalog</td>
<td>667</td>
</tr>
</tbody>
</table>

**PART 7 SAS System Features**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 35</td>
<td><strong>The SAS Registry</strong></td>
<td>671</td>
</tr>
<tr>
<td></td>
<td>Introduction to the SAS Registry</td>
<td>671</td>
</tr>
<tr>
<td></td>
<td>Managing the SAS Registry</td>
<td>674</td>
</tr>
<tr>
<td></td>
<td>Configuring Your Registry</td>
<td>684</td>
</tr>
<tr>
<td>Chapter 36</td>
<td><strong>The SAS Windowing Environment</strong></td>
<td>689</td>
</tr>
<tr>
<td></td>
<td>What Is the SAS Windowing Environment?</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td>Main Windows in the SAS Windowing Environment</td>
<td>690</td>
</tr>
<tr>
<td></td>
<td>Navigating in the SAS Windowing Environment</td>
<td>698</td>
</tr>
<tr>
<td></td>
<td>Getting Help in SAS</td>
<td>703</td>
</tr>
<tr>
<td></td>
<td>List of SAS Windows and Window Commands</td>
<td>705</td>
</tr>
<tr>
<td></td>
<td>Introduction to Managing Your Data in the SAS Windowing Environment</td>
<td>709</td>
</tr>
<tr>
<td></td>
<td>Managing Data with SAS Explorer</td>
<td>710</td>
</tr>
<tr>
<td></td>
<td>Working with VIEWTABLE</td>
<td>715</td>
</tr>
<tr>
<td></td>
<td>Subsetting Data By Using the WHERE Expression</td>
<td>725</td>
</tr>
<tr>
<td></td>
<td>Exporting a Subset of Data</td>
<td>728</td>
</tr>
<tr>
<td></td>
<td>Importing Data into a Table</td>
<td>731</td>
</tr>
<tr>
<td>Chapter 37</td>
<td><strong>Cloud Analytic Services</strong></td>
<td>735</td>
</tr>
<tr>
<td></td>
<td>What is SAS Cloud Analytic Services?</td>
<td>735</td>
</tr>
<tr>
<td></td>
<td>What Does This Mean for the SAS 9 Programmer?</td>
<td>735</td>
</tr>
<tr>
<td></td>
<td>SAS Language Elements for CAS</td>
<td>736</td>
</tr>
<tr>
<td></td>
<td>CAS-specific Language Elements</td>
<td>738</td>
</tr>
<tr>
<td>Chapter 38</td>
<td><strong>Industry Protocols Used in SAS</strong></td>
<td>739</td>
</tr>
<tr>
<td></td>
<td>Sending E-Mail through SMTP</td>
<td>739</td>
</tr>
</tbody>
</table>
System Options That Control SMTP E-Mail ........................................ 740
Statements That Control SMTP E-mail ............................................. 741
Universally Unique Identifiers and the Object Spawner ...................... 742
Using SAS Language Elements to Assign UUIDs .............................. 744
Overview of IPv6 ................................................................................. 745
IPv6 Address Format ......................................................................... 745
Examples of IPv6 Addresses .............................................................. 746
Fully Qualified Domain Names (FQDN) ............................................... 747

PART 8  Appendixes  749

Appendix 1 / Data Sets Used in Examples ........................................... 751
  Data Sets Used in Examples ............................................................. 752
  Description of Column-Binary Data Storage ..................................... 762

Appendix 2 / Understanding How the DATA Step Works ....................... 765
  How the DATA Step Processes Data ................................................. 765
  Processing a DATA Step: A Walk-Through ...................................... 768
  More About DATA Step Execution ................................................. 772
About This Book

Audience

This document is appropriate for all users of the Base SAS programming language. New users can run examples to quickly demonstrate each feature. Experienced users can refer to the associated concepts topics for advanced details.

Syntax and Other References

The scope of this document includes features that are common to most SAS engines. Features that are specific to an engine are covered in the engine document. For example, features such as integrity constraints and catalogs are documented in SAS V9 LIBNAME Engine: Reference. For links to the engine documents, see “Summary of SAS Engines” on page 260.

For a summary of SAS programming interfaces, see “Ways to Submit SAS Programs” on page 12.

This document covers the common concepts for Base SAS syntax. See also these reference books:

- Base SAS Procedures Guide
- SAS Data Set Options: Reference
- SAS Formats and Informats: Reference
- SAS Functions and CALL Routines: Reference
- SAS DATA Step Statements: Reference
- SAS Global Statements: Reference
- SAS System Options: Reference
PART 1

Introduction

Chapter 1
The SAS System ......................................................... 3

Chapter 2
SAS Processing ............................................................. 17

Chapter 3
DATA Step Processing .................................................. 25
The SAS System

About the SAS System

The SAS System is an advanced analytical environment that enables you to access, explore, prepare, analyze, present, and share data. With SAS, you can access and analyze data from almost any data source and create high-quality graphics and reports that can be distributed in a variety of formats. You can also export the results to other systems. You can access, analyze, create, and store your data where it makes the most sense for your needs: on disk, in a database, or in the cloud.

The SAS System is the primary programming environment in the SAS 9.4 software product and industry-specific solutions from SAS.

In SAS Viya products, the SAS System is one of several programming environments. The SAS System is an addition to the action-based cloud analytic programming environment named SAS Cloud Analytic Services (CAS). The SAS System is also a client of the CAS server. Using the CAS engine provided with SAS Viya, you can connect to the CAS server and use Base SAS language statements to
read and manipulate data from a CAS session in your SAS session. In addition, you can load data from a SAS session to a CAS session. However, you cannot submit CAS actions to a CAS server with the Base SAS language.

Definition of Base SAS Software

In this book, the term Base SAS software represents the minimum functionality that is available with the SAS System. Base SAS software provides a powerful programming environment for performing such tasks as these:
- data entry, retrieval, and management
- statistical and mathematical analysis
- report writing and graphics
- applications development

Components of Base SAS Software

DATA step
- a programming language that you use to create, manipulate, and manage your data.

SAS procedures
- software tools for data analysis and reporting.

SAS global statements
- statements that perform an operation that gives information to SAS and can be used anywhere in SAS.

macro facility
- a tool for extending and customizing SAS software programs and for reducing text in your programs.

DATA step debugger
- a programming tool that helps you find logic problems in DATA step programs.

Output Delivery System (ODS)
- a system that delivers output in a variety of easy-to-access formats, such as SAS data sets, Hypertext Markup Language (HTML), and Portable Document Format (PDF).

a choice of user interfaces
- SAS offers batch, interactive command-line, desktop, and web-based graphical user interfaces to enable you to easily write, run, and test your SAS programs.

a choice of engines
- engines enable you to create SAS data sets and to read and write selected external files.
additional languages

Base SAS software includes a set of languages that augment the functionality provided by the preceding components. These languages include a fully integrated proprietary Structured Query Language (SQL) query facility named SAS SQL, an ANSI standard SQL query language named FedSQL, and a language for advanced data manipulation named DS2. The FedSQL and DS2 languages are partially integrated with Base SAS language.

---

Base SAS Language Definitions

SAS Files

When you work with the Base SAS language, you use files that are created and maintained by SAS, as well as files that are created and maintained by other systems that are not related to SAS.

Files that are created and maintained by SAS are referred to as SAS files. Table 1.1 summarizes the key SAS file types.

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS data set</td>
<td>a set of variables and observations that are stored as a unit. A SAS data set is similar to a relational table, which consists of columns and rows, except a SAS data set stores descriptive information about the variables and observations in addition to the data.</td>
</tr>
<tr>
<td>SAS view</td>
<td>a virtual data set that extracts data values from other files in order to provide a customized and dynamic representation of the data.</td>
</tr>
<tr>
<td></td>
<td>There are two types of views:</td>
</tr>
<tr>
<td></td>
<td>DATA step view is a stored DATA step program.</td>
</tr>
<tr>
<td></td>
<td>PROC SQL view is a stored query expression that is created in PROC SQL.</td>
</tr>
<tr>
<td>SAS catalog</td>
<td>a file that stores many different types of information that are used in a SAS job. Examples include instructions for reading and printing data values, or function key settings for SAS user interfaces.</td>
</tr>
<tr>
<td>SAS stored program</td>
<td>a type of SAS file that contains compiled code that you create and save for repeated use.</td>
</tr>
</tbody>
</table>
The contents of an item store are organized in a directory tree structure, which is similar to the directory structures that are used by UNIX System Services or by Windows. For example, a particular value might be stored and located using a directory path (root_dir/sub_dir/value). The SAS Registry is an example of an item store.

The most commonly used SAS file is the SAS data set. In most cases, the functionality that is available for a SAS data set is available for a SAS view.

External Files

Data files that you use to read and write data, but which are in a structure unknown to SAS, are called external files. External files can store any of the following:

- data that has not been processed by SAS
- SAS procedure output
- SAS programming statements

Database Management System Files

SAS software can read and write data to and from other vendors’ proprietary software, such as many common database management system (DBMS) files. Requirement: You must license the SAS/ACCESS software for the DBMS and operating environment in addition to Base SAS software.

SAS Language Elements

The SAS language consists of statements, expressions, options, formats, and functions similar to many other programming languages. In SAS, however, you use these elements within one of two groups of SAS statements:

- DATA steps
- PROC steps

A DATA step consists of a group of statements in the SAS language that create or manipulate SAS data sets. It is called a DATA step because every step begins with a DATA statement. The following figure shows how a DATA step creates a SAS data set in its simplest form:
You typically use a DATA step to read data from an input source, process it, and create a SAS data set. In addition, you can perform the following tasks:

- compute the values for new variables
- check for and correct errors in your data
- produce new SAS data sets by subsetting, merging, and updating existing data sets

A wide variety of statements are permissible in a DATA step.

Once your data is accessible as a SAS data set, you can analyze the data and write reports by using SAS procedures. The following figure shows how a PROC step can be used to operate on a SAS data set in its simplest form:

A group of procedure statements is called a PROC step. Every PROC step begins with a statement whose name begins with the word PROC and is followed by a keyword that describes the purpose of the procedure. Using a PROC step, you can perform simple tasks such as sorting data in a data set (PROC SORT) and printing a data set (PROC PRINT). In addition, you can analyze data in SAS data sets to produce statistics, tables, reports, charts, and plots; create SQL queries; and perform other analyses and operations on your data.

A procedure has a defined set of required and optional statements.

Global Statements

Global statements are SAS language elements that are assigned outside the DATA step and PROC step and provide information to both DATA steps and PROC steps. The primary use of global statements is defining data access for a SAS session. The secondary use is setting system options that modify the default behavior of the SAS System or one of its subsystems.
SAS Macro Facility

The macro facility is a powerful programming tool for extending and customizing your SAS programs, and for reducing the amount of code that you must enter to do common tasks. The macro facility enables you to assign a name to character strings or groups of SAS programming statements. You can work with the names that you created rather than with the text itself. You can use macros to automatically generate SAS statements and commands, write messages to the SAS log, accept input, or create and change the values of macro variables. For complete documentation, see SAS Macro Language: Reference.

SAS Engines

Base SAS provides several engines to read and write SAS data sets and selected external files. For a summary of engines, see Chapter 13, “SAS Engines,” on page 257.

In SAS®9 and SAS Viya, the default Base SAS engine is the V9 engine. Base SAS software includes an alternate storage engine: SAS Scalable Performance Data (SPD) engine. Most SAS language features are supported for both V9 engine and SPD engine, but not all.

The SAS language functionality described in this book applies to both storage engines. Functionality that is specific to the SAS V9 engine is provided in SAS V9 LIBNAME Engine: Reference. Functionality that is specific to SPD engine is provided in SAS Scalable Performance Data Engine: Reference.

Additional Languages

Base SAS software includes the following SAS languages in addition to the DATA step and PROC step. These language statements are submitted through SAS procedures. The languages can query and manipulate files created with the V9 engine, the SPD engine, and third-party databases that are accessed with SAS/ACCESS software.

Table 1.2 Additional Languages Provided with Base SAS

<table>
<thead>
<tr>
<th>Language</th>
<th>Procedure</th>
<th>Description</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>SAS SQL/PROC SQL</th>
<th>SAS FedSQL/PROC FEDSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>a SAS implementation of the ANSI SQL: 1992 core standard. A benefit of SAS SQL is that it is fully integrated with the SAS language: it uses SAS data types, can be managed with SAS system options, and fully supports SAS data set options, formats, and informats. It supports explicit SQL pass-through to external DBMS when appropriate SAS/ACCESS software is licensed.&lt;br&gt;SAS SQL statements are submitted from a SAS session by using the SQL procedure.&lt;br&gt;For more information, see <a href="#">SAS SQL Procedure User’s Guide</a>.</td>
<td>a SAS implementation of the ANSI SQL:1999 core standard. FedSQL uses ANSI SQL 1999 data types and core features and extensions. A benefit of SAS FedSQL is that it supports many more data types than SAS SQL and is a vendor-neutral SQL dialect. When appropriate SAS/ACCESS software is licensed, it can access data from various data sources without having to submit queries in the SQL dialect that is specific to the data source. FedSQL is partially integrated with the SAS language.&lt;br&gt;FedSQL statements are submitted from a SAS session by using the FEDSQL procedure. They can also be executed in a SAS Cloud Analytic Services (CAS) session by using the fedSql.execDirect action, and from SAS Federation Server (a separately licensed product).&lt;br&gt;For more information, see <a href="#">SAS FedSQL Language Reference</a>.</td>
</tr>
</tbody>
</table>
SAS DS2 PROC DS2

a SAS proprietary programming language that is appropriate for advanced data manipulation. It intersects with the DATA step, but supports additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages. DS2 supports most SAS language elements, except system options.

DS2 programs are submitted from a SAS session by using the DS2 procedure. They can also be executed from a CAS session by using the ds2.runDS2 action, and from SAS Federation Server (a separately licensed product).

For more information, see SAS DS2 Language Reference and SAS DS2 Programmer’s Guide.

---

Anatomy of a SAS Program

The following figure shows how the components of the Base SAS System are used in a SAS program:
1 System options determine how SAS initializes its interfaces with your computer hardware and the operating environment, how it reads and writes data, and other global functions. These can be set in your SAS program, or in a configuration file.

2 The Output Delivery System controls how DATA step and procedure output is formatted and displayed. The default output for an interactive SAS program is HTML. Optional output formats include PostScript (PDF), an output data set, and RTF (for Word files), among others. ODS options can be set in your SAS program, or in a configuration file.

3 The SAS macro language enables you to define SAS programming shortcuts. Using the SAS macro language is optional. Character values, numeric values, and SAS statements are associated with a variable that is defined in the %MACRO statement. Later, the variable can be specified in a DATA or PROC step with an ampersand (&). When the SAS System encounters the macro variable, it inserts the associated character value, numeric value, or SAS statements and includes them in its processing. The macro facility provides an easy way to substitute values or code segments in a SAS program. To execute a macro, you must precede the name of the macro by a percent (%) sign (for example, %test).

4 The global statement LIBNAME defines a pathname to a data library and associates a libref with it. The LIBNAME statement does not specify an engine,
so the default storage engine is used. The libref MyLibref can later be specified with a table name in a DATA or PROC step to identify the location of the table. The global statements TITLE and FOOTNOTE define a title and footnote to be appended to any session output.

5 The DATA step creates or modifies the data set MyLibref.MyData.

6 The PROC step analyzes the data set MyLibref.MyData and returns an output. The title and footnote that were defined earlier are included in the output.

---

### Ways to Submit SAS Programs

SAS provides several ways to submit SAS programs. The software for each of these methods is delivered with Base SAS software.

#### Table 1.3 Summary of SAS Software User Interfaces

<table>
<thead>
<tr>
<th>User Interface</th>
<th>Description</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS Studio</td>
<td>a web-based graphical user interface that enables you to write and run SAS code from your browser and display the results.</td>
<td>Getting Started with Programming in SAS Studio and SAS Studio: User’s Guide</td>
</tr>
<tr>
<td>SAS Windowing Environment</td>
<td>a desktop graphical user interface that enables you to write and run SAS code and display the results.</td>
<td>Getting Started Under Windows in SAS Companion for Windows</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Working in the SAS Windowing Environment in SAS 9.4 Companion for UNIX Environments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Overview of Windows in the z/OS Environment in SAS 9.4 Companion for z/OS</td>
</tr>
<tr>
<td>SAS Enterprise Guide</td>
<td>a Windows client application and graphical user interface for writing and submitting SAS code. It uses menus and wizards to guide you through code development.</td>
<td>SAS Enterprise Guide Online Documentation and SAS Enterprise Guide Tutorial</td>
</tr>
<tr>
<td>Non-interactive batch mode</td>
<td>a mode in which you place your SAS statements in a file and submit them for execution along with the control statements and system commands required at your site.</td>
<td>Running SAS in Batch Mode in SAS Companion for Windows Environments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Batch Mode in UNIX Environments in SAS Companion for UNIX Environments</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Destinations of SAS Output Files and Directing SAS Log and Procedure Output in SAS Companion for z/OS</td>
</tr>
</tbody>
</table>
Interactive line mode

Interactive line mode is a mode in which you enter program statements in sequence in response to prompts from the SAS System. DATA and PROC steps execute when a step boundary is reached (for example, a RUN or QUIT statement is encountered).

Customizing Your SAS Session

You can customize a SAS session by setting system options and by configuring SAS statements in a file for automatic execution.

When you customize your SAS session, we recommend that you set system options as follows:

1. Store system options with the settings that you want in a configuration file. When you start SAS, these settings are in effect. See the SAS documentation for your operating environment for more information about the configuration file.

   In some operating environments, you can use both a system-wide and a user-specific configuration file.

2. Specify system options in your SAS session to override the configured system options. System options that are specified directly in your SAS session are available for the duration of the SAS session.

   By placing SAS system options in a configuration file, you can avoid having to specify the options every time you start SAS. For example, you can specify the NODATE system option in your configuration file to prevent the date from appearing at the top of each page of your output.

   To execute SAS statements automatically each time you start SAS, store them in an autoexec file. SAS executes the statements automatically after the system is initialized. You can activate this file by specifying the AUTOEXEC= system option. See the SAS documentation for your operating environment for information about how autoexec files should be set up so that they can be located by SAS.

   Any SAS statement can be included in an autoexec file. For example, you can set report titles, footnotes, or create macros or macro variables automatically with an autoexec file.
Scope of This Book

This book covers SAS global statements and SAS DATA step concepts that apply to both the v9 and SPD engine storage engines. For a complete guide to Base SAS software functionality, also see these documents:

- Base SAS Procedures Guide
- SAS Global Statements: Reference
- SAS System Options: Reference
- SAS Macro Language: Reference
- SAS Logging: Configuration and Programming Reference
- SAS V9 LIBNAME Engine: Reference
- SAS Scalable Performance Data Engine: Reference
- SAS SQL Procedure User’s Guide
- SAS DS2 Language Reference
- SAS FedSQL Language Reference

Operating Environment Information

Base SAS software runs on Windows, UNIX, and z/OS. The implementation of SAS language functionality might differ slightly based on the host operating environment. For example, the details for storing and accessing SAS files depends on the host operating environment. The characteristics of external files also vary according to the host environment. See the following for host-specific information:

- SAS Companion for UNIX Environments
- SAS Companion for Windows
- SAS Companion for z/OS

Using This Book

- Part 1 acquaints you with the SAS System and how it works.
- Part 2 covers basic SAS programming rules.
- Parts 3 through 6 contain specific information about how to access and manipulate data with the SAS System, how to create output, and how to manage files.

- Part 7 provides conceptual information about special features of the SAS System.

- The appendix contains code that creates the SAS data sets that are used in the examples.
SAS Processing

Definition of SAS Processing
SAS processing is the way that the SAS language reads and transforms input data and generates the type of output that you request. The DATA step and the procedure (PROC) step are the two steps in the SAS language. Generally, the DATA step creates and manipulates data. The PROC step analyzes data, produces output, or manages SAS files. These two types of steps, used alone or combined, form the basis of SAS programs.

The following figure shows a high-level view of SAS processing using a DATA step and a PROC step. The figure focuses primarily on the DATA step.

Figure 2.1  SAS Processing Flow
You can use different types of data as input to a DATA step.

In the DATA step, you include SAS statements that contain instructions for processing the data.

As each DATA step in a SAS program is compiling or executing, SAS generates a log that contains processing messages and error messages. These messages can help you debug a SAS program.

PROC steps typically analyze and process data in the form of a SAS data set, but they can sometimes be used to create SAS data sets.

Types of Input to a SAS Program

You can use different sources of input data in your SAS program:

Existing SAS data sets
either a SAS data set or SAS view.

Raw data
unprocessed data that has not been read into a SAS data set. You can read raw data from two sources:

External files
contain records comprised of formatted data (data is arranged in columns) or free-formatted data (data that is not arranged in columns).

Instream data
raw data in the job stream.

SAS library
a SAS library enables you to access one or more files, referenced as a unit, by using an engine. You can access SAS data sets, selected external files, third-party DBMS, and PC files, depending on the engine that you specify in the library assignment. The engine transforms the target data into a form that is recognizable by SAS.

Remote access
enables you to read input data from nontraditional sources such as a TCP/IP socket or a URL. SAS treats this data as if it were coming from an external file. Remote access is available for the following data sources:

Azure
specifies the access method that enables you to access data in Microsoft Azure Data Lake Storage. The Azure access method is supported only for SAS Viya.

SAS catalog
specifies the access method that enables you to reference a SAS catalog as an external file.

Clipboard
specifies the access method that enables you to read or write text data to the clipboard on the host computer.

DATAURL
specifies the access method that enables you to access remote files by using the DATAURL access method.
EMAIL
specifies the access method that enables you to send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) email interface.

FILESRV
specifies the access method that enables you to store and retrieve user content using the SAS Viya Files service. The FILESRV access method is supported only for SAS Viya.

FTP
specifies the access method that enables you to use File Transfer Protocol (FTP) to read from or write to a file from any host computer that is connected to a network with an FTP server running.

Hadoop
specifies the access method that enables you to access files on a Hadoop Distributed File System (HDFS) whose location is specified in a configuration file.

S3
specifies the access method that enables you to access Amazon S3 files. The S3 access method is supported only for SAS Viya.

SFTP
specifies the access method that enables you to use Secure File Transfer Protocol (SFTP) to read from or write to a file from any host computer that is connected to a network with an Open SSH SSHD server running.

TCP/IP socket
specifies the access method that enables you to read from or write to a Transmission Control Protocol/Internet Protocol (TCP/IP) socket.

URL
specifies the access method that enables you to use the uniform resource locator (URL) to read from and write to a file from any host computer that is connected to a network with a URL server running.

WebDAV
specifies the access method that enables you to use the WebDAV protocol to read from or write to a file from any host computer that is connected to a network with a WebDAV server running.

ZIP
specifies the access method that enables you to access ZIP files by using zlib services.

The DATA Step

Definition of the DATA Step

The DATA step processes input data. In a DATA step, you can create a SAS data set. The DATA step uses input from SAS data sets, raw data, SAS libraries, remote access, or assignment statements. The DATA step can compute values, select specific input records for processing, and use conditional logic. The output from the
DATA step can be of several types, such as a SAS data set or a report. You can also write data to the SAS log or to an external data file. For more information, see Chapter 3, “DATA Step Processing,” on page 25.

DATA Step Output

The output from the DATA step can be a SAS data set or an external file such as the program log, a report, or an external data file. You can also update an existing file in place, without creating a separate data set. Data must be in the form of a SAS data set to be processed by many SAS procedures. You can create the following types of DATA step output:

SAS log
contains a list of processing messages and program errors. The SAS log is produced by default.

SAS data file
is a SAS data set that contains two parts: a data portion and a data descriptor portion.

SAS view
is a SAS data set that uses descriptor information and data from other files. SAS views enable you to dynamically combine data from various sources without using disk space to create a new data set. A SAS data file contains actual data values. However, SAS views contain only references to data stored elsewhere. SAS views are of member type VIEW. In most cases, you can use a SAS view as if it were a SAS data file.

External data file
contains the results of DATA step processing. These files are data or text files. The data can be records that are formatted or free-formatted.

Report
contains the results of DATA step processing. Although you usually generate a report by using a PROC step, you can generate the following two types of reports from the DATA step:

Procedure output file
contains printed results of DATA step processing and usually contains headings and page breaks.

HTML file
contains results that you can display on the World Wide Web. This type of output is generated through the Output Delivery System (ODS).

For more information about these output types, see Chapter 27, “Output,” on page 585.
The PROC Step

What Does the PROC Step Do?

The PROC step consists of a group of SAS statements that call and execute a procedure, usually with a SAS data set as input. Use PROCs to analyze the data in a SAS data set, produce formatted reports or other results, or provide ways to manage SAS files. You can modify PROCs with minimal effort to generate the output that you need. PROCs can also perform functions, such as displaying information about a SAS data set. For more information about SAS procedures, see Base SAS Procedures Guide.

PROC Step Output

The output from a PROC step can provide univariate descriptive statistics, frequency tables, crosstabulation tables, tabular reports consisting of descriptive statistics, charts, plots, and so on. Output can also be in the form of an updated data set. For more information about procedure output, see Base SAS Procedures Guide and the SAS Output Delivery System: User’s Guide.

SAS Processing Restrictions for Servers in a Locked-Down State

General Information

If you are running SAS in a client/server environment (for example, you are using SAS Enterprise Guide), the SAS server administrator can restrict access to files and directories on the host system. In addition, when a SAS session is in a locked-down state, certain access methods, functions, CALL routines, and procedures are restricted by default. For more information, see “Sign On to Locked-Down SAS Sessions” in SAS/CONNECT User’s Guide.

When SAS is in a locked-down state, the following SAS language elements are not available by default:
The ENABLE_AMS= option in the LOCKDOWN statement allows administrators to re-enable access methods and procedures that are restricted by default when LOCKDOWN is in effect. The following access methods and procedures can be re-enabled using the ENABLE_AMS= option in the LOCKDOWN statement:

**ENABLE_AMS= Option Values**

- FTP
- EMAIL
- HADOOP (enables PROC HADOOP)
- HTTP (enables PROC HTTP and PROC SOAP)
- SOCKET
- TCPIP
- URL (enables PROC HTTP and PROC SOAP)

If you attempt to use a resource that is locked down, SAS issues an error message to the SAS log. If the SAS session is configured for the SAS logging facility, SAS issues an error message to the *Audit.Lockdown* logger.

For more information, see the following resources. For SAS 9.4:

- LOCKDOWN System Option and LOCKDOWN Statement in *SAS Intelligence Platform: Application Server Administration Guide*
- Locked-Down Servers in *SAS Intelligence Platform: Security Administration Guide*

For Viya 3.5, see "LOCKDOWN System Option and LOCKDOWN Statement" in *SAS Viya Administrator: Programming Run-Time Servers*. 
To see the procedures that do not execute when the SAS server is in a locked-down state, see “Restrictions” and “Interactions” syntax information for the individual procedures in the *Base SAS Procedures Guide*.

### z/OS Specific Information

#### Restricted Features

Access to permanent z/OS data sets and UFS files and directories is not permitted unless enabled in the lockdown list. This restriction applies to all SAS features, most notably FILENAME and LIBNAME statements in SAS programs that are submitted for execution on the server. This restriction also applies to the ability to list files on the server through SAS clients such as SAS Enterprise Guide. When SAS is in the locked-down state, SAS does not permit access to uncataloged z/OS data sets except through externally allocated ddnames that are established by the server administrator. However, there are no restrictions on creating temporary z/OS data sets and UFS files, and processing them within the context of a single client session. The z/OS data sets are considered temporary if they are allocated DISP=(NEW,DELETE). External files are considered temporary if they are assigned using the FILENAME device of TEMP. All members of the client WORK library are considered temporary.

The SAS server administrator at your organization is responsible for the content of the lockdown list. Therefore, if you need to access a z/OS data set or UFS file that is unavailable in the locked-down state, contact your server administrator.

#### Disabled Features

The following SAS procedures, which are specific to z/OS, cannot be executed when SAS is in the locked-down state:

- PDS
- PDSCOPY
- RELEASE
- SOURCE
- TAPECOPY
- TAPELABEL

The following DATA step functions, which are specific to z/OS, cannot be executed when SAS is in the locked-down state:

- ZVOLLIST
- ZDSATTR
- ZDSLIST
- ZDSRATT
- ZDSNUM
- ZDSXATT
- ZDSIDNM
- ZDSYATT

The following access method, which is specific to z/OS, cannot be executed when SAS is in the locked-down state:

- VTOC
Specifying Functions in the Lockdown Path List

If the SAS session in which you are specifying a function is in a locked-down state, and the pathname that is specified in the function has not been added to the lockdown path list, then the function fails. A file access error related to the locked-down data will not be generated in the SAS log unless you specify the SYSMSG function.

The SYSMSG function can be placed after the function call in a DATA step to display lockdown-related file access errors.

This condition is true for the following functions, as well as for any other functions that take physical pathname locations as input:

- DCREATE
- FILEEXIST
- FILENAME
- RENAME
- DSNCATLGD (z/OS specific)
DATA Step Processing

Why Use a DATA Step?

The DATA step is the primary method for creating a SAS data set with Base SAS software. The DATA step also enables you to manipulate existing SAS data sets.

You can use the DATA step to perform the following tasks:

- subset, modify, combine, and update existing SAS data sets
- analyze, manipulate, and present your data
- compute values for new variables
- write reports
- retrieve information
- manage your SAS files
About Creating a SAS Data Set with a DATA Step

Creating a SAS Data Set or a SAS View

You can create either a SAS data set or a SAS view. A SAS view is a virtual data set that references data that is stored elsewhere. By default, you create a SAS data set. To create a SAS view instead, use the VIEW= option in the DATA statement. With a SAS view, you can process current input data values (such as monthly sales figures) without having to edit your DATA step. Whenever you need to create output, the output from a SAS view reflects the current input data values.

The following DATA statement specifies to create a SAS view called Monthly_Sales:

```sas
data monthly_sales / view=monthly_sales;
```

The following DATA statement specifies to create a data set called Test_Results.

```sas
data test_results;
```

Sources of Input Data

You select data-reading statements based on the source of your input data. There are at least six sources of input data:

- raw data in an external file
- raw data in the jobstream (instream data)
- data in SAS data sets
- data that is created by programming statements
- data that you can remotely access through a SAS catalog entry, clipboard, data URL, email, FTP protocol, Hadoop Distributed File System, TCP/IP socket, URL, WebDAV protocol, or through zlib services
- data that is stored in a Database Management System (DBMS) or other vendor's data files

Usually, DATA steps read input data records from only one of the first three sources of input. However, DATA steps can use a combination of some or all of the sources.
Reading Raw Data: Examples

Reading External File Data

Example Code

The components of a DATA step that produce a SAS data set from raw data stored in an external file are outlined here.

data Weight;  1
  infile 'your-input-file';  2
  input IDnumber $ week1 week16;  3
  WeightLoss=week1-week16;  4
run;  5

proc print data=Weight;  6
run;  7

1 Begin the DATA step and create the SAS data set Weight.
2 Specify the external file that contains your data.
3 Read a record and assign values to three variables.
4 Calculate a value for the variable WeightLoss.
5 Execute the DATA step.
6 Print the data set Weight using the PRINT procedure.
7 Execute the PRINT procedure.

Key Ideas

- The DATA statement specifies a name for the SAS data set.
- The INFILE statement specifies the path to the input file, within quotation marks.
- The INPUT statement describes the data set's variables.

See Also

Concepts
- Chapter 15, “Raw Data,” on page 315

Reference
- “DATA Statement” in SAS DATA Step Statements: Reference
Reading Instream Data Lines

Example Code

This example reads raw data from instream data lines.

```sas
data Weight2;  1
  input IDnumber $ week1 week16;  2
  AverageLoss=week1-week16;  3
datalines;  4
2477 195 163
2431 220 198
2456 173 155
2412 135 116
;  5
proc print data=Weight2;  6
run;
```

1. Begin the DATA step and create the SAS data set Weight2.
2. Read a data line and assign values to three variables.
3. Calculate a value for the variable WeightLoss2.
4. Begin the data lines.
5. Signal the end of data lines with a semicolon and execute the DATA step.
6. Print the data set Weight2 using the PRINT procedure.

Key Ideas

When you are providing data instream, recall these points:

- The INPUT statement describes the data set's variables.
- Values for the variables are supplied following a DATALINES statement as data lines.

See Also

Concepts

- Chapter 15, “Raw Data,” on page 315

Reference

- “DATALINES Statement” in SAS DATA Step Statements: Reference
- “INPUT Statement” in SAS DATA Step Statements: Reference
Reading Instream Data Lines with Missing Values

Example Code

You can also take advantage of options in the INFILE statement when you read instream data lines. This example shows the use of the MISSOVER option, which assigns missing values to variables for records that contain no data for those variables.

```sas
data weight2;
  infile datalines missover;
  input IDnumber $ Week1 Week16;
  WeightLoss2=Week1-Week16;
datalines;
  2477 195  163
  2431
  2456 173  155
  2412 135  116
;   
proc print data=weight2;
run;
```

1. Begin the DATA step and create the SAS data set Weight2.
2. Specify the INFILE statement, the Datalines variable, and the MISSOVER option.
3. Read a data line and assign values to three variables.
4. Begin the data lines.
5. Signal the end of data lines with a semicolon and execute the DATA step.
6. Print the data set Weight2 using the PRINT procedure.

Key Ideas

- The INFILE statement provides options for reading a new input data record if the DATALINES statement does not find values in the current input line for all the variables in the data set.
- The MISSOVER option can be used with the INFILE statement to assign missing values to variables that do not contain values in records.

See Also

Concepts

- Chapter 15, “Raw Data,” on page 315

Reference
Using Multiple Input Files in Instream Data

Example Code

This example shows how to use multiple input files as instream data to your program.

```sas
data all_errors;
  length filelocation $ 60;
  input filelocation;
  infile daily filevar=filelocation filename=daily end=done;
  do while (not done);
    input Station $ Shift $ Employee $ NumberOfFlaws;
    output;
  end;
  put 'Finished reading ' daily=;
  datalines;
  path myfile_A
  path myfile_B
  path myfile_C
;
  proc sort data=all_errors out=sorted_errors;
    by Station;
  run;
  proc print data = sorted_errors;
    title 'Flaws Report sorted by Station';
  run;
```

1. Begin the DATA step and create the SAS data set All_Errors.
2. Use the LENGTH statement to specify a length for the variable Filelocation.
3. Use the INPUT statement to read the content of the Filelocation variable.
4. Use the INFILE statement to define temporary variables for the input data. In the FILEVAR= option, specify the Filelocation variable. The FILEVAR= option enables you to specify a physical filename whose change in value causes the INFILE statement to close the current input file and open a new one. When the next INPUT statement executes, it reads from the new file that the FILEVAR= variable specifies.
5. In the FILENAME= option of the INFILE statement, specify the physical name of the currently opened input file. In this example, the file name is Daily.
6. Use the END= option of the INFILE statement to specify a variable that SAS sets to 1, when the current input data record is the last in the input file. In this example, the name of the variable is Done. Until SAS processes the last data record, the END= variable is set to 0.
Use the DO WHILE statement to execute the INPUT statement in a DO loop repetitively while the NOT DONE condition is true. The INPUT statement reads the input data and assigns values to four variables.

Specify the file pathnames following the DATALINES statement.

The program then sorts the observations by Station, and creates a sorted data set called Sorted_Errors.

The print procedure prints the results.

Output 3.1  Multiple Input Files in Instream Data

<table>
<thead>
<tr>
<th>Obs</th>
<th>Station</th>
<th>Shift</th>
<th>Employee</th>
<th>NumberOfFlaws</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Amherst</td>
<td>2</td>
<td>Lynne</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Goshen</td>
<td>2</td>
<td>Seth</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Hadley</td>
<td>2</td>
<td>Jon</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Holyoke</td>
<td>1</td>
<td>Walter</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Holyoke</td>
<td>1</td>
<td>Barb</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Orange</td>
<td>2</td>
<td>Carol</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>Otis</td>
<td>1</td>
<td>Kay</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Poitam</td>
<td>2</td>
<td>Mike</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>Stanford</td>
<td>1</td>
<td>Sam</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Suffield</td>
<td>2</td>
<td>Lisa</td>
<td>1</td>
</tr>
</tbody>
</table>

Key Ideas

The DATALINES statement does not provide input options for reading data. However, you can specify INFILE options to manipulate how data is supplied to the DATALINES statement.

See Also

Concepts

- Chapter 15, “Raw Data,” on page 315

Reference

- “DO WHILE Statement” in SAS DATA Step Statements: Reference
- “INFILE Statement” in SAS DATA Step Statements: Reference
Reading Data from SAS Data Sets

Example Code

This example reads data from one SAS data set, generates a value for a new variable, and creates a new data set.

```
data average_loss;  
   set weight;  
      Percent=round((AverageLoss * 100) / Week1);  
run;  
```

1. Begin the DATA step and create the SAS data set Average_Loss.
2. Specify the name of the existing SAS data set Weight in the SET statement.
3. Calculate a value for the variable Percent.
4. Execute the DATA step.

Key Ideas

The SET statement enables you to create a new SAS data set from an existing SAS data set.

See Also

Concepts

- Chapter 14, “SAS Data Sets,” on page 287

Reference

- “SET Statement” in SAS DATA Step Statements: Reference
Generating Data from Programming Statements

Example Code

Generating Data from Programming Statements

You can create data for a SAS data set by generating observations with programming statements rather than by reading data. A DATA step that reads no input goes through only one iteration.

data investment; 1
  begin='01JAN1990'd;
  end='31DEC2009'd;
  do year=year(begin) to year(end);  2
    Capital+2000 + .07*(Capital+2000);
    output; 3
  end;
  put 'The number of DATA step iterations is '_n_;  4
run; 5

proc print data=investment; 6
  format Capital dollar12.2; 7
run; 8

1 Begin the DATA step and create a SAS data set Investment.
2 Calculate a value based on a $2,000 capital investment and 7% interest each year from 1990 through 2009. Calculate variable values for one observation per iteration of the DO loop.
3 Write each observation to the data set Investment.
4 Write a note to the SAS log proving that the DATA step iterates only once.
5 Execute the DATA step.
6 To see your output, print the Investment data set with the PRINT procedure.
7 Use the FORMAT statement to write numeric values with dollar signs, commas, and decimal points.
8 Execute the PRINT procedure.
Key Ideas

- You can use assignment statements and SAS language statements to create calculated variables.
- This example defines beginning and end dates for the calculation with assignment statements. Then it uses a DO loop and a SUM statement to define the calculation. The SAS data set Investment contains the results of the calculation.

See Also

Concepts

- Chapter 14, “SAS Data Sets,” on page 287
- Chapter 12, “SAS Libraries,” on page 219
- Chapter 13, “SAS Engines,” on page 257
- Chapter 16, “Database and PC Files,” on page 345.

Reference

- “Assignment Statement” in SAS DATA Step Statements: Reference
- “DO WHILE Statement” in SAS DATA Step Statements: Reference
- “Sum Statement” in SAS DATA Step Statements: Reference

DATA Step Processing Time

DATA step processing time occurs in two stages: the first is the start-up (or compilation) time, and the second is the execution time.

- The compilation time is the time that it takes the SAS compiler to scan the SAS source code and convert it to an executable program.
- The execution time is the time that it takes SAS to execute the DATA step for each observation in a SAS file.

The two phases do not occur simultaneously: that is, the DATA step compiles first, and then it executes. For more information about these phases, see “How the DATA Step Processes Data” on page 765.

Understanding these processing times and how they relate to the structure of your SAS programs might be helpful when you are looking for ways to improve performance. In general, the more statements a DATA step processes, the longer the compilation time. Alternatively, DATA steps processing large numbers of
observations tend to have longer execution times because they are more I/O-intensive.

For example, a very large DATA step job that is not I/O-intensive (that is, it must process a relatively small number of observations) might need to be rewritten to reduce complexity and to eliminate repetitive and unused code. DO loops and user-defined functions created with PROC FCMP can reduce compilation time by decreasing the amount of code that must be compiled. For more information about how to improve performance when running CPU-intensive programs, see “Techniques for Optimizing CPU Performance” on page 565.

If most of the time used by the DATA step is for processing hundreds of observations, then other techniques designed to optimize I/O might be more useful. For more information about how to improve performance when running I/O-intensive programs, see “Techniques for Optimizing I/O” on page 557.

Several SAS system options provide information that can help you minimize processing time and optimize performance. For example, the FULLSTIMER option collects and displays performance statistics on each DATA step so that you can determine which resources were used for each step of data processing. For more information about this option and about optimization in general, see Chapter 25, “Optimizing System Performance,” on page 555.

The following example shows how to estimate the compilation time for a very large DATA step job that has a small number of observations. The program uses the DATETIME function with the %PUT macro statement to calculate the compilation start time. It then uses the _N_ automatic variable to find the execution start time (SAS always sets this variable to 1 at the start of the execution phase). By calculating the difference between the two times, the program returns the total compilation time of the DATA step.

**Example Code 3.1 Finding Compilation and Execution Time**

```sas
options nosource;
%put Starting compilation of DATA step: %QSYSFUNC(DATETIME(), DATETIME20.3);
%let startTime=%QSYSFUNC(DATETIME());

data a;
  if _N_ = 1 then do;
    endTime = datetime();
    put 'Starting execution of DATA step: ' endTime:DATETIME20.3;
    timeDiff=endTime-&startTime;
    put 'The Compile time for this DATA Step is approximately ' timeDiff:time20.6;
  end;
/* Lots of DATA step code */
run;
```

**Example Code 3.1 Log for Finding Compilation and Execution Time**

```
Starting compilation of DATA step: 03AUG20:12:58:59.877
Starting execution of DATA step: 03AUG20:12:58:59.894
The Compile time for this DATA Step is approximately 0:00:00.015865
NOTE: The data set WORK.A has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
  real time 0.00 seconds
  cpu time 0.00 seconds
```
Note: Macro statements and macro variables are resolved at compilation time and do not affect the time it takes to execute the DATA step. For information, see “Getting Started with the Macro Facility” in SAS Macro Language: Reference, and “SAS Programs and Macro Processing” in SAS Macro Language: Reference.

Stored Compiled DATA Step Programs

A stored compiled DATA step program is a SAS file that contains a DATA step program that has been compiled and then stored in a SAS library. You can execute stored compiled programs as needed, without having to recompile them. Stored compiled DATA step programs are of member type PROGRAM.

Stored compiled programs are available for the V9 engine and DATA step applications only. For more information, see “Stored, Compiled DATA Step Programs” in SAS V9 LIBNAME Engine: Reference.
PART 2
Syntax

Chapter 4
   Words and Names .................................................. 39

Chapter 5
   Variables ............................................................... 69

Chapter 6
   Data Types ........................................................... 165

Chapter 7
   SAS Expressions .................................................... 167

Chapter 8
   SAS Constants ....................................................... 171

Chapter 9
   Operators ............................................................. 189

Chapter 10
   Dates and Times ...................................................... 213

Chapter 11
   Component Objects .................................................. 215
Definition of a SAS Word

A word in the SAS programming language is a collection of characters that communicates a meaning to SAS. A word cannot be divided into smaller units that can be used independently. A word can contain a maximum of 32,767 bytes.

A word ends when SAS encounters one of the following conditions:

- the beginning of a new word
- a blank after a name or a number
- the ending quotation mark of a literal word

Types of Words or Tokens

Each word in the SAS language belongs to one of four categories:

- name
  - is a series of characters that begin with a letter or an underscore. Later characters can include letters, underscores, and numeric digits. A name can contain up to 32,767 bytes. In most contexts, however, names are limited to a shorter maximum length, such as 32 or 8 bytes. See Table 4.1 on page 43. Here are some examples of name tokens:
    - data
    - _new
    - yearcutoff
    - year_99
    - descending
    - _n_

- literal
  - consists of 1 to 32,767 bytes enclosed in single or double quotation marks. Here are some examples of literals:
    - 'Chicago'
    - "1990-91"
    - 'Amelia Earhart'
    - 'Amelia Earhart''s plane'
    - "Amelia Earhart's plane"
"Report for the Third Quarter"

Note: The surrounding quotation marks identify the character string as a literal, but SAS does not store these marks as part of the literal string.

Number

In general, a number is composed entirely of numeric digits, with an optional decimal point and a leading plus or minus sign. SAS recognizes numeric values in the following forms as number tokens: scientific (E−) notation, hexadecimal notation, missing value symbols, and date and time literals. Here are some examples of numbers:

- 5683
- 2.35
- 0b0x
- -5
- 5.4E-1
- '24aug90'd

Special Character

A special character is usually any single keyboard character other than letters, numbers, the underscore, and the blank. In general, each special character is a single word, although some two-character operators, such as ** and <=, form a single word. The blank can end a name or a number, but it is not a word. Here are some examples of special characters:

- =
- ;
- '
- +
- @
- /

Placement and Spacing of Words

The spacing requirements for words in SAS statements include the following:

- You can begin SAS statements in any column of a line and write several statements on the same line.
- You can begin a statement on one line and continue it on another line, but you cannot split a word between two lines.
- A blank is not treated as a character in a SAS statement unless it is enclosed in quotation marks as a literal or part of a literal. Therefore, you can put multiple blanks any place in a SAS statement where you can put a single blank. It has no effect on the syntax.
- The rules for recognizing the boundaries of words or tokens determine the use of spacing between them in SAS programs. If SAS can determine the beginning of
SAS Names

Definition of a SAS Name

A *SAS name* is a name *token* that represents one of the following SAS language elements:

- array
- format
- option
- catalog entry
- informat
- procedure
- component object
- libref or fileref
- statement label
- data set
- macro or macro variable
- variable

There are two types of names in SAS:

- *SAS language element* names
- user-supplied names

Rules for Most SAS Names

The following list contains the rules that you use when you create most SAS names:

- The length of a SAS name depends on which element it is assigned to. Many SAS names can be 32 bytes long; others have a maximum length of 8 bytes. For a list of SAS names and their maximum length, see Table 4.1.
- The first character must be an English letter (A, a, B, b, C, c, . . . , Z, z,) or an underscore (_). Subsequent characters can be letters, numeric digits (0, 1, . . . , 9), or underscores.

**IMPORTANT** User-defined format names cannot end in a number.
You can use uppercase or lowercase letters.

- Blanks are not allowed.
- Special characters, except for the underscore, are not allowed. In *filerefs* only, you can use the dollar sign ($), the number sign (#), and the at sign (@).
- SAS reserves a few names for automatic variables and variable lists, SAS data sets, and librefs.
  - When creating variables, do not use the names of special SAS automatic variables (for example, _N_ and _ERROR_) or special variable list names (for example, _CHARACTER_, _NUMERIC_, and _ALL_).
  - When associating a *libref* with a SAS library, do not use these *libref* names:
    - Sashelp
    - Sasmsg
    - Sasuser
    - Work
  - When you create SAS data sets, do not use these names:
    - _NULL_
    - _DATA_
    - _LAST_
  - When assigning a *fileref* to an external file, do not use the file name SASCAT.
  - When you create a macro variable, do not use names that begin with SYS.

### Length Rules for Names

The length of a SAS name depends on which element it is assigned to.

#### Table 4.1 Maximum Length of SAS Names by Language Element

<table>
<thead>
<tr>
<th>Language Element Names</th>
<th>Maximum Length in Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array names</td>
<td>32</td>
</tr>
<tr>
<td>CALL routines names</td>
<td>16</td>
</tr>
<tr>
<td>Catalog names</td>
<td>32</td>
</tr>
<tr>
<td>Component object names</td>
<td>32</td>
</tr>
<tr>
<td>Engine names</td>
<td>8</td>
</tr>
<tr>
<td>Fileref names</td>
<td>8</td>
</tr>
<tr>
<td>Format names, character</td>
<td>31</td>
</tr>
<tr>
<td>Language Element Names</td>
<td>Maximum Length in Bytes</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Format names, numeric</td>
<td>32</td>
</tr>
<tr>
<td>Function names</td>
<td>16</td>
</tr>
<tr>
<td>Generation data set names</td>
<td>28</td>
</tr>
<tr>
<td>Index names</td>
<td>32</td>
</tr>
<tr>
<td>Informat names, character</td>
<td>30</td>
</tr>
<tr>
<td>Informat names, numeric</td>
<td>31</td>
</tr>
<tr>
<td>Librefs (library names)</td>
<td>8</td>
</tr>
<tr>
<td>Library Member names</td>
<td>32</td>
</tr>
<tr>
<td>Macro variable names</td>
<td>32</td>
</tr>
<tr>
<td>Macro window names</td>
<td>32</td>
</tr>
<tr>
<td>Macro names</td>
<td>32</td>
</tr>
<tr>
<td>Password names</td>
<td>8</td>
</tr>
<tr>
<td>Procedure names (the first eight characters must be unique and cannot begin with “SAS”)</td>
<td>16</td>
</tr>
<tr>
<td>Statement label names</td>
<td>32</td>
</tr>
<tr>
<td>Variable (includes DATA step variables and SCL variables) names</td>
<td>32</td>
</tr>
<tr>
<td>Variable label names</td>
<td>256</td>
</tr>
<tr>
<td>View names</td>
<td>32</td>
</tr>
<tr>
<td>Window names</td>
<td>32</td>
</tr>
</tbody>
</table>

**Extending SAS Names**

When `VALIDVARNAME=ANY` or `VALIDMEMNAME=EXTEND`, the length of these SAS names must be measured in bytes:

1. Some examples of SAS library member names are data set names, view names, and catalog names. For more information, see “Member Types” in *Base SAS Procedures Guide*. 
When these system option values are set, the maximum number of characters that you can use for a SAS variable name, data set name, view name, or item store name is determined by the number of bytes of storage that are used to store one character. This value is set by the SAS encoding value for your SAS session. VALIDVARNAME=ANY or VALIDMEMNAME=EXTEND must be set to allow the use of National Language Support (NLS) characters. Otherwise, only one-byte characters are allowed.

The SAS encodings for western languages use one byte of storage to store one character. Therefore, in western languages, you can use 32 characters for these SAS names. The SAS encoding for some Asian languages use one to two bytes of storage to store one character. The Unicode encoding, UTF-8, supports one to four bytes of storage for a single character. When the SAS encoding uses four bytes to store one character, the maximum length of one of these SAS names is eight characters.

All SAS encodings support the characters A–Z and a–z as one-byte characters.

Follow these instructions for finding the maximum number of characters that can be used for a SAS name:

1. Find the SAS encoding in one of the following ways:
   - Find the ENCODING= system option in the SAS System Options window:
     1. Type options on the command bar.
     2. Right-click Options and select Find Option.
     3. Type encoding and click OK.
   - In a SAS session, submit the ENCODING= system option in the OPTIONS procedure:
     ```sas
     proc options option=encoding;
     run;
     ```

2. In the table “SBCS, DBCS, and Unicode Encoding Values Used to Transcode Data,” find the maximum number of bytes per character for the SAS encoding. This table is in SAS National Language Support (NLS): Reference Guide.

3. Find the maximum number of bytes for a SAS name from Table 4.1. Divide this number by the bytes per character. The result is the maximum number of characters that you can use for the SAS name.
SAS Name Literals

Definition of SAS Name Literals

A SAS *name literal* is a user-supplied name that is expressed as a string within quotation marks, followed by the uppercase or lowercase letter *n*. Most SAS names allow only the characters __, A–Z, and a–z. Name literals enable you to use characters (including blanks and national characters) that are not otherwise allowed.

You can use name literals in these types of SAS names:

- DBMS column names
- DBMS table
- item store
- SAS data set
- SAS view
- statement label
- variable names and values

To use characters in a name literal other than __, A–Z, or a–z, you must set either the VALIDVARNAME=ANY or VALIDMEMNAME=EXTEND system options. The following table specifies the options that you must set to use SAS name literals.

<table>
<thead>
<tr>
<th>SAS Name Type</th>
<th>Name Literal Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS column</td>
<td>Set VALIDVARNAME=ANY.</td>
</tr>
</tbody>
</table>
### SAS Name Literals

<table>
<thead>
<tr>
<th>SAS Name Type</th>
<th>Name Literal Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBMS table name</td>
<td>Set VALIDMEMNAME=EXTEND.</td>
</tr>
<tr>
<td>item store</td>
<td>Set VALIDMEMNAME=EXTEND.</td>
</tr>
<tr>
<td>SAS data set name</td>
<td>Set VALIDMEMNAME=EXTEND.</td>
</tr>
<tr>
<td>SAS view</td>
<td>Set VALIDMEMNAME=EXTEND.</td>
</tr>
<tr>
<td>statement label</td>
<td>Set VALIDVARNAME=ANY.</td>
</tr>
<tr>
<td>variable</td>
<td>Set VALIDVARNAME=ANY.</td>
</tr>
</tbody>
</table>

Name literals are especially useful for expressing DBMS column and table names that contain special characters and for including national characters in SAS names.

### Important Restrictions

- You can use a name literal only for variables, statement labels, DBMS column and table names, SAS data sets, SAS view, and item stores.
- When the name literal of a SAS data set name, a SAS view name, or an item store name contains any characters that are not allowed when VALIDMEMNAME=COMPAT, then you must set the system option VALIDMEMNAME=EXTEND. See “VALIDMEMNAME= System Option” in SAS System Options: Reference.
- When the name literal of a variable, DBMS table, or DBMS column contains any characters that are not allowed when VALIDVARNAME=V7, then you must set the system option VALIDVARNAME=ANY. See “VALIDVARNAME= System Option” in SAS System Options: Reference.
- If you use either the percent sign (%) or the ampersand (&), then you must use single quotation marks in the name literal in order to avoid interaction with the SAS Macro Facility.
- When the name literal of a DBMS table or column contains any characters that are not valid for SAS rules, you might need to specify a SAS/ACCESS LIBNAME statement option. For more details and examples about the SAS/ACCESS LIBNAME statement and about using DBMS table and column names that do not conform to SAS naming conventions, see SAS/ACCESS for Relational Databases: Reference.
- In a quoted string, SAS preserves and uses leading blanks, but SAS ignores and trims trailing blanks.
- Blanks between the closing quotation mark and the $ are not valid when you specify a name literal.
- Note that even if you set VALIDVARNAME=ANY, the V6 engine does not support names that have intervening blanks.
Using Name Literals in BY Groups

When you designate a name literal as the BY variable in BY-group processing and you want to refer to the corresponding FIRST. or LAST. temporary variables, you must include the FIRST. or LAST. portion of the two-level variable name within quotation marks.

Avoiding Errors When Using Name Literals

For information about avoiding errors when using name literals, see “Avoiding a Common Error with Character Constants”.

See Also

Examples
- “Example: Create a Variable Name Containing Blanks”
- “Example: Specify the FIRST. and LAST. BY Variables as Name Literals”

Statements
- “BY Statement” in SAS DATA Step Statements: Reference
- “OPTIONS Statement” in SAS Global Statements: Reference

System Options
- “VALIDMEMNAME= System Option” in SAS System Options: Reference
- “VALIDVARNAME= System Option” in SAS System Options: Reference

Variable Names

Rules for Variable Names

The rules for SAS variable names have expanded to provide more functionality. The setting of the “VALIDVARNAME= System Option” in SAS System Options: Reference determines which rules apply to the variables that you create in your SAS session, as well as to variables that you want to read from existing data sets.

The VALIDVARNAME= system option has three settings (V7, UPCASE, and ANY), each with varying degrees of flexibility for variable names. If you do not specify the
VALIDVARNAME= system option in your SAS session, the default value, V7, is automatically assigned to your SAS session.

The table below shows a summary of the rules for naming SAS variables when the VALIDVARNAME= system option is set to V7. These are the default settings in Base SAS. In some SAS applications, such as SAS Visual Analytics and SAS Cloud Analytic Services, the VALIDVARNAME= system options are set by default to allow the most flexibility for naming SAS variables. These rules are summarized in Table 4.5.

**Table 4.3  Summary of Default Rules for Naming SAS Variables**

<table>
<thead>
<tr>
<th>Variable Names (with VALIDVARNAME=V7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>can be up to 32 bytes in length.</td>
</tr>
<tr>
<td>cannot contain special characters (except for the underscore), blanks, or national characters.</td>
</tr>
<tr>
<td>must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore.</td>
</tr>
<tr>
<td>can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable.</td>
</tr>
</tbody>
</table>

Note that SAS internally converts all variable names to uppercase.

For example, the names cat, Cat, and CAT all represent the same variable.

---

**Extended Rules for SAS Variables**

The following table shows a summary of rules for naming SAS variables when the VALIDVARNAME= system option is set to UPCASE. All the variable names are uppercase.

**Table 4.4  Summary of Extended Rules for Naming SAS Variables**

<table>
<thead>
<tr>
<th>Variable Names (with VALIDVARNAME=UPCASE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>can be up to 32 bytes in length.</td>
</tr>
<tr>
<td>cannot contain special characters (except for the underscore), blanks, or national characters.</td>
</tr>
<tr>
<td>must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore.</td>
</tr>
<tr>
<td>can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable.</td>
</tr>
</tbody>
</table>

Note that SAS internally converts all variable names to uppercase.

For example, the names cat, Cat, and CAT all represent the same variable.
The following table shows a summary of the rules for naming SAS variables when the highest level of flexibility is allowed (that is, when the VALIDVARNAME= system option is set to ANY).

**Table 4.5 Summary of Extended Rules for Naming SAS Variables**

<table>
<thead>
<tr>
<th>Variable Names (with VALIDVARNAME=ANY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>▪ can be up to 32 bytes in length.</td>
</tr>
<tr>
<td>▪ can contain special characters including / \ * ? &quot; &lt; &gt;</td>
</tr>
<tr>
<td>▪ can begin with any character, including blanks, national characters, special characters, and multibyte characters.</td>
</tr>
<tr>
<td>▪ preserves leading blanks, but trailing blanks are ignored.</td>
</tr>
<tr>
<td>▪ can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable.</td>
</tr>
<tr>
<td>Note that SAS converts all variable names to uppercase. For example, the names cat, Cat, and CAT all represent the same variable.</td>
</tr>
<tr>
<td>▪ cannot contain all blanks.</td>
</tr>
</tbody>
</table>

**IMPORTANT** Throughout SAS, using the name literal syntax with variable names that exceed the 32-byte limit or have excessive embedded quotation marks might cause unexpected results. The intent of the VALIDVARNAME=ANY system option is to enable compatibility with other DBMS variable (column) naming conventions, such as allowing embedded blanks and national characters.

**Note:** If you use any characters other than the ones that are valid when the VALIDVARNAME= system option is set to V7 (letters from the Latin alphabet, numerals, or underscores), then you must express the variable name as a name literal and you must set VALIDVARNAME=ANY. If the name includes either the percent sign (%) or the ampersand (&), then you must use single quotation marks in the name literal in order to avoid interaction with the SAS Macro Facility. See **SAS Name Literals** and **Avoiding Errors When Using Name Literals**.

### See Also

**Examples**
- "Example: Create a Variable Name Containing Blanks"
- "Example: Specify the FIRST. and LAST. BY Variables as Name Literals"
- "Example: Create a SAS Data Set Name Containing a Special Character"

**Statements**
- "OPTIONS Statement" in SAS Global Statements: Reference

**System Options**
Member Names

Rules for Member Names

Member names include SAS data set names, views names, and item store names. The rules for member names have expanded to provide more functionality. The setting of the “VALIDMEMNAME= System Option” in SAS System Options: Reference determines which rules you apply to the member names that you create in your SAS session. The VALIDMEMNAME= system option has three settings, each with varying degrees of flexibility for member names. If you do not specify the VALIDMEMNAME= system option in your SAS session the default is COMPATIBLE.

Table 4.6  Summary of Default Rules for Naming SAS Member Names

<table>
<thead>
<tr>
<th>Member Names (with VALIDMEMNAME=COMPATIBLE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The length of the names can be up to 32 characters.</td>
</tr>
<tr>
<td>- Names must begin with a letter from the Latin alphabet (A–Z, a–z) or an underscore. Subsequent characters can be letters from the Latin alphabet, numerals, or underscores.</td>
</tr>
<tr>
<td>- Names cannot contain blanks or special characters except for the underscore.</td>
</tr>
<tr>
<td>- Names can contain mixed-case letters. SAS internally converts the member name to uppercase. Note that the names, customer, Customer, and CUSTOMER all represent the same member name.</td>
</tr>
</tbody>
</table>

Extended Rules for SAS Member Names

The following table shows a summary of the rules for naming SAS member names when the highest level of flexibility is allowed (that is, when the VALIDMEMNAME= system option is set to EXTEND).
Table 4.7  Summary of Extended Rules for Naming SAS Member Names

**Variable Names**
(with VALIDMEMNAME=EXTEND)

- Names can include national characters.
- The name can include special characters, except for the / \ * ? " < > | : -. characters.  
  **Note:** The SPD Engine does not allow '.' (the period) anywhere in the member name.
- The name must contain at least one character (letters, numbers, valid special characters, and national characters).
- The length of the name can be up to 32 bytes.
- Null bytes are not allowed. Names cannot begin with a blank or a '.' (the period).  
  **Note:** The SPD Engine does not allow '$' as the first character of the member name.
- Leading and trailing blanks are deleted when the member is created.
- Names can contain mixed-case letters. SAS internally converts the member name to uppercase. Note that the names, `customer`, `Customer`, and `CUSTOMER` all represent the same member name.

**IMPORTANT** Throughout SAS, using the name literal syntax with variable names that exceed the 32-byte limit or have excessive embedded quotation marks might cause unexpected results. The intent of the VALIDVARNAME=ANY system option is to enable compatibility with other DBMS variable (column) naming conventions, such as allowing embedded blanks and national characters.

**Note:** Regardless of the value of the VALIDMEMNAME= system option, a member name cannot end in the special character # followed by three digits. This is because it would conflict with the naming conventions for generation data sets. Using such a member name results in an error.

**Note:** When VALIDMEMNAME=EXTEND, SAS data set names, SAS data view names, and item store names must be written as a SAS name literal if the name includes blank spaces, special characters, or national characters. If you use either the percent sign (%) or the ampersand (&), then you must use single quotation marks in the name literal in order to avoid interaction with the SAS Macro Facility. For more information, see “SAS Name Literals”.

---

**See Also**

**Statements**
- “OPTIONS Statement” in SAS Global Statements: Reference

**System Options**
- “VALIDMEMNAME= System Option” in SAS System Options: Reference
- “VALIDVARNAME= System Option” in SAS System Options: Reference
Data Set Names

Definition of a Data Set Name

A data set name is a user-supplied name that you give to a SAS data set when you create it. Output data sets that you create in a DATA step are named in the DATA statement. SAS data sets that you create in a procedure step are usually given a name in the procedure statement or in an OUTPUT statement.

Follow the rules for naming SAS member names when naming SAS data sets. See Rules for Member Names on page 51 for more information.

If you do not specify a name for the output data set in a DATA statement, SAS automatically assigns a default data set name. If you do not specify a name for the input data set in a SET statement, SAS automatically uses the last data set that was created. For more information, see “Special Data Set Names” on page 55.

Here are some statements that might require you to supply a name for a data set:

- DATA= option
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “MODIFY Statement” in SAS DATA Step Statements: Reference
- OPEN function
- “SET Statement” in SAS DATA Step Statements: Reference
- SQL Procedure
- “UPDATE Statement” in SAS DATA Step Statements: Reference

SAS view names are created using one of the following:

- the VIEW= option in the “DATA Statement” in SAS DATA Step Statements: Reference
- the ACCESS procedure

Note: SAS data sets and SAS views that share the same library cannot have the same name.

Structure of a Data Set Name

Levels

The complete name of every SAS data set has three levels. You assign the first two levels and SAS supplies the third. The form for a SAS data set name is as follows:
libref

is the user-supplied logical name that is associated with the physical location of the SAS library.

SAS data set

is the user-supplied data set name, which can be up to 32 bytes long for the Base SAS engine starting in Version 7. Earlier SAS versions are limited to 8-byte names.

member type

is the system-created name assigned by SAS that identifies the type of information that is stored in a SAS file. For example, SAS data sets are of type DATA and SAS DATA step views are of type VIEW. Other member types are ACCESS, AUDIT, DMBD, CATALOG, FDB, INDEX, ITEMSTOR, MDDB, PROGRAM, and UTILITY.

When you refer to SAS data sets in your program statements, use a one- or two-level name. You can use a one-level name when the data set is in the temporary library Work. In addition, if the reserved libref User is assigned, you can use a one-level name when the data set is in the permanent library User. Use a two-level name when the data set is in some other permanent library that you have established.

One-level Names

A one-level name consists of just the data set name. You can omit the libref, and refer to data sets with a one-level name in the following form:

scores

Data sets with one-level names are automatically assigned to one of two SAS libraries:

Work

a SAS system library that is used for temporarily saving data sets with one-level names. Temporarily means that the contents of the library are deleted at the end of the SAS job or session. Data sets with one-level names are stored in the Work library by default.
User

a SAS system library that is used for permanently saving data sets with one-level names.

Most commonly, they are assigned to the temporary library Work and are deleted at the end of a SAS job or session. If you have associated the libref User with a SAS library or used the USER= system option to set the User library, data sets with one-level names are stored in that library.

See “User Library” and “Work Library (Temporary)” for more information about using the Work and User libraries.

Two-level Names

A two-level name consists of both the libref and the data set name. The form most commonly used to create, read, or write to SAS data sets in permanent SAS libraries is the two-level name as shown here:

```
myfiles.scores
```

When you create a new SAS data set, the libref indicates where it is to be stored. When you reference an existing data set, the libref tells SAS where to find it.

Special Data Set Names

**DATAn Data Set**

If you do not specify a name for the output data set or specify the reserved name _NULL_ in a DATA statement, SAS automatically assigns the names DATA1, DATA2, and so on, to each successive data set that you create. This feature is referred to as the DATAn naming convention.

See “Example: Use the Automatic Naming Feature for Naming Data Sets” on page 66 for an example that shows how to use this feature.

**_LAST_ Data Set**

If you do not specify a name for the input data set in a SET statement, SAS automatically uses the last data set that was created. SAS keeps track of the most recently created data set through the reserved name _LAST_. When you execute a DATA or PROC step without specifying an input data set, by default, SAS uses the _LAST_ data set.

Some functions use the _LAST_ default as well.
The _LAST_= system option enables you to designate a data set as the _LAST_ data set.

See “Example: Use the _LAST_= System Option to Specify the Default Input Data Set” on page 67 for an example that shows how to use the _LAST_ system option.

_NULL_ Data Set

If you want to execute a DATA step but do not want to create a SAS data set, you can specify the keyword _NULL_ as the data set name. The following statement begins a DATA step that does not create a data set:

```sas
data _null_;
```

Using _NULL_ causes SAS to execute the DATA step as if it were creating a new data set, but no observations or variables are written to an output data set. This process can be a more efficient use of computer resources if you are using the DATA step for some function, such as report writing, for which the output of the DATA step does not need to be stored as a SAS data set.

---

Rules for Naming SAS Data Sets and Variables

The table below shows a summary of the rules for naming SAS data sets when the VALIDMEMNAME= system option is set to COMPATIBLE and the VALIDVARNAME= system option is set to V7. These are the default settings in Base SAS. In some SAS applications, such as SAS Visual Analytics, the VALIDMEMNAME= and VALIDVARNAME= system options are set by default to allow the most flexibility for naming SAS variables and data sets. These rules are summarized in Table 4.8.
Table 4.8  Summary of Default Rules for Naming SAS Data Sets and Variables

<table>
<thead>
<tr>
<th>SAS Data Set Names, View Names, and Item Store Names (with VALIDMEMNAME=COMPAT)</th>
<th>Variable Names (with VALIDVARNAME=V7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>can be up to 32 bytes in length.</td>
<td>can be up to 32 bytes in length.</td>
</tr>
<tr>
<td>cannot contain special characters (except for the underscore), blanks, or national characters.</td>
<td>cannot contain special characters (except for the underscore), blanks, or national characters.</td>
</tr>
<tr>
<td>must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore.</td>
<td>must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore.</td>
</tr>
<tr>
<td>can contain mixed-case letters. SAS internally converts the member name to uppercase. Therefore, you cannot use the same member name with a different combination of uppercase and lowercase letters to represent different members. For example, cat, Cat, and CAT all represent the same member name. How the name on the disk appears is determined by the operating environment.</td>
<td>can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable. However, when SAS processes variable names, it internally converts them to uppercase. Therefore, you cannot use the same variable name with a different combination of uppercase and lowercase letters to represent different variables. For example, cat, Cat, and CAT all represent the same variable.</td>
</tr>
</tbody>
</table>

Extended Rules for Naming SAS Data Sets and Variables

The following table shows a summary of the rules for naming SAS data sets when the highest level of flexibility is allowed (that is, when the VALIDMEMNAME= system option is set to EXTEND and the VALIDVARNAME= system option is set to ANY).
Table 4.9  Summary of Extended Rules for Naming SAS Data Sets and Variables

<table>
<thead>
<tr>
<th>SAS Data Set Names, View Names, and Item Store Names (with VALIDMEMNAME=EXTEND)</th>
<th>Variable Names (with VALIDVARNAME=ANY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>can be up to 32 bytes in length.</td>
<td>can be up to 32 bytes in length.</td>
</tr>
<tr>
<td>can contain special characters except for / \ * ? &quot; &lt; &gt;</td>
<td>can contain special characters including / \ * ? &quot; &lt; &gt;</td>
</tr>
<tr>
<td>cannot begin with a blank or a period.</td>
<td>can begin with any character, including blanks, national characters, special characters, and multibyte characters.</td>
</tr>
<tr>
<td>ignores leading and trailing blanks.</td>
<td>preserves leading blanks, but trailing blanks are ignored.</td>
</tr>
<tr>
<td>can contain mixed-case letters. SAS internally converts the member name to uppercase. Therefore, you cannot use the same member name with a different combination of uppercase and lowercase letters to represent different variables. For example, cat, Cat, and CAT all represent the same member name.</td>
<td>can contain mixed-case letters. SAS stores and writes the variable name in the same case that is used in the first reference to the variable. However, when SAS processes a variable name, it internally converts the variable name to uppercase. Therefore, you cannot use the same variable name with a different combination of uppercase and lowercase letters to represent different variables. For example, cat, Cat, and CAT all represent the same variable.</td>
</tr>
<tr>
<td>cannot contain all blanks.</td>
<td></td>
</tr>
</tbody>
</table>

See Also

Examples

- “Example: Create a SAS Data Set Name Containing a Special Character”
- “Example: Create a One-Level Data Set Name”
- “Example: Use the Automatic Naming Feature for Naming Data Sets”
- “Example: Use the _LAST_= System Option to Specify the Default Input Data Set”

Functions

- OPEN function

---

In the UNIX operating environment, SAS reads only data set names that are written in all lowercase characters.
Examples: SAS Words and Names

Example: Create a Variable Name Containing Blanks

Example Code

The following example illustrates creating a variable name that contains blanks

```sas
options validvarname=any;   /*
da mydata;
    'First Name'n='John';   /*
run;
```

1. Specify the **VALIDVARNAME=** system option in the OPTIONS statement. **VALIDVARNAME=ANY** enables you to create a variable name containing a blank, special characters, national characters, and multibyte characters.

2. Specify the variable name as a **name literal**. Name literals enable you to use special characters in SAS **names** when you specify a variable or SAS data set. Name literals are created by placing the name in quotation marks followed by the letter **n**.
Key Ideas

- The `VALIDVARNAME=` system option determines which rules apply to the variables that you can create and process in your SAS session.
- If you use any characters other than the ones that are valid when the `VALIDVARNAME=` system option is set to the default setting (V7), then you must express the variable name as a name literal and you must set `VALIDVARNAME=ANY`.
- Blanks between the closing quotation mark and the `n` are not valid when you specify a name literal.

See Also

- “Definition of SAS Name Literals”
- “OPTIONS Statement” in SAS Global Statements: Reference
- “Rules for Most SAS Names”
- “Rules for Variable Names”
- “VALIDVARNAME= System Option” in SAS System Options: Reference

Example: Create a SAS Data Set Name Containing a Special Character

Example Code

The following example illustrates creating a new SAS data set name that contains the special character `$`:

```sas
options validmemname=extend;   /*1*/
data 'my$data'n;               /*2*/
  x=1;
  y=3;
  sum=x+y;
run;
```

1. Specify the `VALIDMEMNAME=system option OPTIONS` statement. `VALIDMEMNAME=EXTEND` enables you to create a SAS data set name containing any special character except for `/ \ * ? " < > | : -`

2. Specify the data set name as a name literal. Name literals enable you to use special characters in SAS names when you specify a SAS data set name or variable. Name literals are created by placing the name in quotation marks followed by the letter `n`.

60 Chapter 4 / Words and Names
Key Ideas

- A SAS *name literal* is a user-supplied name that is expressed as a *string* within quotation marks, followed by the uppercase or lowercase letter *n*.
- Name literals enable you to use characters (including blanks and national characters) that are not otherwise allowed.
- Use the VALIDMEMNAME=EXTEND system option when specifying a SAS data set name that contains special characters.

See Also

- “Definition of SAS Name Literals”
- “Extended Rules for Naming SAS Data Sets and Variables”
- “OPTIONS Statement” in SAS Global Statements: Reference
- “VALIDMEMNAME= System Option” in SAS System Options: Reference

Example: Manage Placement and Spacing of Words

Example Code

In the following example, blanks are not required because SAS can determine the boundary of every word by examining the beginning of the next word.

```sas
total=x+y;
```

In the following example, the equal sign marks the end of the word `total`. The plus sign, another special character, marks the end of the word `x`. The last special character, the semicolon, marks the end of the `y` word. Though blanks are not needed to end any words in this example, you can add them for readability.

```sas
total = x + y;
```

In the following example, blanks are required because SAS cannot recognize the individual words without the spaces. Without blanks, the entire statement up to the semicolon fits the rules for a name. Therefore, the statement requires blanks to distinguish individual names and numbers.

```sas
input group 15 room 20;
```
**Key Ideas**

- A word in the SAS programming language is a collection of characters that communicates a meaning to SAS.
- A word cannot be divided into smaller units that can be used independently. A word can contain a maximum of 32,767 bytes.
- A name is a series of characters that begin with a letter or an underscore. Later characters can include letters, underscores, and numeric digits. A name can contain up to 32,767 bytes. In most contexts, however, names are limited to a shorter maximum length, such as 32 or 8 bytes.

**See Also**

- "Definition of a SAS Word" on page 40
- "Types of Words or Tokens"
- "Rules for Most SAS Names" on page 42

---

**Example: Specify the FIRST. and LAST. BY Variables as Name Literals**

**Example Code**

The following example illustrates specifying FIRST. and LAST. BY variables as name literals:

```sas
options validvarname=any;                 /*1*/
data sedanTypes;
  set cars;
  by 'Sedan Types'n;                     /*2*/
  if first.Sedan Types'n then type=1;   /*3*/
    else type=2;
run;
```

1. Specify the `VALIDVARNAME=` system option in the `OPTIONS` statement. `VALIDVARNAME=ANY` enables you to create a variable name containing a blank, special characters, national characters, and multibyte characters.

2. The variable `Sedan Types` contains a blank between the two words; therefore, enclose the variable name in quotation marks followed by the letter `n` to declare the variable as a name literal.

3. When you designate a name literal as the BY variable in BY-group processing and you want to refer to the corresponding FIRST. or LAST. temporary variables, use the FIRST. or LAST. portion of the two-level variable name within the quotation marks.
Key Ideas

- The VALIDVARNAME= system option determines which rules apply to the variables that you can create and process in your SAS session.
- If you use any characters other than the ones that are valid when the VALIDVARNAME= system option is set to the default setting (V7), then express the variable name as a name literal and set VALIDVARNAME=ANY.
- Specify your FIRST. and LAST. variable as name literals when there are special characters in the variable name.
- Blanks between the closing quotation mark and the \n are not valid when you specify a name literal.

See Also

- “BY Statement” in SAS DATA Step Statements: Reference
- “OPTIONS Statement” in SAS Global Statements: Reference
- “VALIDMEMNAME= System Option” in SAS System Options: Reference
- “VALIDVARNAME= System Option” in SAS System Options: Reference
- “Using Name Literals in BY Groups”

Example: Create a One-Level Data Set Name

Example Code

The following example illustrates the use of one-level names in SAS statements:

```sas
options user='c:\temp';   /* 1 */
data test;                /* 2 */
  x=1;
run;
data samptest;            /* 3 */
  x=1;
  y=6;
  s=x*y+y;
run;
```

1 USER=system option specifies the name of the default permanent SAS library. When you specify the USER= system option, any data set that you create with a one-level name is permanently stored in the specified library.

2 The DATA statement creates the permanent data set, Test, in the SAS system library, User. Note that you do not have to reference User as a part of your data set name.

3 The DATA statement creates the permanent data set, Samptest, in the current directory.
Note: You do not have to reference the USER= system option prior to the DATA step or reference User as a part of your data set name in order for SAS to create Samptest in the User library.

The following is printed to the SAS log:

Output 4.1  SAS Log: One-Level Data Set Names

```sas
85   options user='C:\temp';
86   data test;
87   x=1;
88   run;

NOTE: The data set USER.TEST has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

89   data samptest;
90   x=1;
91   y=6;
92   s=x*y+y;
93   run;

NOTE: The data set USER.SAMPTEST has 1 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds
```

Key Ideas

- Data sets with one-level names are automatically assigned either the Work or User library. You can omit the libref and refer to data sets with a one-level name.
- Most commonly, data sets are assigned to the temporary library, Work, and are deleted at the end of a SAS job or session.
- If you have associated the libref User with a SAS library or used the USER= system option to set the User library, data sets with one-level names are stored in that library.

See Also

- DATA Statement
- One-Level Names
- OPTIONS Statement
- Structure of a Data Set Name
- USER= System Option
Example: Create a Two-Level Data Set Name

Example Code

The following example illustrates the use of two-level names in SAS statements:

```
libname finance 'C:\finance'; /* 1 */
data finance.balances; /* 2 */
  rate=0.03;
  months=12;
  balance=1000;
  endbal=(rate*months)+balance;
run;
```

1 The **LIBNAME statement** associates a SAS library with the *libref*, Finance. You can reference the *libref* in subsequent DATA or PROC steps to create, read, or write SAS data sets in the permanent library.

2 The **DATA statement** references the Finance *libref* and tells SAS to create a new SAS data set in the Finance library and name the new data set *Balances*. The new data set contains four variables and one observation.

The following is printed to the SAS log:

**Output 4.2  SAS Log: Two-Level Data Set Names**

```
94 libname finance 'C:\finance;
NOTE: Libref FINANCE was successfully assigned as follows:
   Engine:         V9
   Physical Name:  C:\finance
95 96 data finance.balances;
97   rate=.03;
98   months=12;
99   balance=1000;
 100  endbal=(rate*months)+balance;
 101  run;
```

NOTE: The data set FINANCE.BALANCES has 1 observations and 4 variables.

<table>
<thead>
<tr>
<th>Time</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>real time</td>
<td>0.06 seconds</td>
</tr>
<tr>
<td>cpu time</td>
<td>0.01 seconds</td>
</tr>
</tbody>
</table>

**Key Ideas**

- A two-level data set name is the most commonly used form to create, read, or write SAS data sets in a permanent SAS library.
- When you create a new SAS data set, the *libref* indicates where it is to be stored. When you reference an existing data set, the *libref* tells SAS where to find it.
See Also

- DATA Statement
- LIBNAME Statement
- Structure of a Data Set Name
- Two-Level Names

Example: Use the Automatic Naming Feature for Naming Data Sets

Example Code

The following example illustrates the use of the DATAn automatic naming feature. If you do not specify a name for the output data set, SAS automatically assigns a name to the data set that you create. SAS assigns a successive name to every unnamed data set. The example contains three unnamed output data sets and one named data set.

```sas
data;
    set sashelp.class;
run;

data;
    set sashelp.air;
run;
data sample;
    set sashelp.cars;
run;
data;
    set sashelp.flags;
run;
```

The following is printed to the SAS log. Note that the three unnamed output data sets are stored in

**Output 4.3  SAS Log: Use the DATAn Automatic Naming Feature**

- NOTE: There were 19 observations read from the data set SASHELP.CLASS.
- NOTE: The data set WORK.DATA3 has 19 observations and 5 variables.
- NOTE: There were 144 observations read from the data set SASHELP.AIR.
- NOTE: The data set WORK.DATA4 has 144 observations and 2 variables.
- NOTE: There were 428 observations read from the data set SASHELP.CARS.
- NOTE: The data set WORK.CARS has 428 observations and 15 variables.
- NOTE: There were 220 observations read from the data set SASHELP.FLAGS.
- NOTE: The data set WORK.DATA5 has 220 observations and 7 variables.
Key Ideas

- If you do not specify a SAS data set name in a DATA statement, SAS automatically creates data sets with the names DATA1, DATA2, and so on, to successive data sets.
- The automatically named data sets are stored in the Work or User library.
- If you do not save your automatically named data sets to a permanent library, they will be removed at the end of your SAS session.

See Also

- DATAn Data Set
- DATA Statement
- SET Statement

Example: Use the _LAST_= System Option to Specify the Default Input Data Set

Example Code

The following example illustrates the use of the _LAST_= system option to specify the default input data set for the current SAS session:

```
options _last_=mysas.class; /*1*/
libname mysas 'C:\Users\';
data mysas.class; /*2*/
    set sashelp.class;
run;
data mysas.test; /*3*/
    set;
run;
```

1 The _LAST_= system option enables you to designate a data set as the _LAST_ data set. The name that you specify is used as the default data set until you create a new data set. You can use the _LAST_= system option when you want to use an existing permanent data set for a SAS job that contains a number of procedure steps. Issuing the _LAST_= system option enables you to avoid specifying the SAS data set name in each procedure statement.

2 The first DATA step creates a data set named mysas.class that is referenced in the _LAST_= system option.

3 The second DATA step creates a permanent data set named mysas.test and does not specify a data set in the SET statement. The SET statement then executes the _LAST_= system option and sets the input data set as mysas.class.
Output 4.4  SAS Log Output

NOTE: There were 19 observations read from the data set MYSAS.CLASS.
NOTE: The data set MYSAS.CLASS has 19 observations and 5 variables.

Key Ideas

- SAS keeps track of the most recently created SAS data set through the reserved name _LAST_.
- When you execute a DATA or PROC step without specifying an input data set, by default, SAS uses the _LAST_ data set. Some functions use the _LAST_ default as well.
- The _LAST_= system option enables you to designate a data set as the _LAST_ data set. The name that you specify is used as the default data set until you create a new data set.
- You can use the _LAST_= system option when you want to use an existing permanent data set for a SAS job that contains a number of DATA step or procedure steps.
- Specifying the _LAST_= system option enables you to avoid specifying the SAS data set name in each DATA step or procedure statement.

See Also

- Data Set Names
- _LAST_ Data Set
- LIBNAME Statement
- OPTIONS Statement
- SET Statement
Chapter 5 / Variables

Examples: Create and Modify SAS Variables .............................................. 111
  Example: Create a SAS Variable Using an Assignment Statement ............. 111
  Example: Create a Variable Using the INPUT Statement ......................... 112
  Example: Modify a SAS Variable Using the LENGTH Statement ................. 112
  Example: Modify a SAS Variable Using the FORMAT and INFORMAT Statement 113
  Example: Modify a SAS Variable Using the ATTRIB Statement ................. 115
  Example: View Variable Attributes .................................................. 115
  Example: Change Variable Attributes .............................................. 117

Examples: Control Output of Variables .................................................. 119
  Example: Select Specific Variables for Output .................................... 119
  Example: Select Specific Variables for Processing .................................. 121

Examples: Reorder and Align Variables .................................................. 122
  Example: Reorder Variables Using the ATTRIB Statement ....................... 122
  Example: Reorder Variables Using the LENGTH Statement ...................... 124
  Example: Reorder Variables Using the RETAIN Statement ....................... 126
  Example: Reorder Variables Using the FORMAT Statement ....................... 128

Examples: Convert Variable Types ....................................................... 129
  Example: Convert Variable Types Using the PUT and INPUT Functions or Use the PUT and INPUT Functions to Convert Variable Types .............................. 129
  Example: Convert Character Variables to Numeric .................................. 131
  Example: Convert Numeric Variables to Character .................................. 132
  Example: Use Automatic Type Conversions ......................................... 134

Examples: Use Automatic Variables ...................................................... 135
  Example: Use the _N_ Automatic Variable ......................................... 135
  Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/THEN Statement .................................................. 136

Examples: Use Variable Lists ............................................................... 138
  Example: Create a Variable Numbered Range List .................................. 138
  Example: Create a Variable Name Range List ....................................... 139
  Example: Use the OF Operator with a Variable List ............................... 140
  Example: Use the OF Operator to Create Multiple Variable Lists ............... 141

Examples: Manage Missing Variable Values .......................................... 142
  Example: Automatically Replacing Missing Values .................................. 142
  Example: Creating Special Missing Values ......................................... 143
  Example: Preventing Propagation of Missing Values .............................. 144

Examples: Manage Problems Related to Precision .................................. 146
  Example: Compare Imprecise Values in SAS ....................................... 146
  Example: Convert a Decimal Value to a Floating Point Representation ........ 147
  Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation .......................................................... 149
  Example: Round Values to Avoid Computational Errors ........................... 151
  Example: Use the LENGTH Statement to Compare Values ....................... 152

Examples: Reorder and Align Variables .................................................. 122
  Example: Reorder Variables Using the ATTRIB Statement ....................... 122
  Example: Reorder Variables Using the LENGTH Statement ...................... 124
  Example: Reorder Variables Using the RETAIN Statement ....................... 126
  Example: Reorder Variables Using the FORMAT Statement ....................... 128

Examples: Convert Variable Types ....................................................... 129
  Example: Convert Variable Types Using the PUT and INPUT Functions or Use the PUT and INPUT Functions to Convert Variable Types .............................. 129
  Example: Convert Character Variables to Numeric .................................. 131
  Example: Convert Numeric Variables to Character .................................. 132
  Example: Use Automatic Type Conversions ......................................... 134

Examples: Use Automatic Variables ...................................................... 135
  Example: Use the _N_ Automatic Variable ......................................... 135
  Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/THEN Statement .................................................. 136

Examples: Use Variable Lists ............................................................... 138
  Example: Create a Variable Numbered Range List .................................. 138
  Example: Create a Variable Name Range List ....................................... 139
  Example: Use the OF Operator with a Variable List ............................... 140
  Example: Use the OF Operator to Create Multiple Variable Lists ............... 141

Examples: Manage Missing Variable Values .......................................... 142
  Example: Automatically Replacing Missing Values .................................. 142
  Example: Creating Special Missing Values ......................................... 143
  Example: Preventing Propagation of Missing Values .............................. 144

Examples: Manage Problems Related to Precision .................................. 146
  Example: Compare Imprecise Values in SAS ....................................... 146
  Example: Convert a Decimal Value to a Floating Point Representation ........ 147
  Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation .......................................................... 149
  Example: Round Values to Avoid Computational Errors ........................... 151
  Example: Use the LENGTH Statement to Compare Values ....................... 152

Truncation in Binary Numbers ............................................................. 99
How SAS Stores Numeric Values .......................................................... 100
Floating-Point Representation Using the IEEE Standard ......................... 104
Floating-Point Representation on Windows ............................................ 105
Floating-Point Representation on IBM Mainframes ................................ 106
Troubleshooting Errors in Precision .................................................... 107
Transferring Data between Operating Systems ..................................... 109
See Also ......................................................................................... 110

Examples: Create and Modify SAS Variables .............................................. 111
  Example: Create a SAS Variable Using an Assignment Statement ............. 111
  Example: Create a Variable Using the INPUT Statement ......................... 112
  Example: Modify a SAS Variable Using the LENGTH Statement ................. 112
  Example: Modify a SAS Variable Using the FORMAT and INFORMAT Statement 113
  Example: Modify a SAS Variable Using the ATTRIB Statement ................. 115
  Example: View Variable Attributes .................................................. 115
  Example: Change Variable Attributes .............................................. 117

Examples: Control Output of Variables .................................................. 119
  Example: Select Specific Variables for Output .................................... 119
  Example: Select Specific Variables for Processing .................................. 121

Examples: Reorder and Align Variables .................................................. 122
  Example: Reorder Variables Using the ATTRIB Statement ....................... 122
  Example: Reorder Variables Using the LENGTH Statement ...................... 124
  Example: Reorder Variables Using the RETAIN Statement ....................... 126
  Example: Reorder Variables Using the FORMAT Statement ....................... 128

Examples: Convert Variable Types ....................................................... 129
  Example: Convert Variable Types Using the PUT and INPUT Functions or Use the PUT and INPUT Functions to Convert Variable Types .............................. 129
  Example: Convert Character Variables to Numeric .................................. 131
  Example: Convert Numeric Variables to Character .................................. 132
  Example: Use Automatic Type Conversions ......................................... 134

Examples: Use Automatic Variables ...................................................... 135
  Example: Use the _N_ Automatic Variable ......................................... 135
  Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/THEN Statement .................................................. 136

Examples: Use Variable Lists ............................................................... 138
  Example: Create a Variable Numbered Range List .................................. 138
  Example: Create a Variable Name Range List ....................................... 139
  Example: Use the OF Operator with a Variable List ............................... 140
  Example: Use the OF Operator to Create Multiple Variable Lists ............... 141

Examples: Manage Missing Variable Values .......................................... 142
  Example: Automatically Replacing Missing Values .................................. 142
  Example: Creating Special Missing Values ......................................... 143
  Example: Preventing Propagation of Missing Values .............................. 144

Examples: Manage Problems Related to Precision .................................. 146
  Example: Compare Imprecise Values in SAS ....................................... 146
  Example: Convert a Decimal Value to a Floating Point Representation ........ 147
  Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation .......................................................... 149
  Example: Round Values to Avoid Computational Errors ........................... 151
  Example: Use the LENGTH Statement to Compare Values ....................... 152
Definition of SAS Variables

automatic variables
are created automatically by the DATA step or by the DATA step statements. These variables are added to the program data vector but are not included in the data set being created. You cannot view these variables in the log or in the results.

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

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.

uninitialized variables
refers to variables that are not present in the input data set or have not been created within the DATA step where SAS tries to use the variable.

Ways to Create Variables

You can create a variable using the assignment statement or read data with the INPUT statement in a DATA step.

The assignment statement evaluates an expression and stores the result in a variable. The INPUT statement describes the arrangement of values in the input data record and assigns input values to the corresponding SAS variables.

Note: There are more ways to create variables. For example, the SET, MERGE, MODIFY, and UPDATE statements can also create variables.
<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment statement</td>
<td>data vars;</td>
<td>The assignment statement creates a character variable called a. When a LENGTH statement is not used, the length of the first literal encountered determines the length of the character variable. The PUT statement output for variable a is truncated as Rita Wils.</td>
</tr>
<tr>
<td></td>
<td>a='Tom Hanks';</td>
<td>The assignment statement creates a new numeric variable called a and its value is 36. The default length for numeric variables is 8 bytes, unless otherwise specified.</td>
</tr>
<tr>
<td></td>
<td>a='Rita Wilson';</td>
<td></td>
</tr>
<tr>
<td></td>
<td>put a;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>run;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Example: Create a SAS Variable Using an Assignment Statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data vars;</td>
<td>The LENGTH statement creates three character variables, a, b, and c with respective lengths of 4, 6, and 2 bytes. The assignment statement creates variable x with a length of 12, which is the sum of the lengths of all the variables on the right side of the assignment operator. The value for variable x is the value that is produced when you concatenate the variables a, b, and c.</td>
</tr>
<tr>
<td></td>
<td>length a $ 4 b $ 6 c $ 2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x=a</td>
<td></td>
</tr>
<tr>
<td></td>
<td>run;</td>
<td></td>
</tr>
<tr>
<td>INPUT statement, List</td>
<td>data vars;</td>
<td>The INPUT statement creates two character variables: Gem and Color, and one numeric variable Carats. The variables are defined based on their positions in the raw data. The length of each variable is 8 bytes. If you create a character variable using the INPUT statement and you do not otherwise specify the variable’s length (for example, in a length statement or in the INPUT statement), then SAS automatically sets the length of the variable to 8 bytes.</td>
</tr>
<tr>
<td></td>
<td>input Gem $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Carats</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Color $</td>
<td></td>
</tr>
<tr>
<td></td>
<td>datalines;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>emerald 2 green</td>
<td></td>
</tr>
<tr>
<td></td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>Method</td>
<td>Example</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>INPUT statement, Formatted</td>
<td>data vars;</td>
<td>The INPUT statement creates two character variables: Gem is created with a length specified in its informat, and Color is created without a length specification. The numeric variable Carats is created using a numeric informat with its length specified. The variable types are defined based on the category of informat that is specified in the INPUT statement. The length of the numeric variable, Carats is 8 bytes. The length of the character variable, Gem, is 10 bytes. The length of the character variable, Color, is 8 bytes. See Creating a New Variable Using the INPUT, Formatted Statement for information about how SAS determines the lengths of variables that are created using formatted input.</td>
</tr>
</tbody>
</table>

Example: Create a Variable Using the INPUT Statement

```
Example: Create a Variable Using the INPUT Statement
```

See Informats by Category for a list of these categories.

---

**Creating a New Variable in a Formatted INPUT Statement**

You can use various forms of the INPUT statement to create a new variable when you read raw data into SAS. If the variable does not already exist and you create it for the first time in a formatted INPUT statement, then SAS defines the variable and its attributes based on the category of the informat specified in the INPUT statement:

- Associating a numeric informat with a variable when it is created for the first time in a DATA step using formatted input causes the variable to be created as a numeric type, with a default length of 8.

- Associating a character informat with a variable when it is created for the first time in a DATA step using formatted input causes the variable to be created as a character type. The length matches the width specified in the informat in the INPUT statement. If you do not specify a length with the informat or anywhere else in the DATA step, then SAS assigns the default length of 8 bytes.
Manage Variables

Modify Variables

You can modify SAS variables using the following methods:

- FORMAT or INFORMAT statement
- LENGTH statement
- ATTRIB statement

<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| FORMAT or INFORMAT statement  | Insurance=100.28; Fee=20.00; format Insurance Fee dollar10.2; | You can create new variables and specify their format or informat with a FORMAT or INFORMAT statement.¹  
<p>|                               | Example: FORMAT Statement                    | The FORMAT statement creates two variables Insurance and Fee with a format of dollar 10.2  |</p>
<table>
<thead>
<tr>
<th>Method</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LENGTH statement</td>
<td><code>length Name $20;</code></td>
<td>For character variables, you must use the longest possible value in the first statement that uses the variable. The reason is that you cannot change the length with a subsequent LENGTH statement within the same DATA step. The maximum length of any character variable in SAS 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 to make it match the length of the target variable.</td>
</tr>
</tbody>
</table>
### Method | Example | Explanation
--- | --- | ---
ATTRIB statement | `attrib Class format=$12.;` | 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.

The ATTRIB statement creates a new variable named `Class`, with a format.

**Note:** You cannot create a new variable by using a `LABEL` statement or the ATTRIB statement’s `LABEL=` attribute by itself. Labels can be applied only to existing variables.

---

1 See “Specifying a New Variable in a FORMAT or INFORMAT Statement” for information about how SAS determines variable attributes when variables are created using the FORMAT or INFORMAT statement.

---

### Specifying a New Variable in a FORMAT or INFORMAT Statement

If the variable does not already exist and you create it for the first time in a FORMAT or INFORMAT statement, then SAS defines the variable and its attributes based on the format or informat specified:

- **Associating a numeric** format or informat with a variable when it is created for the first time in a DATA step using the FORMAT or INFORMAT statement causes the variable to be created as a numeric type, with a default length of 8.

- **Associating a character** format or informat with a variable when it is created for the first time in a DATA step using in the FORMAT or INFORMAT statement causes the variable to be created as a character type. The length matches the width of the format or informat that you specify as part of the format or informat. If you do not specify a length, then SAS assigns the default length of 8 bytes.
See Formats by Category and Informats by Category for a list of these categories.

In the FORMAT statement creates the character variable `Flavor` and the numeric variable `Amount`. These two variables appear for the first time in the FORMAT statement, so SAS determines their type based on the category of format that is assigned to them. Since the variable `Amount` is associated with numeric type format (COMMA,wd), SAS defines it as a numeric type variable, with a default length of 8.

The `Flavor` variable is defined as a character type variable because the `$UPCASE` format is a character type format. When character variables are created using the FORMAT statement, SAS determines their length first based on the length that you specify in the FORMAT statement. If you do not specify a length with the format or anywhere else in the DATA step, then SAS gives the variable a default length of 8. In this example, the format does not include a length specification, so the length for the variable `Flavor` is 8 bytes.

**Example Code 5.1** Specifying a New Variables Using the FORMAT Statement (Without Specifying Lengths)

```sas
data lollipops;
  format Flavor $upcase. Amount comma.;
  Flavor='Cherry';
  Amount=10;
run;
proc contents data=lollipops; run;
```

The next example is identical except that a length is specified for both variables in the FORMAT statement along with the format:

**Output 5.1** PROC CONTENTS Output for Creating New Variables Using the FORMAT Statement (Without Specifying Lengths)

Alphabetic List of Variables and Attributes

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Amount</td>
<td>Num</td>
<td>8</td>
<td>COMMA.</td>
</tr>
<tr>
<td>1</td>
<td>Flavor</td>
<td>Char</td>
<td>8</td>
<td>$UPCASE.</td>
</tr>
</tbody>
</table>

**Example Code 5.2** Specifying New Variables Using the FORMAT Statement (With Length Specified)

```sas
data lollipops;
  format Flavor $upcase10. Amount comma10.;
  Flavor='Cherry';
  Amount=10;
run;
proc contents data=lollipops; run;
```

**Output 5.2** PROC CONTENTS Output for Specifying New Variables Using the FORMAT Statement (With Length Specified)

Alphabetic List of Variables and Attributes

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Amount</td>
<td>Num</td>
<td>8</td>
<td>COMMA10.</td>
</tr>
<tr>
<td>1</td>
<td>Flavor</td>
<td>Char</td>
<td>10</td>
<td>$UPCASE10.</td>
</tr>
</tbody>
</table>
In the example, Example Code 5.3 on page 78, the variables appear for the first time in an assignment statement rather than in the FORMAT statement. So, in this case, the assignment statements create the variables. When a variable appears for the first time on the left side of an assignment statement, SAS automatically sets its type and length based on the expression on the right side of the assignment statement. Since the expression on the right is the 5–letter character string Cherry, SAS assigns a length of 5 bytes and the type character to the variable Flavor. SAS assigns a length of 8 bytes to the numeric variable, Amount.

Example Code 5.3  Changing the Format of an Existing Variable Using the FORMAT Statement

```sas
data lollipops;
  Flavor='Cherry';
  Amount=10;
  format Flavor $upcase10. Amount comma10.;
run;
proc contents data=lollipops; run;
```

Output 5.3  PROC CONTENTS Output for Changing the Format of an Existing Variable Using the FORMAT Statement

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Amount</td>
<td>Num</td>
<td>8</td>
<td>COMMA10.</td>
</tr>
<tr>
<td>1</td>
<td>Flavor</td>
<td>Char</td>
<td>6</td>
<td>$UPCASE10.</td>
</tr>
</tbody>
</table>

See Also

- Table 5.5 on page 82 for more information about how SAS determines variable attributes with assignment statements.
- SAS Formats and Informats: Reference

Specifying a New Variable in the ATTRIB Statement

If the variable does not already exist and you create it for the first time using either the FORMAT= or INFORMAT= option in the ATTRIB statement, then SAS defines the variable and its attributes based on the format or informat specified in the option statement.

- Associating a numeric format or informat with a variable when it is created for the first time in a DATA step using the FORMAT or INFORMAT statement causes the variable to be created as a numeric type, with a default length of 8.
- Associating a character format or informat with a variable when it is created for the first time in a DATA step using in the FORMAT or INFORMAT statement causes the variable to be created as a character type. The length matches the width of the format or informat that you specify as part of the format or informat. If you do not specify a length, then SAS assigns the default length of 8 bytes.

See Formats by Category and Informats by Category for a list of these formats and informats.
Control Output of 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 that you want. The action taken by SAS depends largely on whether you perform one of the following actions:

- 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>
<thead>
<tr>
<th>Statements</th>
<th>Data Set Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply to output data sets only</td>
<td>apply to output or input data sets</td>
</tr>
<tr>
<td>affect all output data sets</td>
<td>affect individual data sets</td>
</tr>
<tr>
<td>can be used in DATA steps only</td>
<td>can be used in DATA steps and PROC steps</td>
</tr>
<tr>
<td>can appear anywhere in DATA steps</td>
<td>must immediately follow the name of each data set to which they apply</td>
</tr>
</tbody>
</table>

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 5.4 Status of Variables and Variable Names When Dropping, Keeping, and Renaming Variables**

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

Your program might require 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.

See Also

Examples

- Using the DROP= Data Set Option and DROP Statement
- Using the DROP= and RENAME= Data Set Options

Statements

- DROP Statement

Data Set Options

- DROP= Data Set Option
- RENAME= Data Set Option

Variable Attributes

Definition

*SAS variables* are containers that you create within a program to store and use character and numeric values. Variables have the following attributes:
### Table 5.5  Variable Attributes

<table>
<thead>
<tr>
<th>Variable Attribute</th>
<th>Definition</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>identifies a variable. A variable name must conform to SAS naming rules.</td>
<td>Any valid SAS name.</td>
<td>None</td>
</tr>
</tbody>
</table>
| 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. | Numeric and character. | Numeric |
| 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.  
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. | 2 to 8 bytes for numeric.  
1 to 32,767 bytes for character. | 8 bytes for numeric and character |
<table>
<thead>
<tr>
<th>Variable Attribute</th>
<th>Definition</th>
<th>Possible Values</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Format</strong></td>
<td>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.</td>
<td>See Dictionary of Formats in SAS Formats and Informats: Reference</td>
<td>BEST12. for numeric, $w. for character</td>
</tr>
<tr>
<td><strong>Informat</strong></td>
<td>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 also assign an informat in the INPUT statement or INPUT function.</td>
<td>See About Informats in SAS Formats and Informats: Reference</td>
<td>w.d for numeric, $w. for character</td>
</tr>
<tr>
<td><strong>Label</strong></td>
<td>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.</td>
<td>Up to 256 characters.</td>
<td>None</td>
</tr>
<tr>
<td><strong>Position in observation</strong></td>
<td>is determined by the order in which the variables are defined in the DATA step.</td>
<td>1- n</td>
<td>None</td>
</tr>
<tr>
<td>Variable Attribute</td>
<td>Definition</td>
<td>Possible Values</td>
<td>Default Value</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Index type</td>
<td>indicates whether the variable is part of an index for the data set.</td>
<td>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.</td>
<td>None</td>
</tr>
<tr>
<td>Extended attribute</td>
<td>is a user-defined attribute that is created using the XATTR ADD VAR statement in the DATASETS procedure.</td>
<td>Numeric or character</td>
<td>None</td>
</tr>
</tbody>
</table>

**Note:** Starting with SAS 9.1, the maximum number of variables can be greater than 32,767. The maximum number depends on your environment and the file’s attributes. For example, the maximum number of variables depends on the total length of all the variables and cannot exceed the maximum page size.

**See Also**

**Examples**
- View Variable Attributes
- Change Attributes of a Variable Using the ATTRIB Statement

**Statements**
- ATTRIB Statement
- FORMAT Statement
- INFORMAT Statement

**Procedures**
- CONTENTS Procedure
Data Types

Definitions

A data type is an attribute of every SAS variable that specifies what type of data the variable stores. The data type identifies a piece of data as a character string, an integer, a floating-point number, or a date or time, for example. The data type also determines how much memory to allocate for the variable’s values. The default SAS engine is the V9 engine, or the Base SAS engine. In Base SAS, the following data types are supported by the DATA step.

- **data values**
  - are character or numeric values.

- **numeric value**
  - contains only numbers, and sometimes a decimal point, a minus sign, or both.
  - When they are read into a SAS data set, numeric values are stored in the floating-point format native to the operating environment. Nonstandard numeric values can contain other characters as numbers; you can use formatted input to enable SAS to read them.

- **character value**
  - is a sequence of characters.

- **standard data**
  - are character or numeric values that can be read with list, column, formatted, or named input.

- **nonstandard data**
  - is data that can be read only with the aid of informats. Examples of nonstandard data include numeric values that contain commas, dollar signs, or blanks; date and time values; and hexadecimal and binary values.

Numeric Data

Numeric data can be represented in several ways. SAS can read standard numeric values without any special instructions. To read nonstandard values, SAS requires special instructions in the form of informats. Table 5.7 on page 86 shows standard, nonstandard, and invalid numeric data values and the special tools, if any, that are required to read them. For complete descriptions of all SAS informats, see *SAS Formats and Informats: Reference*.

*Table 5.6  Reading Standard Numeric Data*

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Informat Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>input right aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>Data</td>
<td>Description</td>
<td>Informat Needed</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>23</td>
<td>input not aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>23</td>
<td>input left aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>00023</td>
<td>input with leading zeros</td>
<td>None needed</td>
</tr>
<tr>
<td>23.0</td>
<td>input with decimal point</td>
<td>None needed</td>
</tr>
<tr>
<td>2.3E1</td>
<td>in E notation, 2.30 (ss1)</td>
<td>None needed</td>
</tr>
<tr>
<td>230E-1</td>
<td>in E notation, 230x10 (ss-1)</td>
<td>None needed</td>
</tr>
<tr>
<td>-23</td>
<td>minus sign for negative</td>
<td>None needed</td>
</tr>
<tr>
<td></td>
<td>numbers</td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.7  Reading Nonstandard Numeric Data**

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3</td>
<td>embedded blank</td>
<td>COMMA. or BZ. informat</td>
</tr>
<tr>
<td>- 23</td>
<td>embedded blank</td>
<td>COMMA. or BZ. informat</td>
</tr>
<tr>
<td>2,341</td>
<td>comma</td>
<td>COMMA. informat</td>
</tr>
<tr>
<td>(23)</td>
<td>parentheses</td>
<td>COMMA. informat</td>
</tr>
<tr>
<td>C4A2</td>
<td>hexadecimal value</td>
<td>HEX. informat</td>
</tr>
<tr>
<td>1MAR90</td>
<td>date value</td>
<td>DATE. informat</td>
</tr>
</tbody>
</table>

**Table 5.8  Reading Invalid Numeric Data**

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 -</td>
<td>minus sign follows number</td>
<td>Put minus sign before number or solve programmatically. It might be possible to use the S370FZDTw.d informat, but positive values require the trailing plus sign (+). You can also use the TRAILSIGN informat.</td>
</tr>
</tbody>
</table>
## Data Types

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>. .</td>
<td>double instead of single periods</td>
<td>Code missing values as a single period or use the ?? modifier in the INPUT statement to code any invalid input value as a missing value.</td>
</tr>
<tr>
<td>J23</td>
<td>not a number</td>
<td>Read as a character value, or edit the raw data to change it to a valid number.</td>
</tr>
</tbody>
</table>

### Character Data

A value that is read with an INPUT statement is assumed to be a character value if one of the following is true:

- A dollar sign ($) follows the variable name in the INPUT statement.
- A character informat is used.
- The variable has been previously defined as character. For example, a value is assumed to be a character value if the variable has been previously defined as character in a LENGTH statement, in the RETAIN statement, by an assignment statement, or in an expression.

Input data that you want to store in a character variable can include any character. Use the guidelines in the following table when your raw data includes leading blanks and semicolons.

#### Table 5.9  Reading Instream Data and External Files Containing Leading Blanks and Semicolons

<table>
<thead>
<tr>
<th>Characters in the Data</th>
<th>What to Use</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>leading or trailing blanks that you want to preserve</td>
<td>formatted input and the $CHARw. informat</td>
<td>List input trims leading and trailing blanks from a character value before the value is assigned to a variable.</td>
</tr>
<tr>
<td>semicolons in instream data</td>
<td>DATALINES4 or CARDS4 statements and four semicolons (;;;;) to mark the end of the data</td>
<td>With the normal DATALINES and CARDS statements, a semicolon in the data prematurely signals the end of the data.</td>
</tr>
</tbody>
</table>
## Variable Type Conversions

### Automatic Character-to-Numeric Conversion

By default, if you reference a character variable in a numeric context such as an arithmetic operation, SAS tries to convert the variable values to numeric. If you use a character variable with an operator that requires numeric operands, such as the plus sign, SAS converts the character variable to numeric. If you use a comparison operator, such as the equal sign, to compare a character variable and a numeric variable, the character variable is converted to numeric.

In the example below, the character variable `Rate` appears in a numeric context. It is multiplied by the numeric variable `Hours` to create a new variable named `Salary`.

```plaintext
Salary=Rate*Hours;
```

When this step executes, SAS automatically attempts to convert the character values of `Rate` to numeric values so that the calculation can occur. This conversion is completed by creating a temporary numeric value for each character value of `Rate`. This temporary value is used in the calculation. The character values of `Rate` are not replaced by numeric values. Whenever data is automatically converted, a message is written to the SAS log stating that the conversion has occurred.

**Example Code 5.1  SAS Log**

```plaintext
NOTE: Character values have been converted to numeric values at the places given by: (Line):(Column).
```

```
9248:8
```

**Note:** If converting a character variable to numeric produces invalid numeric values, SAS assigns a missing value to the result, prints an error message in the log, and sets the value of the automatic variable `_ERROR_` to 1.
Explicit Character-to-Numeric Conversion

You can use the INPUT function to explicitly convert character data values to numeric values. When choosing the informat, be sure to select a numeric informat that can read the form of the values. In the example below, you can use the INPUT function to explicitly convert `Rate` to numeric values. `Rate` has a length of 2, the numeric informat 2. is used to read the values of the variable.

```sas
input (payrate,2.);
```

No conversion messages appear in the SAS log when the INPUT function is used.

Automatic Numeric-to-Character Conversion

The automatic conversion of numeric data to character data is very similar to character-to-numeric conversion. Numeric data values are converted to character values whenever they are used in a character context. Specifically, SAS writes the numeric value with the BEST12. format, and the resulting character value is right-aligned. This conversion occurs before the value is assigned or used with any operator or function. However, automatic numeric-to-character conversion can cause unexpected results. For example, suppose the original numeric value has fewer than 12 digits. The resulting character value has leading blanks, which might cause problems when you perform an operation or function.

Automatic numeric-to-character conversion also causes a message to be written to the SAS log indicating that the conversion has occurred.

In the example below, you want to create a new character variable named `Loc` that concatenates the values of the numeric variable `Site` and the character variable `Dept`. The new variable values must contain the value of `Site` with a slash, and then the value of `Dept`.

```
Loc=Site||'/'||Dept;
```

Submitting this statement in a DATA step causes SAS to automatically convert the numeric values of `Site` to character values because `Site` is used in a character context. The variable `Site` appears with the concatenation operator, which requires character values.

Explicit Numeric-to-Character Conversion

Use the PUT function to explicitly convert numeric data values to character data values. The PUT function in your assignment statement converts the numeric data values to character. In the example below, to explicitly convert the numeric values of `Site` to character values, use the PUT function in an assignment statement, where `Site` is the source variable. Because `Site` has a length of 2, choose 2. as the numeric format.

```
loc=put(site,2.)||'/'||Dept;
```

No conversion messages appear in the SAS log when you use the PUT function.
Automatic Variables

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

.Cmd_
contains the last command from the window’s command line that was not recognized by the window. This automatic variable is created in the DATA step when you use the WINDOW statement to create customized windows for your applications.

Example: _CMD_ Example

_ERROR_
is 0 by default but is set to 1 whenever an error is encountered. Examples of errors include 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.

_IORC_
The automatic variable _IORC_ contains the return code for each I/O operation that the MODIFY statement attempts to perform. The best way to test for values of _IORC_ is with the mnemonic codes that are provided by the SYSRC autocall macro. Each mnemonic code describes one condition. The mnemonics provide an easy method for testing problems in a DATA step program. This automatic variable is created in the DATA step when you use the MODIFY statement to replace, delete, and append observations in an existing SAS data set.

Example: Example: Controlling I/O

_N_
is initially set to 1. Each time the DATA step loops past the DATA statement, the variable _N_ increments by 1. The value of _N_ represents the number of times the DATA step has iterated.
_MSG_
contains a message that you specify to be displayed in the message area of the
window. This automatic variable is created in the DATA step when you use the
WINDOW statement to create customized windows for your applications.

Example: _MSG_ Example

FIRST. variable
are variables that SAS creates for each BY variable. SAS sets FIRST. variable
when it is processing the first observation in a BY group.

Example  “Example: Specify the FIRST. and LAST. BY Variables as Name
Literals” on page 62

LAST. variable
are variables that SAS creates for each BY variable. SAS sets LAST. variable
when it is processing the last observation in a BY group.

Example  “Example: Specify the FIRST. and LAST. BY Variables as Name
Literals” on page 62

The names of automatic variables are reserved for the system-generated variables
that are created when the DATA step executes. It is recommended that you do not
use names that start and end with an underscore in your own applications.

See Also

Examples

■ “Example: Use the _N_ Automatic Variable”
■ “Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/
THEN Statement”

Statements

■ “_INFILE_=variable” in SAS DATA Step Statements: Reference

SAS Variable Lists

Definition of a Variable List

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

■ numbered range lists
■ name range lists
■ name prefix lists
Special SAS Name Lists

SAS keeps track of active variables in the order in which the compiler encounters them within a DATA step. With name variable lists, you refer to the variables in a variable list in the same order.

You can use variable lists when you define variables. You can use variable lists in SAS statements and data set options in the DATA step. Variable lists are useful because they provide a quick way to reference existing groups of data.

Note: Only the numbered range list are supported in the RENAME= option.

Types of Variable Lists

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
</table>
| Numbered range variable list| Lists that consist of variables that are prefixed with the same name, but have different numeric values for the last characters. The values must be consecutive. | input var1-var6;  
Note that this is equivalent to this numbered list that is not specified as a range:  
input var1 var2 var3 var4 var5 var6; |
| Name range lists            | Name range lists rely on the order of variable definition.                  | x--b               
Specifies all columns ordered as they are in the program data vector, from x to b, inclusive. |
| Name prefix lists           | Some SAS functions and statements enable you to use a name prefix list to refer to all variables that begin with a specified character string. | sum(of Sales:)  
This character string 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_, _CHARACTER_, _ALL_.              |                    |
The OF Operator with Functions and Variable Lists

Definition

The OF operator enables you to specify SAS variable lists or SAS arrays as arguments to functions. Here is the syntax for functions used with the OF operator:

FUNCTION (OF variable-list)

FUNCTION (<argument | OF variable-list | OF array-name[*]><..., <argument | OF variable-list | OF array-name[*]>)

The following table shows the types of SAS variable lists that are valid with the OF operator:

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name range lists</td>
<td>Function(OF x-character-a)</td>
<td>Performs the function on all the character variables from x to a inclusive.</td>
</tr>
<tr>
<td>Name prefix lists</td>
<td>Function(OF x:)</td>
<td>Performs the function on all the variables that begin with “x” such as “x1”, “x2” and so on.</td>
</tr>
<tr>
<td>Numbered range lists</td>
<td>Function(OF x1 – xn)</td>
<td>Performs the function on variable values between x1 and xn inclusive.¹</td>
</tr>
<tr>
<td>Arrays</td>
<td>Function(OF array-name(*))</td>
<td>Performs the function on the named array.²</td>
</tr>
<tr>
<td>Special SAS name lists</td>
<td>Function(OF <em>numeric</em>)</td>
<td>Performs the function on the _numeric_variable, which specifies all numeric variables that are already defined in the current DATA step.</td>
</tr>
</tbody>
</table>

¹ Requires you to have a series of variables with the same name except for the last character or characters, which are consecutive numbers.

² If array-name is a temporary array, there are limitations. See “Using the OF Operator with Temporary Arrays” in SAS Functions and CALL Routines: Reference.
See Also

Examples
- “Example: Create a Variable Numbered Range List”
- “Example: Create a Variable Name Range List”
- “Example: Use the OF Operator with a Variable List”
- “Example: Use the OF Operator to Create Multiple Variable Lists”

Statements
- ARRAY Statement
- INPUT Statement, List
- KEEP Statement
- PUT Statement
- VAR Statement

Functions
- SUM Function

Missing Variable Values

Definition of Missing Values
A missing value is a value that indicates that no data value is stored for the variable in the current observation. There are three types of missing values:
- numeric
- character
- special numeric

How to Represent Missing Values in Raw Data
The following table shows how to represent each type of missing value in raw data so that SAS reads and stores the value appropriately.
Table 5.11 Representing Missing Values

<table>
<thead>
<tr>
<th>Missing Values</th>
<th>Representation in Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>. (a single decimal point)</td>
</tr>
<tr>
<td>Character</td>
<td>' ' (a blank enclosed in quotation marks)</td>
</tr>
<tr>
<td>Special</td>
<td>. letter (a decimal point followed by a letter, for example, .B)</td>
</tr>
<tr>
<td>Special</td>
<td>._ (a decimal point followed by an underscore)</td>
</tr>
</tbody>
</table>

Special Missing Values

Definition

A special missing value is a type of numeric missing value that enables you to represent different categories of missing data by using the letters A–Z or an underscore.

Tips for Special Missing Values

- SAS accepts either uppercase or lowercase letters. Values are displayed and printed as uppercase.
- If you do not begin a special numeric missing value with a period, SAS identifies it as a variable name. Therefore, to use a special numeric missing value in a SAS expression or assignment statement, you must begin the value with a period, followed by the letter or underscore. For example:
  ```sas
  x=.d;
  ```
- When SAS prints a special missing value, it prints only the letter or underscore.
- When data values contain characters in numeric fields that you want SAS to interpret as special missing values, use the MISSING statement to specify those characters.

Order of Missing Values

Numeric Variables

Within SAS, a missing value for a numeric variable is smaller than all numbers. If you sort your data set by a numeric variable, observations with missing values for that variable appear first in the sorted data set. For numeric variables, you can
compare special missing values with numbers and with each other. The following
table shows the sorting order of numeric values.

Table 5.12 Numeric Value Sort Order

<table>
<thead>
<tr>
<th>Sort Order</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallest</td>
<td>.</td>
<td>underscore</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>period</td>
</tr>
<tr>
<td></td>
<td>.A-.Z</td>
<td>special missing values A (smallest) through Z (largest)</td>
</tr>
<tr>
<td></td>
<td>-n</td>
<td>negative numbers</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>zero</td>
</tr>
<tr>
<td>largest</td>
<td>+n</td>
<td>positive numbers</td>
</tr>
</tbody>
</table>

For example, the numeric missing value (.) is sorted before the special numeric
missing value .A, and both are sorted before the special missing value .Z. SAS does
not distinguish between lowercase and uppercase letters when sorting special
numeric missing values.

Note: The numeric missing value sort order is the same regardless of whether your
system uses the ASCII or EBCDIC collating sequence.

Character Variables

Missing values of character variables are smaller than any printable character value.
Therefore, when you sort a data set by a character variable, observations with
missing (blank) values of the BY variable always appear before observations in
which values of the BY variable contain only printable characters. However, some
unprintable characters (for example, machine carriage-control characters and real or
binary numeric data that have been read in error as character data) have values
less than the blank. Therefore, when your data includes unprintable characters,
missing values might not appear first in a sorted data set.

When Variable Values Are Automatically Set to Missing by SAS

When Reading Raw Data

At the beginning of each iteration of the DATA step, SAS sets the value of each
variable that you create in the DATA step to missing, with the following exceptions:

- variables named in a RETAIN statement
- variables created in a SUM statement
- data elements in a _TEMPORARY_ array
- variables created with options in the FILE or INFILE statements
- variables created by the FGET function
- data elements that are initialized in an ARRAY statement
- automatic variables

When Reading a SAS Data Set

When variables are read with a SET, MERGE, or UPDATE statement, SAS sets the values to missing only before the first iteration of the DATA step. (If you use a BY statement, the variable values are also set to missing when the BY group changes.) The variables retain their values until new values become available (for example, through an assignment statement or through the next execution of the SET, MERGE, or UPDATE statement). Variables created with options in the SET, MERGE, and UPDATE statements also retain their values from one iteration to the next.

When all rows in a data set in a match-merge operation (with a BY statement) are processed, the variables in the output data set retain their values as described earlier. That is, as long as there is no change in the BY value in effect when all of the rows in the data set have been processed, the variables in the output data set retain their values from the final observation. FIRST.variable and LAST.variable, the automatic variables that are generated by the BY statement, both retain their values. Their initial value is 1.

When the BY value changes, the variables are set to missing and remain missing because the data set contains no additional observations to provide replacement values. When all of the rows in a data set in a one-to-one merge operation (without a BY statement) have been processed, the variables in the output data set are set to missing and remain missing.

When Missing Values Are Generated by SAS

Propagate Missing Values in Calculations

If you use a missing value in an arithmetic calculation, SAS sets the result of that calculation to missing. Then, if you use that result in another calculation, the next result is also missing. This action is called propagation of missing values. SAS prints notes in the log to notify you which arithmetic expressions have missing values and when they were created. However, processing continues.

Prevent Propagation of Missing Values

You can omit missing values from computations by using statistic functions, such as SUM statement and SUM function, if you do not want missing values to propagate in your arithmetic expressions.
Invalid Operations

SAS prints a note in the log and assigns a missing value to the result if you try to perform an invalid operation, such as the following:

- dividing by zero
- taking the logarithm of zero
- using an expression to produce a number too large to be represented as a floating-point number (known as overflow)

Invalid Character-to-Numeric Conversions

SAS automatically converts character values to numeric values if a character variable is used in an arithmetic expression. If a character value contains nonnumeric information and SAS tries to convert it to a numeric value, a note is printed in the log, the result of the conversion is set to missing, and the _ERROR_ automatic variable is set to 1.

See Also

**Examples**

- “Example: Automatically Replacing Missing Values”
- “Example: Creating Special Missing Values”
- “Example: Preventing Propagation of Missing Values”

**Functions**

- MISSING Function

**Statements**

- MISSING Statement

Numeric Precision

Overview

In any number system, whether it is binary or decimal, there are limitations to how precise numbers can be represented. As a result, approximations have to be made.

For example, in the decimal number system, the fraction $1/3$ cannot be perfectly represented as a finite decimal value because it contains infinitely repeating digits.
On computers, because of finite precision, this number cannot be represented with perfect precision. Numerical precision is the accuracy with which numbers are approximated or represented.

In computing, software applications are particularly susceptible to numerical precision errors due to finite precision and machine hardware limitations. Computers are finite machines with finite storage capacity, so they cannot represent an infinite set of numbers with perfect precision.

The problem is further compounded by the fact that computers use a different number system than people do. Decimal infinite-precision arithmetic is the norm for human calculations but computers use finite binary representations of values and finite-precision arithmetic. This representation has been proven adequate for many calculations. Yet, depending on the problem, you might need an extended precision that is wider than what the hardware offers. In that case, representation and arithmetic are done mostly in software and are relatively much slower than hardware arithmetic.

Furthermore, although computers do allow the use of decimal numbers and decimal arithmetic via human-centric software interfaces, all numbers and data are eventually converted to binary format to be stored and processed by the computer internally. It is in the conversion between these 2 number systems – decimal to binary – that precision is affected and rounding errors are introduced.

Truncation in Binary Numbers

Just like there are decimal values with infinitely repeating representations, there are also binary values that have infinitely repeating representations. However, the numbers that are imprecise in decimal are not always the same ones that are imprecise in binary.

For example, the decimal value \( \frac{1}{10} \) has a finite decimal representation (0.1), but in binary it has an infinitely repeating representation. In binary, the value converts to \( 0.00011001100110011 \ldots \)

where the pattern 0011 is repeated indefinitely. As a result, the value is rounded when stored on a computer.

Performing calculations and comparisons on imprecise numbers in SAS can lead to unexpected results. Even the simplest calculations can lead to a wrong conclusion. Hardware cannot always match what might seem obvious and expected in the decimal system.

For example, in decimal arithmetic, the expression \( 3 \times 0.1 \) is expected to be equal to 0.3, so the difference between \( 3 \times 0.1 \) and \( 0.3 \), must be 0. Because the decimal values 0.1 and 0.3 do not have exact binary representations, this equality does not hold true in binary arithmetic. If you compute the difference between the two values in a SAS program, the result is not 0.

There are many decimal fractions whose binary equivalents are infinitely repeating binary numbers, so be careful when interpreting results from general rational numbers in decimal. There are some rational numbers that do not present problems in either number system. For example, \( \frac{1}{2} \) can be finitely represented in both the decimal and binary systems.

To understand better why a simple calculation such as this one can go wrong, or how a number can be out of range, it is important to understand in more detail how SAS stores binary numbers.
How SAS Stores Numeric Values

Maximum Integer Size

SAS stores all numeric values in 8 bytes of storage unless you specify differently. This does not mean that a value is limited to 8 digits, but rather that 8 bytes are allocated for storing the value. In the previous section, you learned how storing non-integer values (fractions) can lead to problems with precision. But you can also encounter problems of magnitude, or range, when working with integers (whole numbers).

On any computer, there are limits to how large the absolute value of an integer can be. In SAS, this maximum integer value depends on two factors:

- the number of bytes that you explicitly specify for storing the variable (using the LENGTH statement)
- the operating environment on which SAS is running

If you have not explicitly specified the number of storage bytes, then SAS uses the default length of 8 bytes, and the maximum integer then depends solely on what operating system you are using.

The following table lists the largest integer that can be reliably stored by a SAS variable in the mainframe, UNIX, and Windows operating environments.

<table>
<thead>
<tr>
<th>When Variable Length Equals ...</th>
<th>Largest Integer z/OS</th>
<th>Largest Integer Windows or UNIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>256</td>
<td>not applicable</td>
</tr>
<tr>
<td>3</td>
<td>65,536</td>
<td>8,192</td>
</tr>
<tr>
<td>4</td>
<td>16,777,216</td>
<td>2,097,152</td>
</tr>
<tr>
<td>5</td>
<td>4,294,967,296</td>
<td>536,870,912</td>
</tr>
<tr>
<td>6</td>
<td>1,099,511,627,776</td>
<td>137,438,953,472</td>
</tr>
<tr>
<td>7</td>
<td>281,474,946,710,656</td>
<td>35,184,372,088,832</td>
</tr>
<tr>
<td>8 (default)</td>
<td>72,057,594,037,927,936</td>
<td>9,007,199,254,740,992</td>
</tr>
</tbody>
</table>

When viewing this table, consider the following points:

- The minimum length for a SAS variable on Windows and UNIX operating systems is 3 bytes, and the maximum length is 8 bytes. On IBM mainframes, the minimum length for a SAS variable is 2 bytes, and the maximum length is 8 bytes.
As the length of the variable increases, so does the size of the integer that can be reliably represented.

For any given variable length, the maximum integer varies by host. This is because mainframes have different specifications for storing floating-point numbers than UNIX and PC machines do.

Always store real numbers in the full 8 bytes of storage. If you want to save disk space by using the LENGTH statement to reduce the length of your variables, you can do so but only for variables whose values are integers. When adjusting the length of variables, make sure that the values are less than or equal to the largest integer allowed for that specified length.

For example, in the UNIX operating environment, if you know that the values of your numeric variables are always integers between -8192 and 8192, then you can safely specify a length of 3 to store the number:

```
data myData;
  length num 3;
  num=8000;
run;
```

**CAUTION**

*Use the full 8 bytes to store variables that contain real numbers.*

---

**Floating-Point Representation**

SAS stores numeric values in 8 bytes of data. The way that the numbers are stored and the space available to store them also affects numerical accuracy. Although there are various ways to store binary numbers internally, SAS uses *floating-point representation* to store numeric values. Floating-point representation supports a wide range of values (very large or very small numbers) with an adequate amount of numerical accuracy.

You might already be familiar with floating-point representation because it is similar to *scientific notation*. In both scientific notation and floating-point representation, each number is represented as a *mantissa*, a *base*, and an *exponent*.

\[
987 = \text{mantissa} \times 10^3
\]

- the *mantissa* is the number that is being multiplied by the base. In the example, the mantissa is .987.
- the *base* is the number that is being raised to a power. In the example, the base is 10.
- the *exponent* is the power to which the base is raised. In the example, the exponent is 3.

One major difference between scientific notation and floating-point representation is that in scientific notation, the base is 10. In floating-point representation, on most operating systems, the base is either 2 or 16 depending on the system.

The following figure shows the decimal value 987 written in the IEEE 754 binary floating-point format. Because it is a small value, no rounding is needed.
To store binary floating-point numbers, computers use standard formats called *interchange formats*, or *byte layouts*. The byte layout is a standard way of grouping and ordering bit strings, from left to right, so that the parts of the floating-point number are represented in a standardized way. Each part of the floating-point value (sign, exponent, mantissa) is allotted a specific number of bits in the string and a specific position in the string. This allows for the exchange of floating-point data in an efficient and compact form.

**Figure 5.1** on page 102 shows the byte layout for a double-precision binary floating-point number. This layout uses the first bit to encode the sign of the number, the next 11 bits to encode the exponent, and the final 52 bits to encode the mantissa. If the sign bit is 1, then the number is negative and if the sign bit is 0, the number is positive.

**Figure 5.1  Byte Layout for a Double-Precision Binary Floating-Point Number**

Different host computers can have different formats and specifications for floating-point representation. All platforms on which SAS runs use 8-byte floating-point representation.

**Precision versus Magnitude**

The largest integer value that can be represented exactly (without rounding) depends on the base and the number of bits that are allotted to the exponent. The precision is determined by the number of bits that are allotted for the mantissa. Whether an operating system truncates or rounds digits affects errors in representation.

SAS stores truncated floating-point numbers using the LENGTH statement, which reduces the number of mantissa bits. The following table shows some differences between floating-point formats for the IBM mainframe and the IEEE standard. The IEEE standard is used by the Windows and UNIX operating systems.
Table 5.14  IBM and IEEE Standard for Floating-Point Formats

<table>
<thead>
<tr>
<th>Specifications</th>
<th>IBM Mainframe</th>
<th>IEEE Standard (Windows/UNIX)</th>
<th>Affects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>16</td>
<td>2</td>
<td>magnitude</td>
</tr>
<tr>
<td>Exponent Bits</td>
<td>7</td>
<td>11</td>
<td>magnitude</td>
</tr>
<tr>
<td>Mantissa bits</td>
<td>56</td>
<td>52</td>
<td>precision</td>
</tr>
<tr>
<td>Round or Truncate</td>
<td>Truncate</td>
<td>Round</td>
<td>precision</td>
</tr>
<tr>
<td>Bias for Exponent</td>
<td>64</td>
<td>1023</td>
<td></td>
</tr>
</tbody>
</table>

The following bullet points describe the table above in more detail:

- **Base 16** – uses digits 0-9 and letters A-F (to represent the values 10-15).
  
  For example, to convert the decimal value 3000 to hexadecimal, you use the base 16 number system:
  
  \[
  
  \begin{array}{c|c|c|c|c|c}
  \text{Base 16} & 16^7 & \ldots & 16^4 & 16^3 & 16^2 & 16^1 & 16^0 \\
  \hline
  268,435,456 & \ldots & 65,536 & 4096 & 256 & 16 & 1 \\
  \end{array}
  \]

  \[
  
  3000 = (\text{B} \times 16^2) + (\text{B} \times 16^3) + (\text{B} \times 16^0)
  
  = (11 \times 256) + (11 \times 256) + (8 \times 1)
  
  So, the value 3000 is represented in hexadecimal as BB8.
  
- **Base 2** – uses digits 0 and 1.
  
  For example, to convert the decimal value 184 to binary, you use the base 2 number system:
  
  \[
  
  \begin{array}{c|c|c|c|c|c}
  \text{Base 2} & 2^7 & \ldots & 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \\
  \hline
  128 & \ldots & 16 & 8 & 4 & 2 & 1 \\
  \end{array}
  \]

  \[
  
  184 = (1 \times 2^7) + (0 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (0 \times 2^0)
  
  = 128 + 0 + 32 + 16 + 8 + 0 + 0 + 0
  
  So, the value 184 is represented in binary as 10111000.
  
- **exponent bits** – the number of bits reserved for storing the exponent, which determines the magnitude of the number that you can store. The number of
exponent bits varies between operating systems. IEEE systems yield numbers of greater magnitude because they use more bits for the exponent.

- **mantissa bits** – the number of bits reserved for storing the mantissa, which determines the precision of the number. Because there are more bits reserved for the mantissa on mainframes, you can expect greater precision on a mainframe compared to a PC.

- **round or truncate** – the chosen conversion method used for handling two or more digits. Because there is room for only two hexadecimal characters in the mantissa, a convention must be adopted on how to handle more than two digits. One convention is to truncate the value at the length that can be stored. This convention is used by IBM Mainframe systems.

An alternative is to round the value based on the digits that cannot be stored, which is done on IEEE systems. There is no right or wrong way to handle this dilemma since neither convention results in an exact representation of the value.

In SAS, the LENGTH statement works by truncating the number of mantissa bits. For more information about the effects of truncated lengths, see “Using the TRUNC Function When Comparing Values”.

- **bias** – an offset used to enable 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 uses a bias of 64, a characteristic with the value 66 represents an exponent of +2, whereas a characteristic of 61 represents an exponent of –3.

---

**Floating-Point Representation Using the IEEE Standard**

The IEEE standard for floating-point arithmetic is a technical standard for floating-point computation created by the Institute of Electrical and Electronic Engineers (IEEE). The standard defines how computers store numbers in floating-point representation. The IEEE standard for floating-point numbers is used by many operating systems, including Windows and UNIX.

Although the IEEE platforms use the same set of specifications, you might occasionally see varying results between the platforms due to compiler differences, and math library differences. Also, because the IEEE standard allows for some variations in how the standard is implemented, there might be differences in how different platforms perform calculations even though they are following the same standard. Hosts might yield different results because the underlying instructions that each operating system uses to perform calculations are slightly different.

There is no standard method for performing computations. All operating systems attempt to compute numbers as accurately as possible. It is not uncommon to get slightly different results between operating systems whose floating-point representation components differ. For example, there are differences between the z/OS and Windows operating systems and between the z/OS and UNIX operating systems.

The IEEE standard for double-precision, floating-point numbers specifies an 11-bit exponent with a base of 2 and a bias of 1023, which means that it has much greater magnitude than the IBM mainframe representation, but sometimes at the expense of 3 bits less in the mantissa. The value of 1 represented by the IEEE standard is as follows:

```
3F  F0 00 00 00 00 00 00
```
On Windows platforms, the processor performs computations in extended real precision. This means that instead of the 64 bits that are used to store numeric values in the basic format (52 bits for the mantissa and 11 bits for the exponent), there are 16 additional bits: 12 additional bits for the mantissa and 4 additional bits for the exponent. Numeric values are not stored in 80 bits (10 bytes) since the maximum width for a numeric variable in SAS is 8 bytes. This simply means that the processor uses 80 bits to represent a numeric value before it is passed back to its 64–bit memory slot. Intermediate calculations might be done in 80 bits, which affects a part of the final answer.

On Windows this allows storage of numbers larger than the basic IEEE floating-point format used by operating systems such as UNIX. This is one reason why you might see slightly different values from operating systems that use the same IEEE standard. *Extended precision* formats provide greater precision and more exponent range than the basic floating-point formats.

### Floating-Point Representation on Windows

#### Storage Format

The byte layout for a 64-bit, double-precision number on Windows is as follows:

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
<th>Byte 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S E E E E E E</td>
<td>E E E E</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Byte 5</th>
<th>Byte 6</th>
<th>Byte 7</th>
<th>Byte 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
</tbody>
</table>

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 remaining M characters in bytes 2 through 8 represent the bits of the mantissa. There is an implied radix point before the left-most 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 format, but the exponent is used to determine how many times the base should be multiplied by the mantissa.

### Accuracy on x64 Windows Processors

Consider this example:

```plaintext
data _null_;```
Although these values appear to be alike, the internal representations differ slightly, because the IEEE floating-point representation can represent only 15 digits. Here is the floating-point representation of both variables using the HEX16. format.

```
x=3FE0000000000000
y=3FDFFFFFFF
```

When the number of significant digits is reduced to 15 or less, the floating-point representation is the same and the values are equal.

```
data _null_;  
x=.5000000000000000;  
y=.5000000000000000;  
if x=y then put 'equal';  
else put 'not equal';  
put x=hex16.;  
y=hex16.;  
run;
```

Log Output

```
not equal
```

This issue pertains to floating-point representation on the x64 processors. The routine used to compute the result is slightly different on Windows than on any other host (Linux, UNIX, AIX, and so on).

---

### Floating-Point Representation on IBM Mainframes

#### Storage Format

SAS for z/OS uses the traditional IBM mainframe floating-point representation as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte 1</td>
<td>Byte 2</td>
<td>Byte 3</td>
<td>Byte 4</td>
</tr>
<tr>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
<td>M M M M M M M M</td>
</tr>
<tr>
<td>Byte 5</td>
<td>Byte 6</td>
<td>Byte 7</td>
<td>Byte 8</td>
</tr>
</tbody>
</table>
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 enable 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 uses a bias of 64, a characteristic with the value of 66 represents an exponent of +2, whereas 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 left-most 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.

### Troubleshooting Errors in Precision

#### Computational Considerations

Regardless of how much precision is available, there are still some numbers that cannot be represented exactly. Most rational numbers (for example, .1) cannot be represented exactly in base 2 or base 16. This is why it is often difficult to store fractions in floating-point representation.

Consider the IBM mainframe representation of 1:

```
1: 40 19 99 99 99 99 99 99
```

Notice that here is an infinitely repeating 9 digit similar to the trailing 3 digit in the attempted decimal representation of one-third (.33333 ...). This lack of precision can be compounded when arithmetic operations are performed on these values repeatedly.

For example, when you add .33333 to .99999, the theoretical answer is 1.33333, but in practice, this answer is not possible. The sums become more imprecise as the values continue to be calculated.

For example, consider the following DATA step:

```plaintext
data _null_
  do i=-1 to 1 by .1;
    put i=;
    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 errors that the exact value of 0 is never encountered. The calculated result is close to 0, but never exactly equal to 0. Therefore, when numbers cannot be represented exactly in floating point,
performing mathematical operations with other non-exact values can compound the imprecision.

**Using the LENGTH Statement When Comparing Values**

You can use the LENGTH statement to control the number of bytes that are used to store variable values. However, you must use it carefully to avoid errors and 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 is truncated to `41 10`. In this case, you still have the full range of magnitude because the exponent remains intact, but there are 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 zeros must be used.

For example, consider the number `1234567890`, which is `.1234567890` to the 10th power of 10 in base 10 floating-point notation. 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).

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 enables 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>
<thead>
<tr>
<th>Value</th>
<th>Sign or Exp</th>
<th>Mantissa 1</th>
<th>Mantissa 2-7</th>
<th>Considerations</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>43</td>
<td>10</td>
<td>000000000000</td>
<td>trailing zeros; multiple of 16</td>
</tr>
<tr>
<td>257</td>
<td>43</td>
<td>10</td>
<td>100000000000</td>
<td>extra byte needed</td>
</tr>
<tr>
<td>258</td>
<td>43</td>
<td>10</td>
<td>200000000000</td>
<td></td>
</tr>
<tr>
<td>259</td>
<td>43</td>
<td>10</td>
<td>300000000000</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>271</td>
<td>43</td>
<td>10</td>
<td>F00000000000</td>
<td>trailing zeros; multiple of 16</td>
</tr>
<tr>
<td>272</td>
<td>43</td>
<td>11</td>
<td>000000000000</td>
<td></td>
</tr>
</tbody>
</table>
Using the TRUNC Function When Comparing Values

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

\[ x = \frac{1}{3} \]

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

\[ \text{if } x = \frac{1}{3} \text{ then ...;} \]

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

\[ \text{if } x = \text{trunc}(\frac{1}{3}, 3) \text{ then ...;} \]

See “TRUNC Function” in SAS Functions and CALL Routines: Reference for more information about this function.

Double-Precision versus Single-Precision Floating-Point Numbers

You might have data that has been 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 RBw.d informat to read in the data. However, there are exceptions. The \( \text{RBw.d} \) 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 \( \text{RB8.} \) informat corresponds to a full 8-byte floating point. The \( \text{RB4.} \) 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 a float point 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 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 PL/I.

On the IBM mainframes, 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

Problems of precision and magnitude can occur when you transfer data containing very large or very small numeric values that are represented in floating-point notation. “Floating-Point Representation Using the IEEE Standard”, “Floating-Point Representation on Windows”, and “Floating-Point Representation on IBM Mainframes” 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 computer to another.
Consider transporting data between an IBM mainframe and a PC, for example. The IBM mainframe has a range limit of approximately \(5.4 \times 10^{-78}\) to \(7.2 \times 10^{76}\) (and their negative equivalents and 0) for its floating-point numbers.

Other computers, such as the PC, have wider limits (the PC has an upper limit of approximately \(1 \times 10^{308}\)). Therefore, if you are transferring numbers in the magnitude of \(1 \times 10^{100}\) 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 \(1 \times 10^{100}\) on a PC is converted to a value that is approximately \(.72 \times 10^{76}\) on an IBM mainframe.

**CAUTION**

Transfer of data between computers can affect numerical 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. This means that 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.

An alternative solution, and probably the safest way to avoid numerical precision problems when transferring data between operating systems, is to convert the numbers in your data to integers.

For more information about moving data between operating systems, see *Moving and Accessing SAS Files*.

---

### See Also

**Examples**

- “Example: Compare Imprecise Values in SAS”
- “Example: Convert a Decimal Value to a Floating Point Representation”
- “Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation”
- “Example: Round Values to Avoid Computational Errors”
- “Example: Use the LENGTH Statement to Compare Values”
- “Example: Compare Values That Have Imprecise Representations”
- “Example: Confirm Precision Errors Using Formats”
- “Example: Determine How Many Bytes Are Needed to Store a Number Accurately”

**Functions**

- “ROUND Function” in *SAS Functions and CALL Routines: Reference*
- “TRUNC Function” in *SAS Functions and CALL Routines: Reference*

**Statements**

- “LENGTH Statement” in *SAS DATA Step Statements: Reference*
Examples: Create and Modify SAS Variables

Example: Create a SAS Variable Using an Assignment Statement

Example Code

This example shows how to create a numeric and character variable using an assignment statement. A numeric variable evaluates the value on the right side of the expression as an integer and the variable type is implicitly set to a numeric data type. To create the character variable, you can enclose the value in quotation marks specifies that the variable type is character. The length of the variable is set to the length of the character value. The length for the variable Patient is set to 12.

```sas
data newvar;
  Insurance=100;
  Patient="John Whitman"
run;
```

Key Ideas

- 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.
- The variable gets the same type and length as the expression on the right side of the assignment statement.

See Also

- Assignment Statement
- Ways to Create Variables
Example: Create a Variable Using the INPUT Statement

Example Code

This example demonstrates using simple list input to create a SAS data set, named `gems`, and define four variables based on the data provided.

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

The INPUT statement reads in four variables: Name, Color, Carats, and Owner. Name, Color, and Owner were character variables as indicated by the $ and Carats is a numeric variable.

Key Ideas

- When reading raw data into SAS, you can use the INPUT statement to define your variables based on positions within the raw data.
- You can use one of the following methods with the INPUT statement to provide information about the raw data organization:
  - column input
  - list input (simple or modified)
  - formatted input
  - named input

See Also

- INPUT Statement, List
- Assignment Statement

Example: Modify a SAS Variable Using the LENGTH Statement

Example Code

This example shows how to modify a SAS variable using the LENGTH statement.
data sales;
  length Salesperson $25;
  Salesperson='Jonathon Mark Walker';
run;

Key Ideas

- For character variables, you must use the longest possible value in the first
  statement that uses the variable. The reason is that you cannot change the length
  with a subsequent LENGTH statement within the same DATA step. The maximum
  length of any character variable in SAS 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.

See Also

- LENGTH Statement
- Assignment Statement

Example: Modify a SAS Variable Using the FORMAT and
INFORMAT Statement

Example Code

This example demonstrates how to read in raw data from an external data file and
create new variables in an output data set using both simple list INPUT and
formatted input. The INPUT statement creates the variable Name as a character
variable with a default length of 8 using simple list input (no length is specified). The
INPUT statement also creates a numeric variable Carats that also has a length of 8.
Because the last variable in the INPUT statement, Color, specifies an INFORMAT in
the INPUT statement, it is being read in using formatted input, SAS defines the
variable as a character variable with a length of 10.

data gems;
  input Name $ Carats comma3.1 Color $char10.;
  informat Name $char10.;
  format Carats comma3.1;
datalines;
emerald    15 green
aquamarine 20 blue
;
proc print data=gems; run;
proc contents data=gems; run;
data gems;
  input Name $char10. Carats comma3.1 Color $char.;
datalines;
emerald    15 green
aquamarine 20 blue
;
proc print data=gems; run;
proc contents data=gems; run;

Output 5.4  PROC PRINT Result

<table>
<thead>
<tr>
<th>Obs</th>
<th>Sale_Price</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$49.99</td>
<td>January 10, 2019</td>
</tr>
<tr>
<td>2</td>
<td>$29.99</td>
<td>January 14, 2019</td>
</tr>
<tr>
<td>3</td>
<td>$32.59</td>
<td>January 16, 2019</td>
</tr>
</tbody>
</table>

Key Ideas

- If a variable does not already exist and you create it for the first time in a formatted INPUT statement, then SAS defines the variable and its attributes based on the category of the informat specified in the INPUT statement:
  - Associating a numeric informat with a variable when it is created for the first time in a DATA step using formatted input causes the variable to be created as a numeric type, with a default length of 8.
  - Associating a character informat with a variable when it is created for the first time in a DATA step using formatted input causes the variable to be created as a character type. The length matches the width specified in the informat in the INPUT statement. If you do not specify a length with the informat or anywhere else in the DATA step, then SAS assigns the default length of 8 bytes.
- You can modify a variable and specify its format or informat with a FORMAT or INFORMAT statement.

See Also

- FORMAT Statement
- INFORMAT Statement
- Ways to Create Variables
Example: Modify a SAS Variable Using the ATTRIB Statement

Example Code

The following DATA step creates a variable named Flavor in a data set named lollipops.

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

Key Ideas

- 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.
- You cannot create a new variable by using a LABEL statement or the ATTRIB statement's LABEL= attribute by itself. Labels can be applied only to existing variables.

See Also

- ATTRIB Statement
- Ways to Create Variables

Example: View Variable Attributes

Example Code

This example shows you how to use the CONTENTS procedure to view Sashelp.Cars variable names, types, and attributes.

The CONTENTS procedure generates summary information about the contents of a data set, including:
The number of observations in a data set
The number of variables in a data set
The maximum number of observations in each page and file size
The variables' names, types, and attributes (including formats, informats, and labels)
If your data is sorted by any variable, then the sort information is also displayed.

```
proc contents data=sashelp.cars;
run;
```

The output of the CONTENTS procedure is engine dependent.

The alphabetic list of variables and attributes table generated by the CONTENTS procedure shows #, Variable, Type, Len, Format and Label.

#  
The original order of the variable in the columns of the data set. PROC CONTENTS prints the variables in alphabetical order with respect with the name, instead of the order in which they appear in the data set.

Variable  
Identifies the variable name. This might be different from what you see as the name displayed in the output.

Type  
Whether the variable is numeric (Num) or character (Char).

Len  
Short for "Length". Represents the width of the variable.

Format  
The assigned format that will be used when the variables are printed in the Results window.

Informat  
The original format of the variable when it was read into SAS.

Label  
The assigned variable label that will be used when the name of the variable is printed in the Output window. If your variables do not have labels, this column is identical to the Variable column.

Key Ideas

- PROC CONTENTS describes the structure and displays the variable attributes of the data set, rather than the data values.
- This procedure is especially useful if you have imported your data from a file and want to check that your variables have been read correctly, and have the appropriate variable type and format.

See Also

- CONTENTS Procedure
- Variable Attributes
Example: Change Variable Attributes

Example Code

This example shows how to change variable attributes using the ATTRIB statement. The ATTRIB statement associates a format, informat, label, and length with one or more variables.

data flightmiles;
  attrib
  Atlanta label='ATL' length=8 format=comma8.
  Chicago label='ORD' length=8 format=comma8.
  Denver label='DEN' length=8 format=comma8.
  Houston label='IAH' length=8 format=comma8.
  LosAngeles label='LAX' length=8 format=comma8.
  Miami label='MIA' length=8 format=comma8.
  NewYork label='JFK' length=8 format=comma8.
  SanFrancisco label='SFO' length=8 format=comma8.
  Seattle label='SEA' length=8 format=comma8.
  WashingtonDC label='DCA' length=8; format=comma8.
;
  set sashelp.mileages;
run;
title1 'Flying Miles Between Ten US Cities';
proc print data=flightmiles label;
run;

Output 5.5  Using ATTRIB Statement — Label and Format

If you add a PROC CONTENTS step, you can see how the ATTRIB statement changed your variable attributes. Each variable is formatted (boxed in orange), length (boxed in brown), and a label (boxed in red).
For character variables, you must use the longest possible value in the first statement that uses the variable. The reason is that you cannot change the length with a subsequent LENGTH statement within the same DATA step. The maximum length of any character variable in SAS is 32,767 bytes.

Using the ATTRIB statement in the DATA step permanently associates attributes with variables by changing the descriptor information of the SAS data set that contains the variables.

You can use either an ATTRIB statement or an individual attribute statement such as FORMAT, INFORMAT, LABEL, and LENGTH to change an attribute that is associated with a variable.
Examples: Control Output of Variables

Example: Select Specific Variables for Output

Example Code

This example uses the DROP statement and the DROP= data set option to control the output of variables to the new SAS data sets. The DROP statement drops the MSRP variable from the new data set, `newcars`, while the DROP= data set option drops seven variables from the original data set, `sashelp.cars`, and does not bring those variables into the new data set when reading from `sashelp.cars`.

There are 428 observations and 15 variables in `sashelp.cars`. The new data set, `newcars`, contains 17 observations and 7 variables.

```sas
data newcars;
  set sashelp.cars(drop=origin enginesize cylinders horsepower weight wheelbase length);
  if MSRP>75000 then output;
  drop msrp;
run;
proc print data=newcars;
run;
```
Output 5.7  DROP= and DROP Statement Output

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>Model</th>
<th>Type</th>
<th>DriveTrain</th>
<th>Invoice</th>
<th>MPG_City</th>
<th>MPG_Highway</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acura</td>
<td>NSX coupe 2dr manual S</td>
<td>Sports</td>
<td>Rear</td>
<td>$79,978</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>Audi</td>
<td>RS 6 4dr</td>
<td>Sports</td>
<td>Front</td>
<td>$76,417</td>
<td>15</td>
<td>22</td>
</tr>
<tr>
<td>3</td>
<td>Cadillac</td>
<td>XLR convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$70,546</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Dodge</td>
<td>Viper SRT-10 convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$74,451</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>Jaguar</td>
<td>XKR convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$74,676</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Jaguar</td>
<td>XKR convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$79,225</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Mercedes-Benz</td>
<td>G500</td>
<td>SUV</td>
<td>All</td>
<td>$71,540</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>Mercedes-Benz</td>
<td>CL500 2dr</td>
<td>Sedan</td>
<td>Rear</td>
<td>$88,324</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>9</td>
<td>Mercedes-Benz</td>
<td>CL600 2dr</td>
<td>Sedan</td>
<td>Rear</td>
<td>$119,600</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>10</td>
<td>Mercedes-Benz</td>
<td>S500 4dr</td>
<td>Sedan</td>
<td>All</td>
<td>$80,939</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>11</td>
<td>Mercedes-Benz</td>
<td>SL500 convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$84,325</td>
<td>15</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>Mercedes-Benz</td>
<td>SL55 AMG 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$113,383</td>
<td>14</td>
<td>21</td>
</tr>
<tr>
<td>13</td>
<td>Mercedes-Benz</td>
<td>SL600 convertible 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$117,654</td>
<td>13</td>
<td>19</td>
</tr>
<tr>
<td>14</td>
<td>Porsche</td>
<td>911 Carrera convertible 2dr (coupe)</td>
<td>Sports</td>
<td>Rear</td>
<td>$69,229</td>
<td>18</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>Porsche</td>
<td>911 Carrera 4S coupe 2dr (convert)</td>
<td>Sports</td>
<td>All</td>
<td>$72,206</td>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td>Porsche</td>
<td>911 Targa coupe 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$67,128</td>
<td>18</td>
<td>26</td>
</tr>
<tr>
<td>17</td>
<td>Porsche</td>
<td>911 GT2 2dr</td>
<td>Sports</td>
<td>Rear</td>
<td>$173,560</td>
<td>17</td>
<td>24</td>
</tr>
</tbody>
</table>

Key Ideas

- The DROP statement or the DROP= data set options control which variables are processed during the DATA step.
- 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.

See Also

- DROP Statement
- DROP= Data Set Option
- Using Statements or Data Set Options
Example: Select Specific Variables for Processing

Example Code

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

```sas
data fails (drop=Process);
  length Day 8;
  set sashelp.failure(rename=(Day=Weekday));
    Day=input(Weekday,8.);
  run;
proc print data=fails;
run;
```

Key Ideas

- The DROP= data set options control which variables are processed or output during the DATA step.
- 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.

See Also

- RENAME= Data Set Option
- DROP= Data Set Option
- Using Statements or Data Set Options
Examples: Reorder and Align Variables

Example: Reorder Variables Using the ATTRIB Statement

Example Code

In the following example, the data set sashelp.class contains variables Name, Sex, Age, Height, and Weight (in that order). The ATTRIB statement is specified before the SET statement so that the variable Sex is moved to the first position in the output data set.

```sas
data class3;
  attrib Sex length=$8;
  set sashelp.class;
  if Sex='M' then Sex='Male';
  else Sex='Female';
run;
proc print data=class3;
run;
```

The output displays the Sex variable listed first because the ATTRIB statement preceded the SET statement.
### Key Ideas

- You can control the order in which variables are displayed in SAS output by using the ATTRIB statement.
- Use the ATTRIB statement prior to the SET, MERGE, or UPDATE statement in order for you to reorder the variables.
- Variables not listed in the ATTRIB statement retain their original position.

### See Also

- ARRAY Statement
- ATTRIB Statement
Example: Reorder Variables Using the LENGTH Statement

Example Code

In the following example, the data set `sashelp.class` contains variables `Name`, `Sex`, `Age`, `Height`, and `Weight` (in that order). The LENGTH statement is specified before the SET statement so that the variable `Height` is moved to the first position in the output data set.

```sas
data class1;
    length Height 3;
    set sashelp.class;
run;
proc print data=class1;
run;
```

The output displays the `Height` variable listed first because the LENGTH statement preceded the SET statement.
You can use the CONTENTS procedure on the Class1 data set to view the length of each variable.

Key Ideas

- You can control the order in which variables are displayed in SAS output by using the LENGTH statement.
- Use the LENGTH statement prior to the SET, MERGE, or UPDATE statement in order for you to reorder the variables.
- Variables not listed in the LENGTH statement retain their original position.

See Also

- ARRAY Statement
- ATTRIB Statement
- FORMAT Statement
Example: Reorder Variables Using the RETAIN Statement

Example Code

In the following example, the RETAIN statement causes the variables Weight and Age to be listed first in the output data set. The data set sashelp.class contains variables Name, Sex, Age, Height, and Weight (in that order).

```sas
data class2;
  retain Weight Age;
  set Sashelp.Class;
  run;
  proc print data=class2;
  run;
```

The output displays the Weight and Age variables listed first because the RETAIN statement preceded the SET statement.
Output 5.10  Using the RETAIN Statement to Reorder Variables

<table>
<thead>
<tr>
<th>Obs</th>
<th>Weight</th>
<th>Age</th>
<th>Name</th>
<th>Sex</th>
<th>Height</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>112.5</td>
<td>14</td>
<td>Alfred</td>
<td>M</td>
<td>69.0</td>
</tr>
<tr>
<td>2</td>
<td>84.0</td>
<td>13</td>
<td>Alice</td>
<td>F</td>
<td>56.5</td>
</tr>
<tr>
<td>3</td>
<td>98.0</td>
<td>13</td>
<td>Barbara</td>
<td>F</td>
<td>65.3</td>
</tr>
<tr>
<td>4</td>
<td>102.5</td>
<td>14</td>
<td>Carol</td>
<td>F</td>
<td>62.8</td>
</tr>
<tr>
<td>5</td>
<td>102.5</td>
<td>14</td>
<td>Henry</td>
<td>M</td>
<td>63.5</td>
</tr>
<tr>
<td>6</td>
<td>83.0</td>
<td>12</td>
<td>James</td>
<td>M</td>
<td>57.3</td>
</tr>
<tr>
<td>7</td>
<td>84.5</td>
<td>12</td>
<td>Jane</td>
<td>F</td>
<td>59.8</td>
</tr>
<tr>
<td>8</td>
<td>112.5</td>
<td>15</td>
<td>Janet</td>
<td>F</td>
<td>62.5</td>
</tr>
<tr>
<td>9</td>
<td>84.0</td>
<td>13</td>
<td>Jeffrey</td>
<td>M</td>
<td>62.5</td>
</tr>
<tr>
<td>10</td>
<td>99.5</td>
<td>12</td>
<td>John</td>
<td>M</td>
<td>59.0</td>
</tr>
<tr>
<td>11</td>
<td>50.5</td>
<td>11</td>
<td>Joyce</td>
<td>F</td>
<td>51.3</td>
</tr>
<tr>
<td>12</td>
<td>90.0</td>
<td>14</td>
<td>Judy</td>
<td>F</td>
<td>64.3</td>
</tr>
<tr>
<td>13</td>
<td>77.0</td>
<td>12</td>
<td>Louise</td>
<td>F</td>
<td>56.3</td>
</tr>
<tr>
<td>14</td>
<td>112.0</td>
<td>15</td>
<td>Mary</td>
<td>F</td>
<td>66.5</td>
</tr>
<tr>
<td>15</td>
<td>150.0</td>
<td>16</td>
<td>Philip</td>
<td>M</td>
<td>72.0</td>
</tr>
<tr>
<td>16</td>
<td>128.0</td>
<td>12</td>
<td>Robert</td>
<td>M</td>
<td>64.8</td>
</tr>
<tr>
<td>17</td>
<td>133.0</td>
<td>15</td>
<td>Ronald</td>
<td>M</td>
<td>67.0</td>
</tr>
<tr>
<td>18</td>
<td>85.0</td>
<td>11</td>
<td>Thomas</td>
<td>M</td>
<td>57.5</td>
</tr>
<tr>
<td>19</td>
<td>112.0</td>
<td>15</td>
<td>William</td>
<td>M</td>
<td>66.5</td>
</tr>
</tbody>
</table>

Key Ideas

- The RETAIN statement is most often used to reorder variables simply because no other variable attribute specifications are required.
- The RETAIN statement has no effect on retaining values of existing variables being read from the data set.
- Only the variables whose positions are relevant need to be listed. Variables not listed in the RETAIN statement retain their original position.
- Use the RETAIN statement prior to the SET, MERGE, or UPDATE statement in order for you to reorder the variables.

See Also

- ARRAY Statement
- ATTRIB Statement
- FORMAT Statement
- INFORMAT Statement
- LENGTH Statement
Example: Reorder Variables Using the FORMAT Statement

Example Code

In the following example, the first DATA step creates the data set `investment` where the variables `Material`, `Item`, `Investment`, and `Profit` are defined.

The FORMAT statement causes the variable `Item` to be listed first in the output data set. The data set `investment` contains variables `Material`, `Item`, `Investment`, and `Profit` (in that order).

```plaintext
data investment;
  input Material $1-7 Item $9-15 Investment Profit;
  datalines;
    cotton shirts 2256354 83952175
    silk ties 498678 2349615
    silk suits 9482146 69839563
    leather belts 7693 14893
    leather shoes 7936712 22964
  ;
run;

data invest01;
  format Item Material $upcase9. Investment Profit dollar15.2;
  set investment;
run;
proc print data=invest01;
run;
```

The output below has a boxed red section that illustrates the `Item` variable listed first. The boxed orange section illustrates the FORMAT statement transformations where `Item` and `Material` variables are uppercased and `Investment` and `Profit` variables include dollar signs, commas, and two decimal places.

**Output 5.11 Using the FORMAT Statement to Reorder Variables**
Key Ideas

- You can control the order in which variables are displayed in SAS output by using the FORMAT statement.
- Use the FORMAT statement prior to the SET, MERGE, or UPDATE statement in order for you to reorder the variables.
- A single FORMAT statement can associate the same format with several variables, or it can associate different formats with different variables.
- When you use the FORMAT statement to reorder your variables, you do not have to associate a format with your variables.
- You can also use the INFORMAT statement to reorder your variables.

See Also

- ARRAY Statement
- ATTRIB Statement
- FORMAT Statement
- INFORMAT Statement
- LENGTH Statement
- RETAIN Statement
- CONTENTS Procedure

Examples: Convert Variable Types

Example: Convert Variable Types Using the PUT and INPUT Functions or Use the PUT and INPUT Functions to Convert Variable Types

Example Code

In the following example, the PUT and INPUT functions convert the data types of the variables. The example starts by creating a data set from raw instream data using the DATALINES statement. The data contains a listing of eight Oscar-winning films and the year in which the award was granted. The Year variable is initially defined as a character data type in the INPUT statement.

```sas
data oscars;
  input Title $ 1-20 Year $ 23-26 Rating $ 29-32
```
Duration 35-37 Gross 40-45 8.2;
datalines;
Slumdog Millionaire 2008 R 120 141.32
The Hurt Locker 2009 R 131 15.70
The King’s Speech 2010 R 118 138.80
The Artist 2011 PG13 100 44.67
Argo 2012 R 120 136.02
12 Years a Slave 2013 R 134 56.67
Birdman 2014 R 119 42.34
Spotlight 2015 R 128 44.99;
.proc contents data=oscars;
run;

The following example reads in the oscars data set and converts two of the
variables to another data type.
data convert;
  set oscars;
  ch_year = input(year, 5.);            /* 1 */
  ch_gross = put(gross, 8.2);           /* 2 */
  drop year gross;                      /* 3 */
  rename ch_year=Year ch_gross=Gross;   /* 4 */
  proc contents data=convert;
run;

1 Use the INPUT function to assign the value of the character variable year to
   ch_year. The informat 5. specifies that ch_year is a numeric data type with a
   width of 5.

2 Use the PUT function to assign the value of the numeric variable gross to the
   character variable ch_gross. The format 8.2 must be the same type as the
   source variable, gross in this example.

3 Remove the original year and gross variables.

4 Rename the ch_year and ch_gross variables to replace the variables with the
   new data type.

The output of PROC CONTENTS for both data sets shows

<table>
<thead>
<tr>
<th>PROC CONTENTS for Oscars Data Set</th>
<th>PROC CONTENTS for Convert Data Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Alphabetic List of Variables and Attributes</strong></td>
<td><strong>Alphabetic List of Variables and Attributes</strong></td>
</tr>
<tr>
<td>#</td>
<td>Variable</td>
</tr>
<tr>
<td>---</td>
<td>----------</td>
</tr>
<tr>
<td>1</td>
<td>Title</td>
</tr>
<tr>
<td>2</td>
<td>Year</td>
</tr>
<tr>
<td>3</td>
<td>Rating</td>
</tr>
<tr>
<td>4</td>
<td>Duration</td>
</tr>
<tr>
<td>5</td>
<td>Gross</td>
</tr>
<tr>
<td>3</td>
<td>Duration</td>
</tr>
<tr>
<td>5</td>
<td>Gross</td>
</tr>
<tr>
<td>2</td>
<td>Rating</td>
</tr>
<tr>
<td>1</td>
<td>Title</td>
</tr>
<tr>
<td>4</td>
<td>Year</td>
</tr>
</tbody>
</table>
Key Ideas

- If you define a numeric or character variable and assign the result of an expression to it that does not match the data type, SAS tries to convert the result of the expression to the correct value. Then it tries 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.
- 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. SAS uses the BESTw. format, where w is the width of the character variable and has a maximum value of 32.
- You cannot change the type of a variable. You will need to create a new variable from the old variable with the new data type and then drop and rename to keep the original names.

See Also

- Variable Type Conversions
- "INPUT Function" in SAS Functions and CALL Routines: Reference
- NOOBS Option
- PUT Function

Example: Convert Character Variables to Numeric

Example Code

This example shows how to explicitly convert character data values to numeric values. The INPUT function explicitly converts the rate variable to a numeric and rate has a length of 2, the numeric informat 2. is used to read the values of the variable. You can print the data set to see your new variable pay_chk with the numeric values.

```sas
data work.weeksal;
  set work.payscale;
  pay_chk=input(rate,2.)*Hours;
run;
proc print data=work.weeksal;
run;
```

Note: No conversion messages appear in the SAS log when the INPUT function is used.
Key Ideas

- Explicit conversions help you to control the data type and avoid having conversion errors.

See Also

- "INPUT Function" in SAS Functions and CALL Routines: Reference
- Explicit Character-to-Numeric Conversion

Example: Convert Numeric Variables to Character

Example Code

This example shows how to explicitly do numeric to character conversion using the PUT function. Use the PUT function in an assignment statement, where Site is the source variable. Because Site has a length of 2, choose 2. as the numeric format. The DATA step adds the new variable named Loc from the assignment statement to the data set. You use PROC PRINT to view your Loc variable.

data work.dept;
  set work.payscale;
  Loc=catx('/', put(Site,2.), Dept);
run;
proc print data=work.dept;
run;
### Key Ideas

- Parentheses or a minus sign preceding the number (without an intervening blank) indicates a negative value.
- Leading zeros and the placement of a value in the input field do not affect the value assigned to the variable. Leading zeros and leading and trailing blanks are not stored with the value. Unlike some languages, SAS does not read trailing blanks as zeros by default. To cause trailing blanks to be read as zeros, use the BZ. informat described in *SAS Formats and Informats: Reference*.
- Numeric data can have leading and trailing blanks but cannot have embedded blanks (unless they are read with a COMMA. or BZ. informat).
- To read decimal values from input lines that do not contain explicit decimal points, indicate where the decimal point belongs by using a decimal parameter with column input or an informat with formatted input. See the full description of the INPUT statement in *SAS Formats and Informats: Reference* for more information. An explicit decimal point in the input data overrides any decimal specification in the INPUT statement.

### See Also

- “INPUT Function” in *SAS Functions and CALL Routines: Reference*
- Explicit Numeric-to-Character Conversion
Example: Use Automatic Type Conversions

Example Code

By default, if you reference a character variable in a numeric context such as an arithmetic operation, SAS tries to convert the variable values to numeric. In the example, Rate is a character variable, but it is used in a numeric context. Therefore, SAS automatically converts the values of Rate to numeric to complete the arithmetic operation.

The automatic conversion of numeric data to character data is very similar to character-to-numeric conversion. Numeric data values are converted to character values whenever they are used in a character context. SAS automatically converts the numeric values of Site to character values because Site is used in a character context. The variable Site appears with the concatenation operator, which requires character values.

```sas
data work.autosal;
  set work.payscale;
  PayChk=Rate*Hours;
  Loc=Site||'/'||Dept;
run;
proc print data=work.autosal;
run;
```

Whenever data is automatically converted, a message is written to the SAS log stating that the conversion has occurred.

Output 5.14  SAS Log

```
NOTE: Character values have been converted to numeric values at the places given by: (Line):(Column).
     75:10
NOTE: Numeric values have been converted to character values at the places given by: (Line):(Column).
     76:7
NOTE: There were 9 observations read from the data set WORK.PAYSCALE.
NOTE: The data set WORK.AUTOSAL has 9 observations and 7 variables.
```
Key Ideas

- Automatic conversions occur in the following circumstances:
  - A character value is assigned to a previously defined numeric variable.
  - A character value is used in an arithmetic operation.
  - A character value is compared to a numeric value, using a comparison operator.
  - A character value is specified in a function that requires numeric arguments.
  - Numeric data values are converted to character values whenever they are used in a character context.

- The following statements are true about automatic conversion:
  - It uses the $w$. informat, where $w$ is the width of the character value that is being converted.
  - It produces a numeric missing value from any character value that does not conform to standard numeric notation (digits with an optional decimal point, leading sign, or scientific notation).

See Also

- Automatic Character-to-Numeric Conversion
- Automatic Numeric-to-Character Conversion

Examples: Use Automatic Variables

Example: Use the _N_ Automatic Variable

Example Code

The following example illustrates how to use the _N_ automatic variable with the PUT function.

```plaintext
data test;
  input x $ y;
  put 'This is row ' _n_;
datalines;
a 1
b 2
c 3
x 24
z 26
;```
The following is the output from the PRINT procedure.

**Output 5.16 Automatic Variable _N_ PRINT Output**

<table>
<thead>
<tr>
<th>Obs</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>24</td>
</tr>
<tr>
<td>5</td>
<td>z</td>
<td>26</td>
</tr>
</tbody>
</table>

**Key Ideas**

- Each time the DATA step loops past the DATA statement, the variable _N_ increments by 1.
- The value of _N_ represents the number of times the DATA step has iterated.

**See Also**

- Automatic Variables
- _N_ Automatic Variable

**Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/THEN Statement**

**Example Code**

The following example illustrates the use of _ERROR_ and _INFILE_ to write to the SAS log, during each iteration of the DATA step, the contents of an input record in which an input error is encountered.
data testerr;
  input x $ y;
  if _error_=1 then put 'Error before row ' _n_ ' which contains ' _infile_;
  put _infile_;
datalines;
  a 1
  *^*#
  b 2
  c3
; run;

The following is printed to the SAS log:

Example Code 5.2  SAS Log: _ERROR_ and _INFILE_

NOTE: Invalid data for y in line 65 2-2.

Error before row 2 which contains
b 2

Key Ideas

- 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 sent as output to the data set being created.
- _INFILE_ is an automatic character variable that gets created automatically by the DATA step when you use the INFILE statement. It contains the value of the current input record (row) read in either a file or in data in the DATALINES statement.
- Use the value of _ERROR_ to help locate errors in data records and to print an error message to the SAS log.

See Also

- Automatic Variables
- _ERROR_ Automatic Variable
- "_INFILE_=variable" in SAS DATA Step Statements: Reference
Examples: Use Variable Lists

Example: Create a Variable Numbered Range List

Example Code

This example shows you how to create variable numbered range lists. You can begin with any number and end with any number as long as you do not violate the rules for user-supplied names and the numbers are consecutive. Use the INPUT statement to write out a numbered range list. You can also use a numbered range list in an ARRAY statement.

```sas
data temperatures;
  input Day1-Day7;
datalines;
    74 75 82 84 85 86 89
;
run;
proc print data=temperatures noobs;
title "Average Daily Low Temperature";
run;
data tempCelsius(drop=i);
set temperatures;
array celsius{7} Day1-Day7;
do i=1 to 7;
  celsius{i}=(celsius{i}-32)*5/9;
end;
run;
proc print data=tempCelsius;
title "Average Daily Low Temperature in Celsius";
run;
```

The following graphic displays the PROC PRINT output for temperatures. The box highlighted in red shows the variable numbered range list Day1–Day7.
Key Ideas

- Numbered range variable lists are lists consisting of variables that are prefixed with the same name but that have different numeric values for the last characters.
- The numeric values must be consecutive numbers.
- In a numbered range list, you can refer to variables that were created in any order, provided that their names have the same prefix.

See Also

- ARRAY Statement
- INPUT Statement, List
- Types of Variable Lists

Example: Create a Variable Name Range List

Example Code

In the example, the name range list specified in the KEEP statement keeps all numeric variables between and including `nAtBat` and `nOuts`. The name range list specified in the ARRAY statement reads all variables between `nAtbat` and `nRuns`. In this case that also includes variables `nHits` and `nHome`.

```
data changeStats(where=(YrMajor>18));
  set sashelp.baseball;
```

Note: All variables in the range list must be the same case.

The name range list specified in the VAR statement in the PRINT procedure specifies that only the variables between and including `Name` and `nBB` are printed in the PROC PRINT output.
keep Name nAtBat-numeric-nOuts YrMajor;
array stats(4) nAtBat--nRuns;
do i=1 to 4;
   stats{i} = stats{i}*10;
end;
run;
proc print data=changeStats;
   var Name--nBB;
run;

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>nAtBat</th>
<th>nHits</th>
<th>nHome</th>
<th>nRuns</th>
<th>nRBI</th>
<th>nBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Baker, Dusty</td>
<td>2420</td>
<td>580</td>
<td>40</td>
<td>250</td>
<td>19</td>
<td>27</td>
</tr>
<tr>
<td>2</td>
<td>Nettles, Graig</td>
<td>3540</td>
<td>770</td>
<td>160</td>
<td>360</td>
<td>55</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>Rose, Pete</td>
<td>2370</td>
<td>520</td>
<td>0</td>
<td>150</td>
<td>25</td>
<td>30</td>
</tr>
<tr>
<td>4</td>
<td>Jackson, Reggie</td>
<td>4190</td>
<td>1010</td>
<td>180</td>
<td>650</td>
<td>58</td>
<td>92</td>
</tr>
<tr>
<td>5</td>
<td>Perez, Tony</td>
<td>2000</td>
<td>510</td>
<td>20</td>
<td>140</td>
<td>29</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Simmons, Ted</td>
<td>1270</td>
<td>320</td>
<td>40</td>
<td>140</td>
<td>25</td>
<td>12</td>
</tr>
</tbody>
</table>

Key Ideas

- You can use a name range list in an ARRAY declaration as long as you have already defined the variables prior to declaring the array. The variables can be defined in the same DATA step or in a previous DATA step.
- Notice that name range lists use a double hyphen ( -- ) to designate the range between variables, and numbered range lists use a single hyphen to designate the range.

See Also

- ARRAY Statement
- KEEP Statement
- VAR Statement
- Types of Variable Lists

Example: Use the OF Operator with a Variable List

Example Code

In the following example, arguments are passed in as numbered range lists, both with and without the use of the OF operator.

data _null_;  
x1=30; x2=20; x3=10;
\[ T = \text{sum}(x_1-x_3); \quad /* \text{#1} */ \]
\[ T_2 = \text{sum}(\text{OF } x_1-x_3); \quad /* \text{#2} */ \]
\[ \text{put } T=; \quad /* \text{#3} */ \]
\[ \text{put } T_2=; \quad /* \text{#4} */ \]
\[ \text{run; } \]

1. The first \textsc{sum} function returns \( T = 20 \).
2. The second \textsc{sum} function returns \( T_2 = 60 \).
3. Writes to the log the difference between \( x_1 \) and \( x_3 \).
4. Writes to the log the difference between \( x_1 \) and \( x_3 \) (inclusive).

**Key Ideas**

- The \textsc{OF} operator enables you to specify SAS \texttt{variable lists} or SAS \texttt{arrays} as arguments to functions.
- The \textsc{OF} operator is important when used with functions whose arguments are in the form of a numbered-range list. For example, an argument in the form of a numbered range (\( x_1 – x_n \)) is read in as a range of values only if the list is preceded by the \textsc{OF} operator.
- If the same list is not preceded by the \textsc{OF} operator and it is used with the \textsc{sum} function, the (-) character is treated as a subtraction sign. The function returns the difference between the variables rather than the sum of the range of values.

**See Also**

- \texttt{SUM Function}
- \texttt{PUT Statement}
- Using the \textsc{OF} Operator with Temporary Arrays

**Example: Use the \textsc{OF} Operator to Create Multiple Variable Lists**

**Example Code**

The following example illustrates a ranged variable list in which the entire list is preceded by the \textsc{OF} operator and the lists are separated by spaces or commas.

```
data _null_;  
T=sum(OF x_1-x_3 y_1-y_3 z_1-z_3);  
run;  
or  
data _null_;  
T=sum(OF x_1-x_3, OF y_1-y_3, OF z_1-z_3);  
run;  
```
Key Ideas

- The OF operator enables you to specify SAS variable lists or SAS arrays as arguments to functions.
- The OF operator is also used to distinguish one variable list from another when multiple ranged lists are used as arguments in functions.

See Also

- Types of Variable Lists
- The OF Operator with Functions and Variable Lists
- Using the OF Operator with Temporary Arrays

Examples: Manage Missing Variable Values

Example: Automatically Replacing Missing Values

Example Code

SAS replaces the missing values as it encounters values that you assign to the variables. Thus, if you use program statements to create new variables, their values in each observation are missing until you assign the values in an assignment statement, as shown in the following DATA step:

```sas
data new;
  input x;
  if x=1 then y=2;
  datalines;
  4
  1
  3
  1
;
run;
proc print data=new;
run;
```

y is created by the IF statement, so it is a numeric variable where all values are missing. When the IF statement is executed, for example, when x=1, the value of y is set to 2. Since no other statements set y’s value when x is not equal to 1, y remains missing (.) for those observations. This DATA step produces a SAS data set with the following variable values:
**Output 5.18**  DATA Step Output for Missing Values

<table>
<thead>
<tr>
<th>Obs</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

**Key Ideas**

- SAS replaces the missing values as it encounters values that you assign to the variables.
- At the beginning of each iteration of the DATA step, SAS sets the value of each variable that you create in the DATA step to missing.

**See Also**

- When Reading Raw Data

---

**Example: Creating Special Missing Values**

**Example Code**

The following example uses data from a marketing research company. Five testers were hired to test five different products for ease of use and effectiveness. If a tester was absent, there is no rating to report, and the value is recorded with an X for “absent.” If the tester was unable to test the product adequately, there is no rating, and the value is recorded with an I for “incomplete test.” The following program reads the data and displays the resulting SAS data set. Note the special missing values in the first and third data lines:

```sas
data period_a;
  missing X I;
  input Id $4. Foodpr1 Foodpr2 Foodpr3 Coffeem1 Coffeem2;
  datalines;
  1001 115 45 65 I 78
  1002 86 27 55 72 86
  1004 93 52 X 76 88
  1015 73 35 43 112 108
  1027 101 127 39 76 79
 ;

proc print data=period_a;
  title 'Results of Test Period A';
  footnote1 'X indicates TESTER ABSENT';
  footnote2 'I indicates TEST WAS INCOMPLETE';
```

---

Examples: Manage Missing Variable Values 143
run;
The following output is produced:

Output 5.19  Output with Multiple Missing Values

<table>
<thead>
<tr>
<th>Obs</th>
<th>Id</th>
<th>Foodpr1</th>
<th>Foodpr2</th>
<th>Foodpr3</th>
<th>Coffee1</th>
<th>Coffee2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1001</td>
<td>115</td>
<td>45</td>
<td>65</td>
<td>1</td>
<td>78</td>
</tr>
<tr>
<td>2</td>
<td>1002</td>
<td>86</td>
<td>27</td>
<td>55</td>
<td>72</td>
<td>86</td>
</tr>
<tr>
<td>3</td>
<td>1004</td>
<td>93</td>
<td>52</td>
<td>X</td>
<td>76</td>
<td>88</td>
</tr>
<tr>
<td>4</td>
<td>1015</td>
<td>73</td>
<td>35</td>
<td>43</td>
<td>112</td>
<td>108</td>
</tr>
<tr>
<td>5</td>
<td>1027</td>
<td>101</td>
<td>127</td>
<td>39</td>
<td>76</td>
<td>79</td>
</tr>
</tbody>
</table>

X indicates TESTER ABSENT
I indicates TEST WAS INCOMPLETE

Key Ideas

- When data values contain characters in numeric fields that you want SAS to interpret as special missing values, use the MISSING statement to specify those characters.
- If you do not begin a special numeric missing value with a period, SAS identifies it as a variable name. Therefore, to use a special numeric missing value in a SAS expression or assignment statement, you must begin the value with a period, followed by the letter or underscore. Here is an example: x=.d;

See Also

- MISSING Statement
- Representing Missing Values

Example: Preventing Propagation of Missing Values

Example Code

If you do not want missing values to propagate in your arithmetic expressions, you can omit missing values from computations by using the sample statistic functions. The SUM statement also ignores missing values, so the value of c is also 5. For example, consider the following DATA step:

```sas
data test;
  x=.;
y=5;
run;
```
a=x+y;
b=sum(x,y);
c=5;
c+x;
put a= b= c=;
run;

Adding $x$ and $y$ together in an expression produces a missing result because the value of $x$ is missing. The value of $a$, therefore, is missing. However, since the SUM function ignores missing values, adding $x$ to $y$ produces the value 5, not a missing value.

**Output 5.20  SAS Log Results for a Missing Value in a Statistic Function**

```
NOTE: Missing values were generated as a result of performing an operation on missing values.
Each place is given by: (Number of times) at (Line):(Column).
1 at 187:6
NOTE: The data set WORK.TEST has 1 observations and 5 variables.
NOTE: DATA statement used (Total process time):
   real time           0.01 seconds
   cpu time            0.00 seconds
```

**Key Ideas**

- The SUM function and SUM statement ignore missing values when making the calculation.
- If you use a missing value in an arithmetic calculation, SAS sets the result of that calculation to missing.
- SAS prints notes in the log to notify you which arithmetic expressions have missing values and when they were created.

**See Also**

- **MISSING Function**
- **Missing Variable Values**
Examples: Manage Problems Related to Precision

Example: Compare Imprecise Values in SAS

Example Code

In the example, SAS sets the variables `point_three` and `three_time_point_one` to 0.3 and \((3 \times 0.1)\), respectively. It then compares the two values by subtracting one from the other and writing the result to the SAS log:

```
data a;
  point_three=0.3;
  three_time_point_one=3*0.1;
  difference=point_three - three_times_point_one;
  put 'The difference is ' difference;
run;
```

The log output shows that \((3 \times 0.1) - 0.3\) does not equal 0, as it does in decimal arithmetic. The reason for this is because the values are stored in floating-point representation and cannot be stored precisely. For more information, see “Storage Format”.

Output 5.21 Log Output for Comparing Imprecise Values in SAS

![](image)

Key Ideas

- The numbers that are imprecise in decimal are not always the same ones that are imprecise in binary.
- Performing calculations and comparisons on imprecise numbers in SAS can lead to unexpected results.
- There are many decimal fractions whose binary equivalents are infinitely repeating binary numbers, so be careful when interpreting results from general rational numbers in decimal. There are some rational numbers that do not present problems in either number system.

See Also

- Numeric Precision
Example: Convert a Decimal Value to a Floating Point Representation

Example Code

This example shows the conversion process for the decimal value 255.75 to floating-point representation.

1 Use the base 2 number system to write out the value 255.75 in binary.

Note: Each bit in the mantissa represents a fraction whose numerator is 1 and whose denominator is a power of 2. The mantissa is the sum of a series of fractions such as 1/2, 1/4, 1/8, and so on. Therefore, for any floating-point number to be represented exactly, you must express it as the previously mentioned sum.

<table>
<thead>
<tr>
<th>Base 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^7</td>
</tr>
<tr>
<td>128</td>
</tr>
</tbody>
</table>

255.75 = 1 x 2^7 + 1 x 2^6 + 1 x 2^5 + 1 x 2^4 + 1 x 2^3 + 1 x 2^2 + 1 x 2^1 + 1 x 2^0 + 1 x 2^-1 + 1 x 2^-2

So, the value 255.75 is represented in binary format as 1111 1111.11

2 Move the decimal over until there is only one digit to the left of it. This process is called normalizing the value. Normalizing a value in scientific notation is the process by which the exponent is chosen so that the absolute value of the mantissa is at least one but less than ten. For this number, you move the decimal point 7 places:

1.111 1111 11

Because the decimal point was moved 7 places, the exponent is now 7.

3 The bias is 1023, so add 7 to 1023 to get

1030

4 Convert the decimal value, 1030, to hexadecimal using the base 16 number system:

<table>
<thead>
<tr>
<th>Base 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16^7</td>
</tr>
</tbody>
</table>

1030 = (4 \times 16^3) + (0 \times 16^2) + (6 \times 16^0)

= 1024 + 0 + 6

The converted hexadecimal value for 1030 is placed in the exponent portion of the final result.

5 Convert 406 to binary format:

0100  0000  0110

4 0 6

If the value that you are converting is negative, change the first bit to 1:

1100  0000  0110

This translates in hexadecimal to

C 0 6

6 In Step 2 above, delete the first digit and decimal (the implied one-bit):

11111111

7 Break these up into nibbles (half bytes) so that you have

1111 1111  1

8 To have a complete nibble at the end, add enough zeros to complete 4 bits:

1111 1111  1000

9 Convert

1111 1111  1000

to its hexadecimal equivalent to get the mantissa portion:

1111  1111  1000

F  F  8

The final floating-point representation for 255.75 is

406F F800 0000 0000

The final floating-point representation for −255.75 is

C06F F800 0000 0000

In this example, the starting decimal value, 255.75, conveniently converts to a finite binary value that can be represented without rounding in both binary and hexadecimal. The following section shows the conversion process for a decimal number that cannot be represented precisely in floating-point representation.
Key Ideas

- On Windows platforms, the processor performs computations in extended real precision.
- On Windows this allows storage of numbers larger than the basic IEEE floating-point format used by operating systems such as UNIX. This is one reason why you might see slightly different values from operating systems that use the same IEEE standard.

See Also

- Floating-Point Representation on Windows

Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation

Example Code

The following example shows the conversion process for the decimal value 512.1 to hexadecimal floating-point representation. This example illustrates how values that can be precisely represented in decimal cannot be precisely represented in hexadecimal floating point.

1 Because the base is 16, you must first convert the value 512.1 to hexadecimal notation.

2 First, convert the integer portion, 512, to hexadecimal using the base 16 number system:

<table>
<thead>
<tr>
<th>Base 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>16^7</td>
</tr>
<tr>
<td>16^6</td>
</tr>
<tr>
<td>268,435,456</td>
</tr>
</tbody>
</table>

The value 512 is represented in hexadecimal as 200.

3 Write the hexadecimal number, 200, in floating-point representation. To do this, move the decimal point all the way to the left, counting the number of positions that you moved it. The number that you moved it is the exponent:

\[ 200 = 0.200 \times 16^3 \]

4 Convert the fraction portion (.1) of the original number, 512.1 to hexadecimal:
.1 = \frac{1}{10} = \frac{1.6}{16}

The numerator cannot be a fraction, so keep the 1 and convert the .6 portion again.

.6 = \frac{6}{10} = \frac{9.6}{16}

Again, there cannot be fractions in the numerator, so keep the 9 and reconver the .6 portion.

The .6 continues to repeat as 9.6, which means that you keep the 9 and reconver. The closest that .1 can be represented in hexadecimal is

.1 = .1999999 \times 16^0

5 The exponent for the value is 3 (Step 2 above). To determine the actual exponent that will be stored, take the exponent value and add the bias to it:

true exponent + bias = 3 + 40 = 43 (hexadecimal) = stored exponent

The final portion to be determined is the sign of the mantissa. By convention, the sign bit for positive mantissas is 0, and the sign for negative mantissas is 1. This information is stored in the first bit of the first byte. From the hexadecimal value in Step 4, compute the decimal equivalent and write it in binary format. Add the sign bit to the first position. The stored value now looks like this:

43 \text{ hexadecimal} = (4 \times 16^1) + (3 \times 16^0) = 67 \text{ decimal} = 01000003 \text{ binary}

11000003 = 195 \text{ in decimal} = C3 \text{ in hexadecimal}

6 The final step is to put it all together:

4320019999999999 - floating point representation for 512.1

C320019999999999 - floating point representation for -512.1

Therefore, the decimal value 512.1 cannot be precisely represented in binary or hexadecimal floating point notation. When the number 512.1 is converted, the result is an infinitely repeating number. This is analogous to representing the fraction 1/3 in decimal form.

The closest approximation is .33333333 with infinitely repeating ‘3s’.

Key Ideas

- Values that can be represented exactly in decimal notation cannot always be represented precisely in floating-point notation.
- If a floating-point value has a repeating pattern of numbers, there is a good chance that the value cannot be represented exactly.

See Also

- Floating-Point Representation on Windows
Example: Round Values to Avoid Computational Errors

Example Code

The following example shows how you can use the ROUND function to round the results for one iteration of the DATA step.

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

The following is printed to the SAS log:

**Output 5.22  SAS Log Output for the ROUND Function**

```
i=-1
i=-0.9
i=-0.8
i=-0.7
i=-0.6
i=-0.5
i=-0.4
i=-0.3
i=-0.2
i=-0.1
i=0
AT ZERO
i=0.1
i=0.2
i=0.3
i=0.4
i=0.5
i=0.6
i=0.7
i=0.8
i=0.9
i=1
```

Example Code

You can avoid comparison errors by explicitly rounding the values before performing the comparison. The next example compares the calculated result of $1/3$ to the assigned value $0.33333$. Because $1/3$ is an imprecise number, the value is not equal to $0.33333$, and the PUT statement is not executed. However, if you add the ROUND function, as in the following example, the PUT ‘MATCH’ statement is executed:

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

- Errors that are caused by the accumulation of performing calculations on imprecise values can be resolved by rounding.
- In general, if you are doing comparisons with fractional values, it is good practice to use the ROUND function before performing any computations or comparisons.

See Also

- "ROUND Function" in SAS Functions and CALL Routines: Reference

Example: Use the LENGTH Statement to Compare Values

Example Code

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:

```sas
data ab;
  length x 2;
  x=257;
  y1=x+1;
data abc;
  set ab;
  if x=257 then put 'FOUND';
  else put 'NOT FOUND';
  y2=x+1;
run;
```

The following is printed to the SAS log:
Example Code 5.3  SAS Log

```
337  data ab;
338    length x 2;
339    x=257;
340    y1=x+1;
352  ERROR 352-185: The length of numeric variables is 3-8.
341  data abc;
342    set ab;
343    if x=257 then put 'FOUND';
344      else put 'NOT FOUND';
345    y2=x+1;
346  run;
```

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.AB may be incomplete. When this step was stopped there were 0 observations and 2 variables.
WARNING: Data set WORK.AB was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
    real time 0.01 seconds
    cpu time 0.01 seconds

```
341  data abc;
342    set ab;
343    if x=257 then put 'FOUND';
344      else put 'NOT FOUND';
345    y2=x+1;
346  run;
```

NOTE: There were 0 observations read from the data set WORK.AB.
NOTE: The data set WORK.ABC has 0 observations and 3 variables.
NOTE: DATA statement used (Total process time):
    real time 0.01 seconds
    cpu time 0.00 seconds

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.

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 truncated only 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.
Key Ideas

- You can use the LENGTH statement to control the number of bytes that are used to store variable values. However, you must use it carefully to avoid errors and significant data loss.
- During compilation SAS allocates as many bytes of storage space as there are characters in the first value that it encounters for that variable.

See Also

- “LENGTH Statement” in SAS DATA Step Statements: Reference
- Using the LENGTH Statement When Comparing Values

Example: Compare Values That Have Imprecise Representations

Example Code

In decimal arithmetic, the expression 15.7 – 11.9 = 3.8 is true. But, in SAS, if you compare the literal value of 3.8 to the calculated value of 15.7 – 11.9 and output the result to the SAS log, you will get a result of ‘not equal’.

```
data a;
  x=15.7-11.9;
  if x=3.8 then put 'equal';
  else put 'not equal';
run;
proc print data=a;
run;
```

The log output indicates that the values 3.8 and (15.7 – 11.9) are not equivalent. This is because the values involved in the computation cannot be precisely represented in floating-point representation.

Output 5.24  Log Output for Comparing Values That Have Imprecise Representations

```
not equal
```

The PROC PRINT statement displays the value for x as 3.8 rather than the actual stored value because the procedure automatically applies a format and rounds the results before displaying them. This example shows how non-explicit rounding can cause confusion because, in this case, PROC PRINT rounds only the final results after they are calculated.
Output 5.25  Output for Comparing Values

The SAS System

<table>
<thead>
<tr>
<th>Obs</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Key Ideas

- When comparing non-integer values that do not have precise floating-point representations you can sometimes encounter surprising results.

See Also

- Floating-Point Representation

Example: Confirm Precision Errors Using Formats

Example Code

In the next example, two different formats are applied to the results given and displayed in the SAS log. The first format, 10.8 shows that the value of \( x \) is 3.8; however, displaying the value using the 18.16 format indicates that \( x \) is slightly less than 3.8.

```sas
data a;
x=15.7-11.9;
if x=3.8 then put 'equal';
else put 'not equal';
put x=10.8;
put x=18.16;
run;
```

Output 5.26  Log Output: Using Formats to Confirm Precision Errors

```
not equal
x=3.80000000
x=3.7999999999999900
```

Example Code

You can also use the width of 16 with the HEXw.d format to show floating-point representation.

```sas
data a;
```
x=15.7-11.9;
if x=3.8 then put 'equal';
    else put 'not equal';
put x=hex16.;
run;

Output 5.27  Using the HEX16 Format to Verify Calculated Results

not equal
x=400B666666666664

Key Ideas

- The HEXw. format is a special format that can be used to show floating-point representation.
- When comparing non-integer values that do not have precise floating-point representations you can sometimes encounter surprising results.

See Also

- "HEXw. Format" in SAS Formats and Informats: Reference
- "Dictionary of Formats" in SAS Formats and Informats: Reference
- Floating-Point Representation

Example: Determine How Many Bytes Are Needed to Store a Number Accurately

Example Code

You can also use the TRUNC function to determine the minimum number of bytes that are needed to store a value accurately. The following program finds the minimum length of bytes (MinLen) that are needed for numbers stored in a native SAS data set named Numbers in an IBM mainframe environment. The data set Numbers contains the variable Value. Value contains a range of numbers from 269 to 272:

```sas
   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 data=temp noobs;
  var value minlen;
run;

The following output shows the results from this example:

Output 5.28  Determining How Many Bytes Are Needed to Store a Number Accurately

<table>
<thead>
<tr>
<th>value</th>
<th>minlen</th>
</tr>
</thead>
<tbody>
<tr>
<td>269</td>
<td>3</td>
</tr>
<tr>
<td>270</td>
<td>3</td>
</tr>
<tr>
<td>271</td>
<td>3</td>
</tr>
<tr>
<td>272</td>
<td>2</td>
</tr>
</tbody>
</table>

Key Ideas

- 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.

See Also

- “TRUNC Function” in SAS Functions and CALL Routines: Reference
Example: Generate Inaccurate Results For Large Data

Example Code

The following example illustrates an inaccurate result for large data.

```sas
data _null_;  
lo2hi =0;  
hi2lo =0;  
do i = -10 to 10 by 0.1;  
  lo2hi = lo2hi + 10**i;  
end;  
do i = 10 to -10 by -0.1;  
  hi2lo = hi2lo + 10**i;  
end;  
diff = hi2lo-lo2hi;  
put lo2hi;  
put hi2lo;  
put diff;  
run;
```

The following output shows the log output from this example:

```
Example Code 5.4   SAS Log

lo2hi=48621160939  
hi2lo=48621160939  
diff=0.0041885376
```

Key Ideas

- Calculated statistics can vary slightly, depending on the order in which observations are processed. Such variations are due to numerical error introduced by floating-point arithmetic, the results of which should be considered approximate and not exact.
- Order of observation processing can be affected by non-deterministic effects of multi-threaded or parallel processing.
- Order of processing can also be affected by inconsistent or non-deterministic ordering of observations produced by a data source, such as a DBMS delivering query results through an ACCESS engine.

See Also

- "DO Statement: Iterative" in SAS DATA Step Statements: Reference
- "Accuracy on x64 Windows Processors"
Examples: Encrypt Variable Values

Example: Create a Simple 1-Byte-to-1-Byte Swap Using the TRANSLATE Function

Example Code

This example shows how to use a simple 1-byte-to-1-byte swap with the TRANSLATE function.

data sample1;          /*1*/
   input @1 name $;
   length encrypt decrypt $ 8;
   /*ENCRYPT*/
   do i = 1 to 8;      /*2*/
      encrypt=strip(encrypt)||translate(substr(name,i,1),
         '0123456789!@#$%^&*()-=,./?<','ABCDEFGHIJKLMNOPQRSTUVWXYZ');
   end;
   /*DECRYPT*/
   do j = 1 to 8;      /*3*/
      decrypt=strip(decrypt)||translate(substr(encrypt,j,1),
         'ABCDEFGHIJKLMNOPQRSTUVWXYZ','0123456789!@#$%^&*()-=,./?<');
   end;
   drop i j;
datalines;
ROBERT
JOHN
GREG
;
proc print;
run;*/

1 The DATA step reads a name that is 8 or fewer characters in length and uses a DO loop to process the TRANSLATE function and SUBSTR function 1 byte at a time.

2 The first DO loop creates the encrypted value.

3 The second DO loop creates the decrypted value by reversing the order and returning the original value.
Key Ideas

- SAS provides encryption for SAS data sets with the ENCRYPT= data set option, but this option is typically used to encrypt data at the data set level.
- To encrypt data at the SAS variable level, you can use a combination of DATA step functions and logic to create your own encryption and decryption algorithms.
- However, if you create your own algorithms, it is important that you create a program that not only is secure and hidden from public view, but that also contains methods to both encrypt and decrypt the data.

See Also

- SUBSTR-left Function
- SUBSTR-right Function
- TRANSLATE Function

Example: Use a 1-Byte-to-2-Byte Swap Using the TRANWRD Function

Example Code

This example shows how to encrypt values using a 1-byte-to-2-byte swap with the TRANWRD function. In the following sample code, the DATA step reads an ID that is 6 or fewer characters in length. However, the variable is assigned a length of 12 to double the character length, because this is a 1-byte-to-2-byte exchange. A DO loop processes the TRANWRD function 1 byte at a time.

```sas
data sample2;           /*1*/
  input @1 id $12.;
  encrypt=id;            /*ENCRYPT*/
i=21;
do from_1 = "C","F","E","A","D","B";   /*2*/
to_1=put(i,2.);
  encrypt=tranwrd(encrypt,from_1,to_1);
i+1;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>encrypt</th>
<th>decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ROBERT</td>
<td>14%</td>
<td>ROBERT</td>
</tr>
<tr>
<td>2</td>
<td>JOHN</td>
<td>7%6</td>
<td>JOHN</td>
</tr>
<tr>
<td>3</td>
<td>GREG</td>
<td>6%6</td>
<td>GREG</td>
</tr>
</tbody>
</table>
The DATA step reads an ID that is 6 or fewer characters in length. However, the variable is assigned a length of 12 to double the character length, because this is a 1-byte-to-2-byte exchange.

The first DO loop creates the encrypted value.

The second DO loop creates the decrypted value by reversing the order and returning the original value. New variables are assigned to the ID variable before the TRANWRD function to avoid overwriting the original ID variable.

**Output 5.30 PROC PRINT Result: A Simple 1-Byte-to-2-Byte Swap**

<table>
<thead>
<tr>
<th>Obs</th>
<th>id</th>
<th>encrypt</th>
<th>decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABCDEF</td>
<td>242621252322</td>
<td>ABCDEF</td>
</tr>
<tr>
<td>2</td>
<td>FEDC</td>
<td>22232521</td>
<td>FEDC</td>
</tr>
<tr>
<td>3</td>
<td>ACE</td>
<td>242123</td>
<td>ACE</td>
</tr>
<tr>
<td>4</td>
<td>BDFA</td>
<td>26252224</td>
<td>BDFA</td>
</tr>
<tr>
<td>5</td>
<td>CAFDEB</td>
<td>212422252325</td>
<td>CAFDEB</td>
</tr>
<tr>
<td>6</td>
<td>BADC</td>
<td>2624252122</td>
<td>BADC</td>
</tr>
<tr>
<td>7</td>
<td>ABC</td>
<td>242621</td>
<td>ABC</td>
</tr>
</tbody>
</table>
Key Ideas

- SAS provides encryption for SAS data sets with the ENCRYPT= data set option, but this option is typically used to encrypt data at the data set level.
- To encrypt data at the SAS variable level, you can use a combination of DATA step functions and logic to create your own encryption and decryption algorithms.
- However, if you create your own algorithms, it is important that you create a program that not only is secure and hidden from public view, but that also contains methods to both encrypt and decrypt the data.

See Also

- TRANWRD Function

Example: Use Different Functions to Encrypt Numeric Values as Character Strings

Example Code

This example shows how to encrypt a numeric value to create a character value using a different character every third time. This method uses the PUT, SUBSTR, INDEXC, TRANSLATE, CATS, and INPUT functions, as well as array processing.

```sas
data sample3;

/* 1 */
input num;
array from(3) $ 10 from1-from3 ('0123456789','0123456789','0123456789'); /* 2 */
array to(3) $ 10 to1-to3 ('ABCDEFGHIJ','KLMNOPQRSTUVWXYZ','UVWXYZABCD');
array old(5) $ old1-old5;
array new(5) $ new1-new5;
char_num=put(num,5.);
/* 3 */
do i = 1 to 5; /* 4 */
   old(i)=substr(char_num,i,1);
end;
j=1;
do k = 1 to 5; /* 5 */
   if indexc(old(k),from(j)) > 0 then do;
      new(k)=translate(old(k),to(j),from(j));
      j+1;
      if j=4 then j=1;
   end;
end;
```
end;
encrypt_num=cats(of new1-
new5);                         /*6*/
keep num encrypt_num;
datalines;
12345
70707
99
1111
;
r
run;
data
sample4;
/*7*/
set sample3;
array to(3) $ 10 to1-to3 ('0123456789','0123456789','0123456789');
array from(3) $ 10 from1-from3
('ABCDEFGHIJ','KLMNOPQRST','UVWXYZABCD');
array old(5) $ old1-old5;
array new(5) $ new1-new5;
do i = 1 to 5;
   old(i)=substr(encrypt_num,i,1);
end;
j=1;
do k = 1 to 5;
   if indexc(old(k),from(j)) > 0 then do;
      new(k)=translate(old(k),to(j),from(j));
      j+1;
   if j=4 then j=1;
   end;
decrypt_num=input(cats(of new1-new5),5.);
keep num encrypt_num decrypt_num;
run;
proc print;
run;

The first DATA step reads numeric values that are 5 digits or fewer. The numeric variable is converted to a character variable and is split into five separate values.

Four ARRAY statements are used: the first array sets up the from values; the second sets up the to values; the third holds the five separate numeric values; and the fourth holds the five new, separate encrypted values. The from and to arrays are each created with three elements. The from ARRAY is assigned the same string of numbers for all three elements, and the to ARRAY is assigned a different string of letters for each of the three elements to build the every-third-time rotating pattern.

The PUT function converts the numeric value to a character value.

The first DO loop uses the SUBSTR function to split the value into five separate values and assigns each to the old ARRAY.

The second DO loop translates each value by using the INDEXC function to find the original number in the from ARRAY and, if found, translates the value using the from ARRAY, and rotates through the list of elements every third time.
6 The encrypted value is created by using the CATS function to concatenate the five translated values. The same process that is used to encrypt the values is also used to decrypt the values. The only differences are that the encrypted variable is passed to the SUBSTR function, and the final decrypted variable is passed to the INPUT function following the CATS function. This is done so that the final values are numeric values.

7 The second DATA step reverses the encryption done in the first DATA step, converting the values back to their original values. If you compare the two DATA steps, you can see that the values in the to and from arrays are reversed. This is because the second DATA step reverses the encryption done in the first DATA step, converting the values back to their original values.

**Output 5.31**  PROC PRINT Result: Use Different Functions to Encrypt Numeric Values as Character Strings

<table>
<thead>
<tr>
<th>Obs</th>
<th>num</th>
<th>encrypt_num</th>
<th>decrypt_num</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12345</td>
<td>6M6XEP</td>
<td>12345</td>
</tr>
<tr>
<td>2</td>
<td>70707</td>
<td>HkBAR</td>
<td>70707</td>
</tr>
<tr>
<td>3</td>
<td>99</td>
<td>JT</td>
<td>99</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
<td>5LVB</td>
<td>1111</td>
</tr>
</tbody>
</table>

**Key Ideas**

- SAS provides encryption for SAS data sets with the ENCRYPT= data set option, but this option is typically used to encrypt data at the data set level.
- To encrypt data at the SAS variable level, you can use a combination of DATA step functions and logic to create your own encryption and decryption algorithms.
- However, if you create your own algorithms, it is important that you create a program that not only is secure and hidden from public view, but that also contains methods to both encrypt and decrypt the data.

**See Also**

- CATS Function
- INDEXC Function
- INPUT Function
- PUT Function
- SUBSTR-left Function
- SUBSTR-right Function
- TRANSLATE Function
Data Types

Data Types in SAS

A data type is an attribute of a SAS variable that tells SAS what kind of data the value is. SAS variables can be defined as either character (CHAR) or numeric (NUMERIC). For information about data types in SAS, see “Data Types” on page 85.

- Arrays in SAS on page 499
- “About SAS Date, Time, and Datetime Values” in SAS Language Reference: Concepts
- Chapter 8, “SAS Constants,” on page 171
- Chapter 5, “Variables,” on page 69
SAS Expressions

Definitions

expression
is a sequence of operands and operators that form a set of instructions that are performed to produce a resulting value.

operands
are constants or variables that can be numeric or character.

operators
are symbols that represent a comparison, arithmetic calculation, or logical operation; a SAS function; or grouping parentheses.

simple expression
is an expression with no more than one operator. A simple expression can consist of one of the following single operators:

- constant
- variable
- function

compound expression
is an expression that includes several operators. When SAS encounters a compound expression, it follows rules to determine the order in which to evaluate each part of the expression.
WHERE expression is a type of SAS expression that is used within a WHERE statement or WHERE= data set option to specify a condition for selecting observations for processing in a DATA or PROC step.

See Also

- Operators
- Conditionally Selecting Data
- WHERE Statement

Use of SAS Expressions

SAS expressions are used in assignment statements and many other SAS programming statements to do the following:

- transform variables
- create new variables
- conditionally process variables
- calculate new values
- assign new values

SAS expressions can resolve to numeric values, character values, or Boolean values.

SAS Constants in Expressions

A SAS constant is a number or character string that indicates a fixed value. Constants can be used as expressions in many SAS statements. For more information, see “SAS Constants in Expressions” on page 171.

SAS Variables in Expressions

A SAS variable can be used in an expression. However, if the variable value does not match the data type called for then SAS attempts to convert the value to the expected type. For more information, see SAS variables on page 71.
SAS Functions in Expressions

A SAS function is a keyword that you use to perform a specific computation or system manipulation. Functions return a value, might require one or more arguments, and can be used in expressions. For more information about SAS functions, see *SAS Functions and CALL Routines: Reference*.

SAS Operators in Expressions

A SAS operator is a symbol that represents a comparison, arithmetic calculation, or logical operation; a SAS function; or grouping parentheses. For more information, see “SAS Operators” on page 189.
SAS Constants in Expressions

Definitions

A SAS constant is a number or a character string that indicates a fixed value. Constants can be used as expressions in many SAS statements, including variable assignment and IF-THEN statements. Here are some examples of constants used in SAS expressions:

- x=10;
- name="James";
- date=01/23/2018;

They can also be used as values for certain options. Constants are also called literals. The following are types of SAS constants:

- character constants
- numeric constants
Character Constants

Definition
A character constant consists of 1 to 32,767 characters and must be enclosed in quotation marks. The quotation marks can be either single or double quotation marks. 'Tom' and "Tom" are equivalent.

Character Constants and Character Variables
It is important to remember that character constants are enclosed in quotation marks, but names of variables are not. This distinction applies wherever you can use a character constant, such as in titles, footnotes, labels, and other descriptive strings; in option values; and in operating environment-specific strings, such as file specifications and commands.

The following statements use character constants:
- x='abc';
- if name='Smith' then do;

The following statements use character variables:
- x=abc;
- if name=Smith then do;

In the second set of examples, SAS searches for variables named ABC and SMITH, instead of constants.

Note: SAS distinguishes between uppercase and lowercase when comparing character constants. For example, the character constants 'Smith' and 'SMITH' are not equivalent.

Apostrophes and Quotation Marks within Character Constants
- If a character constant includes a single quotation mark, enclose it in double quotation marks.
- If a character constant includes an apostrophe, you can enclose the string in single quotation marks and express the apostrophe as two consecutive quotation marks. SAS treats the two consecutive quotation marks as one quotation mark.
  - name='Tom' 's'
- If a character constant includes a single quotation mark or an apostrophe, you can enclose the string and the apostrophe in two consecutive, double quotation marks. SAS treats the two consecutive double quotation marks as one double quotation mark.
CAUTION

Matching quotation marks correctly is important. Missing or extraneous quotation marks cause SAS to misread both an erroneous statement and the statements that follow it. For example, in name='O'Brien';, O is the character value of Name, Brien is extraneous, and '; begins another quoted string.

Character Constants Expressed in Hexadecimal Notation

SAS character constants can be expressed in hexadecimal notation, with each character represented by a pair of hexadecimal characters. The character constant string is enclosed in single or double quotation marks, followed immediately by an X, as in this example:

'534153'x

Commas can be used to make the string more readable, but they are not part of and do not alter the hexadecimal value, as in this example:

'31,32,33,34'x

Note: Trailing or leading blanks within the quotation marks cause an error message to be written to the log.

For example, see "Example: Define Character Constants in Hexadecimal Notation" on page 181.

Avoiding a Common Error with Character Constants

Always put a blank space after a character constant. Otherwise, SAS might interpret the letters that follow the character constant as the type of value of the character constant.

In the following statement, '821't is evaluated as a time constant because there is no space between the string and the t in then.

if flight='821'then flight='230';

The following lines appear in the SAS log:

```
ERROR: Invalid date/time/datetime constant '821't.
ERROR 77-185: Invalid number conversion on '821't.
```

A space after the ending quotation mark eliminates this misinterpretation.

**Table 8.1 Characters That Cause Misinterpretation When Following a Character Constant**

<table>
<thead>
<tr>
<th>Character</th>
<th>Possible Interpretation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>bit testing constant</td>
<td>'00100000'b</td>
</tr>
<tr>
<td>d</td>
<td>date constant</td>
<td>'01jan04'd</td>
</tr>
<tr>
<td>dt</td>
<td>datetime constant</td>
<td>'18jan2005:9:27:05am'dt</td>
</tr>
</tbody>
</table>
### Numeric Constants

#### Definition

A numeric constant is a number that appears in a SAS statement. Numeric constants can be presented in many forms, including these:

- standard notation
- scientific (E) notation
- hexadecimal notation

#### Numeric Constants Expressed in Standard Notation

Most numeric constants are written just as numeric data values are. The numeric constant in the following expression is 100:

```
part/all*100
```

Numeric constants can be expressed in standard notation in the following ways:

#### Table 8.2 Standard Notation for Numeric Constants

<table>
<thead>
<tr>
<th>Numeric Constant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>is an unsigned integer</td>
</tr>
<tr>
<td>-5</td>
<td>contains a minus sign</td>
</tr>
<tr>
<td>+49</td>
<td>contains a plus sign</td>
</tr>
</tbody>
</table>
Numeric constants that are larger than \((10^{32})^{-1}\) must be written using scientific notation. For example, a number such as 2E4 would need to be written in scientific notation.

### Numeric Constants Expressed in Scientific Notation

In scientific notation (such as 2E4), the number before the E is multiplied by the power of ten, which is indicated by the number after the E. For example, 2E4 is the same as \(2 \times 10^4\) or 20,000.

### Numeric Constants Expressed in Hexadecimal Notation

A numeric constant that is expressed in hexadecimal notation starts with a numeric digit (usually 0). It can be followed by more hexadecimal characters, and ends with the letter X. The constant can contain up to 16 valid hexadecimal characters (0 to F).

### See Also

- “Example: Define Numeric Constants in Standard Notation”
- “Example: Define Numeric Constants in Scientific Notation”
- “Example: Define Numeric Constants in Hexadecimal Notation”

### Statements

- FORMAT Statement

### Date, Time, and Datetime Constants

A date constant, time constant, or datetime constant is a date or time or datetime in single or double quotation marks, followed by a D (date), T (time), or DT (datetime) to indicate the type of value.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Format</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>'ddmm&lt;yy&gt;yy'D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'ddmm&lt;yy&gt;yy'D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'1jan2018'd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'01jan18'd</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>'hh:mm&lt;ss.s&gt;T</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'9:25't</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'9:25:19pm't</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>Format</td>
<td>Examples</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>-------------------------------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Datetime</td>
<td>'ddmm&lt;yy&gt;yy:hh:mm:ss.s'DT</td>
<td>'01may18:9:30:00' dt</td>
</tr>
<tr>
<td></td>
<td>&quot;ddmm&lt;yy&gt;yy:hh:mm:ss.s&quot;DT</td>
<td>&quot;18jan2018:9:27:05am&quot;dt</td>
</tr>
<tr>
<td>Universal Coordinate Time (UTC)</td>
<td>'yyyy-mm-ddThh:mm:ssZ'DT</td>
<td>'2018-05-17T09:15:30–05:00'dt</td>
</tr>
<tr>
<td>(ISO 8601 standard)</td>
<td>'yyyy-mm-ddThh:mm:ss+-hh:ss'DT</td>
<td>'2018-07-20T12:00:00+00:00'dt</td>
</tr>
</tbody>
</table>

Trailing blanks or leading blanks that are included within the quotation marks do not affect the processing of the date constant, time constant, or datetime constant.

See Also

Express Date, Time, and Datetime Values in Date on page 184

Bit Testing Constants

A bit testing constant is a bit mask that is used in bit testing to compare internal bits in a value’s representation. You can perform bit testing on both character and numeric variables.

The general form of the operation is:

```
expression comparison-operator bit-mask
```

Here are the components of the bit testing operation:

expression
- can be any valid SAS expression. Both character and numeric variables can be bit tested.

comparison-operator
- compares an expression with the bit mask. For more information, see Operators.

bit-mask
- is a string of 0s, 1s, and periods in quotation marks that is immediately followed by a B, such as ‘..1.0000'b. Zeros test whether the bit is off; ones test whether the bit is on; and periods ignore the bit. Commas and blanks can be inserted in the bit mask for readability without affecting its meaning.

CAUTION
**Truncation can occur when SAS uses a bit mask.** If the expression is longer than the bit mask, SAS truncates the expression before it compares it with the bit mask. A false comparison might result. An expression's length (in bits) must be less than or equal to the length of the bit mask. If the bit mask is longer than a character expression, SAS generates a warning in the log, stating that the bit mask is truncated on the left, and continues processing.

**Note:** Bit masks cannot be used as bit literals in assignment statements. For example, the following statement is not valid:

```sas
x='0101'b;    /* incorrect*/
```

**TIP** The `$BINARYw.` and `BINARYw.` formats and the `$BINARYw.d`, `BINARYw.d`, and `BITSw.d` informats can be useful for bit testing. You can use them to convert character and numeric values to their binary values, and vice versa, and to extract specified bits from input data. For complete descriptions of these formats and informats, see *SAS Formats and Informats: Reference*.

### See Also

#### Examples

- *Example: Bit Test a Variable’s Value*

#### Formats

- `$BINARYw. Format`

### Examples: Expressions and Constants

#### Example: Use Character Constants in Expressions

**Example Code**

The following example illustrates how to create character constants using single and double quotation marks:

```sas
data example;
  char_const = 'Tom';           /*1*/
  apostrophe = "Tom's";         /*2*/
  singlequote = 'Tom''s';       /*3*/
run;
proc print data=example;
```

---

*Examples: Expressions and Constants*
run;

1 A character constant is created using single quotation marks around the string value.

2 A character constant contains an apostrophe, which is created using double quotation marks on the outside of the string. A single quotation mark on the inside represents the apostrophe.

3 You can also use two single quotation marks in the string with single quotation marks around the whole value.

Output 8.1 Results of Single and Double Quotations in Character Constants

<table>
<thead>
<tr>
<th>Obs</th>
<th>char_const</th>
<th>apostrophe</th>
<th>singlequote</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tom</td>
<td>Tom's</td>
<td>Tom's</td>
</tr>
</tbody>
</table>

Key Ideas

- A character constant is a fixed-value that consists of 1 to 32,767 characters and must be enclosed in quotation marks. Character constants can also be represented in hexadecimal form.
- A character variable contains alphabetic characters, numeric digits 0 through 9, and other special characters.
- Matching quotation marks correctly is important.
- Missing or extraneous quotation marks cause SAS to misread the statement and generate an error.
- If a character constant contains an apostrophe (or single quotation mark), it could be created by enclosing the whole string in double quotation marks.

See Also

- “Character Constants and Character Variables”
- “Apostrophes and Quotation Marks within Character Constants”
- “Example: Use Quotation Marks Within Strings”

Example: Compare Character Constants with Character Variables

Example Code

The following example illustrates creating character constants with character variables:

```
data compare;
```
var1 = 'abc'; /* 1 */
var2 = 'def'; /* 2 */
name1 = var1; /* 3 */
name2 = var2; /* 4 */
run;

proc print data=compare;
run;

1 Create character variables, var1 and var2, by placing their assigned values in single or double quotation marks. The variables var1 and var2 are created from character constants because their values are contained within quotation marks.

2 Create character variables name1 and name2. Do not place the values in quotation marks. The value for the variables is the name of the character constants created in Step 1.

3 The variable name1 is a character variable whose value is the value of var1. The value for var1 is 'abc'.

4 The variable name2 is a character variable whose value is the value of var2. The value for var2 is 'def'.

The variables name1 and name2 are not created from character constants. They are character variables because the values assigned to them are not contained within quotation marks. Their values are the names of the previously created character variables var1 and var2. If you assign a character string to a variable and do not use quotation marks around the value, SAS expects the value to be the name of an existing variable.

Output 8.2 Results of Character Constants and Character Variable Comparison

<table>
<thead>
<tr>
<th>Obs</th>
<th>var1</th>
<th>var2</th>
<th>name1</th>
<th>name2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
<td>def</td>
<td>abc</td>
<td>def</td>
</tr>
</tbody>
</table>

Key Ideas

- Character constants are enclosed in quotation marks, but variable names are not.
- A character constant consists of 1 to 32,767 characters. Character constants can also be represented in hexadecimal form.
- A character variable is a variable of type character that contains alphabetic characters, numeric digits 0 through 9, and other special characters.
- This distinction applies wherever you can use a character constant, such as in titles, footnotes, labels, and other descriptive strings.
- SAS distinguishes between uppercase and lowercase when comparing character expressions.

See Also

- “Character Constants and Character Variables”
- “Apostrophes and Quotation Marks within Character Constants”
Example: Use Quotation Marks Within Strings

Example Code

The following example illustrates using quotation marks, both double and single, within character constants:

```
data titles;
  book1 = "Uncle Tom's Cabin"; /* 1 */
  book2 = 'Uncle Tom''s Cabin'; /* 2 */
  book3 = "Ben Hur"; /* 3 */
  book4 = '""Ben Hur""""'; /* 4 */
run;
proc print data=titles;
run;
```

1 book1 uses alternating double and single quotation marks.
2 book2 uses single quotation marks to escape the inner quotation character.
3 book3 uses single and double quotation marks.
4 book4 uses three double quotation marks.

Output 8.3  Results of Literal Constants Containing Apostrophes

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Uncle Tom's Cabin</td>
<td>Uncle Tom's Cabin</td>
<td>&quot;Ben Hur&quot;</td>
<td>&quot;Ben Hur&quot;</td>
</tr>
</tbody>
</table>

Key Ideas

- You can use one of two methods to escape quotation marks in character constants. You can use alternating double and single quotation marks, or you can double up on the quotation marks to escape the character.
- If a constant contains an apostrophe or a single quotation mark, then use double quotation marks around the entire constant value and use the single quotation mark inside the value. Alternatively, you can “escape” the single quotation mark by using another single quotation mark.

See Also

- “Character Constants and Character Variables”
- “Apostrophes and Quotation Marks within Character Constants”
- “Character Constants Expressed in Hexadecimal Notation”
- “Example: Define Character Constants in Hexadecimal Notation”
Example: Define Character Constants in Hexadecimal Notation

Example Code
The following example illustrates character constants in hexadecimal notation:

```sas
data _null_
   value1='534153'x;        /* 1 */
   value2='53,41,53'x;      /* 2 */
   put value1 "is identical to " value2;
run;
```

1. `value1` is defined by enclosing the constant string in single quotation marks, followed immediately by an `x`.
2. `value2` is defined by enclosing the constant string in single quotation marks and uses commas to make the string more readable, followed immediately by an `x`.

The following is printed to the SAS log:

**Example Code 8.1 SAS Log**

| SAS is identical to SAS |

Key Ideas
- SAS character constants can be expressed in hexadecimal notation, with each character represented by the hexadecimal notation.
- The character constant string is enclosed in single or double quotation marks, followed immediately by an `X`.
- Commas can be used to make the string more readable, but they are not part of and do not alter the hexadecimal value.

See Also
- "Character Constants Expressed in Hexadecimal Notation"
- "Numeric Constants Expressed in Hexadecimal Notation"
Example: Define Numeric Constants in Standard Notation

Example Code

The following example defines numeric constants expressed in standard notation. The numeric constants assigned to `num1`–`num5` have different representations. `num1` is an unsigned integer, `num2` contains a leading zero, `num3` and `num4` contain a plus sign, and `num5` represents the signed integer, negative 1.25. The minus sign (`-`) is used to represent negative integers or negative numbers.

```
data _null_;  
    num1=1;  
    num2=01;  
    num3+=1;  
    num4=+01;  
    put num1 "=" num2 "=" num3 "=" num4;  
    num5=-1.25;  
    put num5;  
run;
```

Note: Even if you specify a leading zero when creating your variables, leading zeros are dropped by default for numeric variables.

```
data _null_;  
    num1=1;  
    num2=01;  
    num3+=1;  
    num4=+01;  
    put num1 "=" num2 "=" num3 "=" num4;  
    num5=-1.25;  
    put num5;  
run;
```

The following is printed to the SAS log:

```
Example Code 8.2   SAS Log
```

```
1 = 1 = 1 = 1
-1.25
```

Key Ideas

- You can express a numeric constant in standard notation using plus and minus signs to indicate positive and negative numbers.
- You can also express numeric constants in scientific and hexadecimal notation.

See Also

- Numeric Constants Expressed in Standard Notation
- Numeric Constants Expressed in Scientific Notation
- Numeric Constants Expressed in Hexadecimal Notation
Example: Define Numeric Constants in Scientific Notation

Example Code

The following example defines numeric constants in scientific notation:

```sas
data _null;
  large1=1.2e23;
  med1=0.5e-10;
  put "large1=" large1;
  put "med1=" med1;
run;
```

The following is printed to the SAS log:

```
large1=1.2E23
med1=5E-11
```

Key Ideas

- In scientific notation, the number before the E is multiplied by the power of ten, which is indicated by the number after the E.
- For numeric constants that are larger than $(10^{32})^{-1}$, you must use scientific notation.
- Use E and an exponent after the value to signify scientific notation.
- You can also express numeric constants in standard and hexadecimal notation.

See Also

- "Numeric Constants Expressed in Standard Notation"
- "Numeric Constants Expressed in Scientific Notation"
- "Numeric Constants Expressed in Hexadecimal Notation"

Example: Define Numeric Constants in Hexadecimal Notation

Example Code

The following example defines numeric constants in hexadecimal notation:

```sas
data _null;
```

hex1=0c1x;
hex2=9x;
put "hex1=" hex1;
put "hex2=" hex2;
run;

The following is printed to the SAS log:

Example Code 8.3  SAS Log

<table>
<thead>
<tr>
<th>hex1</th>
<th>hex2</th>
</tr>
</thead>
<tbody>
<tr>
<td>193</td>
<td>9</td>
</tr>
</tbody>
</table>

Key Ideas

- A numeric constant that is expressed as a hexadecimal value starts with a numeric digit (usually 0), can be followed by more hexadecimal characters, and ends with the letter x.
- A numeric constant can contain up to 16 valid hexadecimal characters (0 to 9, A to F).
- You can express numeric constants in standard and scientific notation.

See Also

- “Numeric Constants Expressed in Standard Notation”
- “Numeric Constants Expressed in Scientific Notation”
- “Numeric Constants Expressed in Hexadecimal Notation”

Example: Define Date, Time, and Datetime Values in Date Constants

Example Code

The following example contains three illustrations of date, time, and datetime constants:

data dtconsts;
  date='1jan2018'd;        /*1*/
  time="9:25:19pm"t;      /*2*/
  datetime='18jan2018:9:27:05am'dt;  /*3*/
run;

proc print data=dtconsts; /*4*/
  format date date.
  time time.
  datetime datetime.;
run;
1 The variable `date` is expressed as a date constant, which has single quotation marks and is followed by a `d` signifying date.

2 The variable `time` is expressed as a time constant, which has double quotation marks and is followed by a `t` signifying time.

3 The variable `datetime` is expressed as a datetime constant, which has single quotation marks and is followed by a `dt` signifying datetime.

4 PROC PRINT uses the `FORMAT` statement to format the date, time, and datetime constants. Without the `FORMAT` statement the result is displayed as a number, which represents the SAS date, time, and datetime values.

*Output 8.4 Results of Date, Time, and DateTime Constants*

<table>
<thead>
<tr>
<th>Obs</th>
<th>date</th>
<th>time</th>
<th>datetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01JAN'18</td>
<td>21:25:19</td>
<td>18JAN'18 09:27:05</td>
</tr>
</tbody>
</table>

**Key Ideas**

- The constants are enclosed in single or double quotation marks.
- Use `d`, `t`, or `dt` after the value to signify the type of constant.
- Trailing blanks or leading blanks that are included within the quotation marks do not affect the processing of the date constant, time constant, or datetime constant.

**See Also**

- “Date, Time, and Datetime Constants”
- “Dates, Times, and Intervals”
- “FORMAT Statement” in SAS DATA Step Statements: Reference

**Example: Bit Test a Variable’s Value**

**Example Code**

The following example uses bit masks to do bit testing. The example tests whether bit 4 is 1 (TRUE) for the values 8 and 7.

```sas
data _null_;  
var=8;  
if var="1..."b then state="1";  
else state="0";  
put "For value " var " bit 4 is " state;  

var=7;  
if var="1..."b then state="1";  
else state="0";
```
put "For value " var "bit 4 is " state;
run;

The following is printed to the SAS log:

Example Code 8.4  SAS Log

<table>
<thead>
<tr>
<th>For value 8 bit 4 is 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>For value 7 bit 4 is 0</td>
</tr>
</tbody>
</table>

Key Ideas

- A bit testing constant is a bit mask that is used in bit testing to compare internal bits in a value's representation.
- You can perform bit testing on both character and numeric variables.
- Enclose the bit mask in quotation marks and use'b' to signify a bit mask.

See Also

- "Bit Testing Constants"
- $BINARYw. Format

Example: Avoiding a Common Error with Constants

Example Code

The following example illustrates a common error that you might get when you use a string in quotation marks. It is followed by a variable name, which does not have a space between the quoted string and the variable name. '821't is evaluated as a time constant because there is no space between the numeric value of 821 and the t in the THEN statement.

data aireuro;
   set sasuser.europe;
   if flight='821't then flight='230';
run;

The following errors are printed to the SAS log:

Example Code 8.5  SAS Log

ERROR: Invalid date/time/datetime constant '821't.
ERROR 77-185: Invalid number conversion on '821't.
ERROR 388-185: Expecting an arithmetic operator.
ERROR 202-322: The option or parameter is not recognized and will be ignored.

To correct this error, insert a blank space between the ending quotation mark and the t in the THEN statement. This eliminates the misinterpretation. No error
message is generated and all observations with a FLIGHT value of 821 are replaced with a value of 230.

    if flight='821' then flight='230';

Key Ideas

- Always insert a blank space between ending quotation marks and variable names to avoid errors.
- Without the blank space, SAS misinterprets a character constant followed by a letter as a special SAS constant.

See Also

- Avoiding a Common Error with Character Constants
Definitions for SAS Operators

A SAS operator is a symbol that is used to perform a comparison, arithmetic calculation, or logical operation. SAS uses two major types of operators:

- prefix operators
- infix operators

A prefix operator applies to the variable, constant, function, or parenthetical expression that the operator precedes. The plus sign (+) and the minus sign (−) can be used as prefix operators. The word NOT and its equivalent symbols can also be used as prefix operators. Here are examples of how to use prefix operators:
Variables:

\( x \)

Constants:

\( 25 \)

Functions:

\( \cos(\text{angle1}) \)

Parenthetical expressions:

\( (x \cdot y) \)

An infix operator is an operator that is placed between operands. Infix operators include the following types of operators:

- **Arithmetic:**
  
  \( a=b+c \)

- **Comparison:**
  
  Weight > 150

- **Logical or Boolean:**
  
  if \( x \) or \( y \) eq 1

- **Minimum and maximum:**
  
  where \( a=(b \max c) \)

- **Concatenation:**
  
  Name=Firstname ||| Lastname

SAS also provides several other operators that are used only with certain SAS statements. The WHERE statement uses a special group of SAS operators, valid only when used with WHERE expressions. For a discussion of these operators, see WHERE Statement.

### Arithmetic Operators

Arithmetic operators perform calculations, as shown in the following table.

**Table 9.1 Arithmetic Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Raise A to the third power.</td>
<td>( a^{**3} )</td>
</tr>
<tr>
<td>( \ast )</td>
<td>Multiply 2 by the value of ( Y ).</td>
<td>( 2 \ast y )</td>
</tr>
<tr>
<td>( / )</td>
<td>Divide the value of ( VAR ) by 5.</td>
<td>( \text{var} / 5 )</td>
</tr>
<tr>
<td>( + )</td>
<td>Add 3 to the value of ( NUM ).</td>
<td>( \text{num} + 3 )</td>
</tr>
</tbody>
</table>
Symbol | Description | Example
---|---|---
- | Subtract the value of DISCOUNT from the value of SALE. | sale-discount

1 The asterisk (*) is always necessary to indicate multiplication; 2Y and 2(Y) are not valid expressions.

See “Order of Operation in Compound Expressions” on page 198 for the order in which SAS evaluates these operators.

**Note:** When a value that is used with an arithmetic operator is missing, the result is a missing value. See “Missing Variable Values” on page 94 for information about how to prevent the propagation of missing values.

---

**Comparison Operators**

Comparison operators express a condition. If the comparison is true, the result is 1. If the comparison is false, the result is 0.

**Ways to Make Comparisons**

Comparison operators can be expressed as symbols or with their mnemonic equivalents, which are shown in the following table.

You can add a colon (:) modifier to any of the operators to compare only a specified prefix of a character string. See “Character Comparisons” in SAS Language Reference: Concepts for details.

**Table 9.2 Comparison Operators**

<table>
<thead>
<tr>
<th>Symbol/Mnemonic</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>= or EQ</td>
<td>equal to</td>
<td>a=3</td>
</tr>
<tr>
<td>^=, ^=, ^=, or NE ¹</td>
<td>not equal to</td>
<td>a ne 3</td>
</tr>
<tr>
<td>&gt; or GT</td>
<td>greater than</td>
<td>num&gt;5</td>
</tr>
<tr>
<td>&lt; or LT</td>
<td>less than</td>
<td>num&lt;8</td>
</tr>
<tr>
<td>&gt;= or GE²</td>
<td>greater than or equal to</td>
<td>sales&gt;=300</td>
</tr>
<tr>
<td>&lt;= or LE³</td>
<td>less than or equal to</td>
<td>sales&lt;=100</td>
</tr>
<tr>
<td>Symbol/Mnemonic</td>
<td>Definition</td>
<td>Example</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>or IN</td>
<td>equal to one of a list</td>
<td>num in (3, 4, 5)</td>
</tr>
</tbody>
</table>

1. The symbol that you use for NE depends on your operating environment.
2. The symbol => is also accepted for compatibility with previous releases of SAS. It is not supported in WHERE clauses or in PROC SQL.
3. The symbol <= is also accepted for compatibility with previous releases of SAS. It is not supported in WHERE clauses or in PROC SQL.

**Numeric Comparisons**

SAS makes numeric comparisons that are based on true and false values. The expression evaluates to 1 if the expression is true and the expression evaluates to 0 if the expression is false. For example, in the expression A<B, if A has the value 4 and B has the value 3, then A<B has the value 0, or false.

Numeric comparisons are commonly used in the following instances:

- IF-THEN/ELSE Statement
- Expression in an Assignment Statement

For numeric comparison examples, see the following:

- “Example: Subset Data Using a Comparison Operator”
- “Example: Create Variables Using Comparison Operators”

You might get an incorrect result when you compare numeric values of different lengths because values less than 8 bytes have less precision than those longer than 8 bytes. Rounding also affects the outcome of numeric comparisons. See “SAS Variables” in SAS Language Reference: Concepts for a complete discussion of numeric precision.

A missing numeric value is smaller than any other numeric value, and missing numeric values have their own sort order. See “Missing Variable Values” on page 94 for more information.

**Character Comparisons**

Character variable values are compared character by character from left to right. Character order depends on the collating sequence used by your computer, and your SAS session encoding option.

For example, in the Latin 1 (cp1252 West European) and UTF-8 (UTF-8 Unicode) collating sequences, G is greater than A. Therefore, this expression is true:

`'Gray'>'Adams'`

There are several important notes to keep in mind when making character comparisons:

- A blank and a period in character strings are smaller than any other printable character in the string. A blank is smaller than a period. For example, the following expressions are true:

  `'C.Jones'<'CharlesJones'`
  `'C Jones' < 'CJones'`
  `'C Jones' < 'C.Jones'`
Character values of unequal length are compared as if blanks were attached to the end of the shorter value before the comparison is made. For example, the following expression is true:

'ab'<'abc'

Trailing blanks are ignored in a comparison. For example, 'fox ' is equivalent to 'fox'.

A colon modifier after the comparison operator compares the quoted characters after the colon with value on the other side of the comparison operator. SAS truncates the longer value to the length of the shorter value during the comparison. For example, see “Example: Compare a Specified Prefix of a Character Expression” on page 208.

Character values are case sensitive.

IN Operator

The IN operator checks whether a value exists in a list and selects records that match the search. The value that is checked against the list can be the result of an expression. Individual values in the list can be separated by commas or spaces. You can use a colon to specify a range of sequential integers.

The three forms of the IN comparison are:

Form 1: Individual values can be separated by commas.

expression IN(value-1<...value-n>)

Form 2: Individual values can be separated by spaces.

expression IN(value-1<... value-n>)

Form 3: A range of values can be specified by placing a colon between values.

expression IN(value-1<...:value-n>)

The components of the comparison are as follows:

expression

- can be any valid SAS expression, but is usually a variable name when it is used with the IN operator.

value

- must be a constant.

For more information and examples of using the IN operator, see “The IN Operator in Numeric Comparisons” in SAS Language Reference: Concepts and “Example: Search an Array of Numeric Variable Values Using the IN Operator”.

Why Use the IN Operator?

The IN operator enables you to determine whether a variable's value is among a list of character or numeric values. You can use an array with the IN operator to search variable values, or you can search with a range or string.

The following table shows how to use the IN operator with numeric and character variables.
### Table 9.3 Use Cases for the IN Operator

<table>
<thead>
<tr>
<th>Use Cases</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Numeric Variables</strong></td>
<td>Specify a range of sequential integers to search. You can also use multiple ranges in the same IN list.</td>
</tr>
<tr>
<td></td>
<td>▪ y=x in(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);</td>
</tr>
<tr>
<td></td>
<td>▪ y=x in(1 2 3 4 5 6 7 8 9 10);</td>
</tr>
<tr>
<td></td>
<td>▪ y=x in(1:10);</td>
</tr>
<tr>
<td>Use both ranges and constants in the list of values to compare.</td>
<td>if x in {0,9,1:5};</td>
</tr>
<tr>
<td><strong>Character Variables</strong></td>
<td>Determine whether a variable’s value is among a list of character values.</td>
</tr>
<tr>
<td></td>
<td>▪ if state in ('NY','NJ','PA') then region+1;</td>
</tr>
<tr>
<td></td>
<td>▪ if state in ('NY' 'NJ' 'PA') then region+1;</td>
</tr>
<tr>
<td></td>
<td>▪ if state='NY' or state='NJ' or state='PA' then region+1; 1</td>
</tr>
<tr>
<td>Search an array of character values.</td>
<td>“Example: Search an Array of Character Variable Values Using the IN Operator”</td>
</tr>
<tr>
<td>1 These statements produce the same results.</td>
<td></td>
</tr>
<tr>
<td>2 The IN operator accepts character variable values of different lengths.</td>
<td></td>
</tr>
</tbody>
</table>

### Logical Operators

Logical operators, also called Boolean operators, are usually used in expressions to link a sequence of expressions into compound expressions.

### Ways to Use Logical Operators

The logical operators are shown in the following table.

A numeric expression without any logical operators can serve as a logical numeric expression.
Table 9.4 Logical Operators

<table>
<thead>
<tr>
<th>Symbol/Mnemonic</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; or AND</td>
<td>If both of the quantities linked by an AND are 1 (true), then the result of the AND operation is 1. Otherwise, the result is 0.</td>
<td>(a&gt;b &amp; c&gt;d)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>or OR¹</td>
<td>If either of the quantities linked by an OR is 1 (true), then the result of the OR operation is 1 (true). Otherwise, the OR operation produces a 0.</td>
</tr>
<tr>
<td></td>
<td>or NOT²</td>
<td></td>
</tr>
<tr>
<td>! or OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>; or OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬ or NOT²</td>
<td>not(a&gt;b)</td>
<td></td>
</tr>
<tr>
<td>¸ or NOT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬ or NOT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 The symbol that you use for OR depends on your operating environment.
2 The symbol that you use for NOT depends on your operating environment.

See "Order of Operation in Compound Expressions" for the order in which SAS evaluates these operators.

AND Operator

Two comparisons with a common variable linked by AND can be condensed with an implied AND. The following two subsetting IF statements produce the same result:

- if 16<=age and age<=65;
- if 16<=age<=65;

OR Operator

A comparison using the OR operator resolves as true if only one of the operands is true. Any nonzero, nonmissing constant is always evaluated as true. Therefore, in the list below, the first subsetting IF statement is always true and the second is not necessarily true:

- if x=1 or 2;
- if x=1 or x=2;
NOT Operator

The NOT operator is a prefix operator and a logical operator. The NOT operator inverts the truth of a statement value. The result of negating a false statement (0) is true (1). The result of negating a true statement (1) is false (0).

Comparisons that use the NOT operator can be written differently and yield the same results. The following two expressions are equivalent:

- `not(a=b & c>d)` is the same as `a ne b | c le d`
- `not(name='SMITH')` is the same as `name ne 'SMITH'

For example, see “Example: Use the NOT Operator to Reverse the Logic of a Comparison” on page 210.

Logical Numeric Expressions

In computing terms, a value of true is a 1 and a value of false is a 0. In SAS, any numeric value other than 0 or missing is true, and a value of 0 or missing is false. Therefore, a numeric variable or expression can stand alone in a condition. If its value is a number other than 0 or missing, the condition is true. If its value is 0 or missing, the condition is false.

For example, suppose that you want to assign values to the variable `Remarks` depending on whether the value of `Cost` is present for a given observation. You can write the IF-THEN statement as follows:

```
if cost then remarks='Ready to budget';
```

This statement is equivalent to the following:

```
if cost ne . and cost ne 0
    then remarks='Ready to budget';
```

A numeric expression can be a numeric constant, as follows:

```
if 5 then do;
```

The numeric value that is returned by a function is also a valid numeric expression:

```
if index(address,'Avenue') then do;
```

MIN and MAX Operators

Ways to Use MIN and MAX Operators Summary Table

The MIN and MAX operators are used to find the minimum or maximum value of two quantities. The MIN and MAX operators are commonly used in WHERE expressions and Subsetting IF Statement. Surround the operators with the two quantities whose minimum or maximum value you want to know.
Table 9.5  Ways to Use the MIN and MAX Operators

<table>
<thead>
<tr>
<th>Symbol/Mnemonic</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
</table>
| >< or MIN ¹      | Returns the lower of the two values. | where x = (b min c);
|                 |             | if x = (y><z); |
| <> or MAX ²      | Returns the higher of the two values. | where a=(b max c)
|                 |             | if x = (a<>b) |

¹ In a WHERE expression, the symbol representation >< is not supported. The MIN mnemonic is converted to >< in the LOG.
² In a WHERE expression, the symbol representation <> is interpreted as “not equal to”.

If missing values are part of the comparison, SAS uses the sorting order for missing values that is described in “Order of Missing Values” on page 95.

Comparison of MIN and MAX Operators and MIN and MAX Functions

The MIN and MAX operators should not be confused with the MIN and MAX functions. The MIN and MAX operators compare exactly two variables, but the MIN function and the MAX function compare any number of variables.

Concatenation Operators

The concatenation operator combines character values. It is indicated by the double vertical bar ||. The results of a concatenation operation are usually stored in a variable with an assignment statement, as in newname='Mr. or Ms. ' ||oldname; If the value of OldName is 'Jones', then NewName has the value 'Mr. or Ms. Jones'

The concatenation operator does not trim leading or trailing blanks. If variables are padded with trailing blanks, check the lengths of the variables and use the TRIM function to trim trailing blanks from values before concatenating them.

Concatenation Operator Summary Table

The concatenation operator is useful for combining character values. The following table demonstrates use cases for the concatenation operator:

Table 9.6  Use Cases for the Concatenation Operator

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate the value of a variable with a character constant.</td>
<td>newname='Mr. or Ms. '</td>
</tr>
</tbody>
</table>
Use Case | Example
---|---
Convert a numeric value to a character value using the PUT function. | `month='sep '; year=99; date=trim(month) || left(put(year,8.));` The value of DATE is the character value 'sep99'

Eliminate trailing blanks using the TRIM function in a concatenation operation. | Use the TRIM function in a Concatenation Operation to Eliminate Trailing Blanks on page 211

Comparison of the Concatenation Operator and the Concatenation Function

Concatenation functions perform concatenation operations without needing to use the TRIM and PUT functions. The concatenation functions include the following:

- **CAT Function**: same as the concatenation operator (||)
- **CATQ Function**: adds a delimiter and quotes to individual items
- **CATS Function**: removes leading and trailing blanks
- **CATT Function**: removes only trailing blanks
- **CATX Function**: removes leading and trailing blanks and inserts delimiters

Order of Operation in Compound Expressions

The order of operations is determined by the following conditions:

- In compound expressions, SAS evaluates the part of the expression containing priority 1 operators first, then each group in order.
- Consecutive operations that have the same priority are performed from right to left within priority 1 and from left to right within priority 2 and 3.
- Expressions within parentheses are evaluated before those outside of them.
- SAS does not guarantee the order in which subsequent expressions are evaluated. For more information, see “Short-Circuit Evaluation in SAS”.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Order of Evaluation</th>
<th>Symbol/ Mnemonic</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
</table>
| Group 1 | right to left       | **              | exponentiation | `y=a**2;`  
`x=2**3**4` is evaluated as `x=(2**3)**4` |
<p>|         |                     | +               | positive prefix | <code>y=+(a*b);</code> |
|         |                     | -               | negative prefix | <code>z=-(a+b);</code> |</p>
<table>
<thead>
<tr>
<th>Priority</th>
<th>Order of Evaluation</th>
<th>Symbol/ Mnemonic</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>++ ¬ or NOT</td>
<td>logical not&lt;sup&gt;3&lt;/sup&gt;</td>
<td>if not z then put x;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;&gt; or MIN</td>
<td>minimum</td>
<td>x=(a&gt;&gt;b); -3&gt;&gt;-3 is evaluated as -(3&gt;&gt;-3). These are equal to -(-3), which equals +3.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&gt; or MAX</td>
<td>maximum</td>
<td>x=(a&lt;&gt;b);</td>
<td></td>
</tr>
<tr>
<td>Group 2</td>
<td>left to right</td>
<td>*</td>
<td>c=a*b;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
<td>f=g/h;</td>
<td></td>
</tr>
<tr>
<td>Group 3</td>
<td>left to right</td>
<td>+</td>
<td>c=a+b;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtraction</td>
<td>f=g-h;</td>
<td></td>
</tr>
<tr>
<td>Group 4</td>
<td>left to right</td>
<td></td>
<td></td>
<td>!!</td>
</tr>
<tr>
<td>Group 5&lt;sup&gt;5&lt;/sup&gt;</td>
<td>left to right&lt;sup&gt;6&lt;/sup&gt;</td>
<td>&lt; or LT</td>
<td>less than if x&lt;y then c=5;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;= or LE</td>
<td>less than or equal to if x le y then a=0;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>= or EQ</td>
<td>equal to        if y eq (x+a) then output;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>¬= or NE</td>
<td>not equal to    if x ne z then output;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt;= or GE</td>
<td>greater than or equal to if y&gt;=a then output;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>&gt; or GT</td>
<td>greater than    if z&gt;a then output;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IN</td>
<td>equal to one of a list if state in ('NY', 'NJ', 'PA') then region='NE';</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group 6</td>
<td>left to right</td>
<td>&amp; or AND</td>
<td>logical and if a=b &amp; c=d then x=1;</td>
<td></td>
</tr>
</tbody>
</table>
### Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>Order of Evaluation</th>
<th>Symbol/ Mnemonic</th>
<th>Action</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 7</td>
<td>left to right</td>
<td></td>
<td>logical or</td>
<td>if y=2 or x=3 then a=d;</td>
</tr>
</tbody>
</table>

1. The plus (+) sign can be either a prefix or arithmetic operator. A plus sign is a prefix operation only when it appears at the beginning of an expression or when it is immediately preceded by an open parenthesis or another operator.

2. The minus (−) sign can be either a prefix or arithmetic operator. A minus sign is a prefix operator only when it appears at the beginning of an expression or when it is immediately preceded by an open parenthesis or another operator.

3. Depending on the characters available on your keyboard, the symbol can be the not sign (¬), tilde (~), or caret (^). The SAS system option CHARCODE allows various other substitutions for unavailable special characters.

4. Depending on the characters available on your keyboard, the symbol that you use as the concatenation operator can be a double vertical bar (||), broken vertical bar (¦), or exclamation mark (!!).

5. Group 5 operators are comparison operators. The result of a comparison operation is 1 if the comparison is true and 0 if it is false. Missing values are the lowest in any comparison operation. The symbols =< (less than or equal to) are also allowed for compatibility with previous versions of SAS.

6. An exception to this rule occurs when two comparison operators surround a quantity. For example, the expression x<y<z is evaluated as (x<y) and (y<z).

7. Depending on the characters available on your keyboard, the symbol that you use for the logical or can be a single vertical bar (|), broken vertical bar (¦), or exclamation mark (!). You can also use the mnemonic equivalent OR.

---

**Note:** When a value that is used with an arithmetic operator is missing, the result is a missing value. See “Missing Variable Values” on page 94 for information about how to prevent the propagation of missing values.

---

### Short-Circuit Evaluation in SAS

Minimal evaluation, or short-circuit evaluation, is a method that is used by some programming languages to evaluate Boolean operators. In short-circuit evaluation, the second argument in a Boolean expression is executed only if the first argument does not determine the value of the overall expression. For example, in the expression \((X \text{ AND } Y)\), if the first argument \(X\) evaluates to FALSE, then the overall expression \((X \text{ AND } Y)\) must evaluate to FALSE. So there is no need to calculate the second argument \(Y\).

SAS does not guarantee short-circuit evaluation. When using Boolean operators to join expressions, you might get undesired results if your intention is to short circuit, or to avoid the evaluation of the second expression. To guarantee the order in which SAS evaluates an expression, you can rewrite the expression using nested IF statements. The following examples show how SAS might use short-circuit evaluation at some times and not at others. The final example shows how you can use nested IF statements to guarantee the order of evaluation.

In the first example below, SAS uses short-circuit evaluation when it evaluates the first argument of the condition \(a>0\). The expression evaluates to FALSE, and as a result, SAS does not evaluate the second expression \(a=1/a\), which contains an invalid, division-by-zero operation. Since SAS does not evaluate this second expression, the program does not return the error.

```sas
data test;
  a=0;
  if (a>0 AND a=1/a) then put 'hello';
  else put 'goodbye';
run;
```
In the next example, short-circuit evaluation is not used. Even though the first argument in the condition, \( a \), evaluates to FALSE, the second argument, \( a=1/a \), is evaluated and a division-by-zero error is returned.

```sas
data test;
a=0;
  if (a AND a=1/a) then put 'hello';
  else put 'goodbye';
run;
```

NOTE: Division by zero detected at line 12 column 19.

goodbye

```
a=0 _ERROR_=1 _N_=1
```

NOTE: Mathematical operations could not be performed at the following places. The results of the operations have been set to missing values.

To guarantee the order in which SAS evaluates an expression such as this one, you can rewrite the expression using nested IF statements.

```sas
data test;
a=0;
  if a>0 then
    if a=1/a then put 'hello';
    else put 'goodbye';
  else put 'goodbye again';
run;
```

NOTE: Division by zero detected at line 12 column 19.

goodbye again

```
a=0 _ERROR_=1 _N_=1
```

NOTE: Mathematical operations could not be performed at the following places. The results of the operations have been set to missing values.

---

### Summary of Ways to Use Operators

#### Summary of Ways to Use Operators Tables

*Table 9.8*  Summary of Ways to Use SAS Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>Raise A to the third power.</td>
<td>a**3</td>
</tr>
<tr>
<td>*</td>
<td>Multiply 2 by the value of Y.</td>
<td>2*y</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>Example</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>/</td>
<td>Divide the value of VAR by 5.</td>
<td>var/5</td>
</tr>
<tr>
<td>+</td>
<td>Add 3 to the value of NUM.</td>
<td>num+3</td>
</tr>
<tr>
<td>-</td>
<td>Subtract the value of DISCOUNT from the value of SALE.</td>
<td>sale-discount</td>
</tr>
</tbody>
</table>

**Comparison Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>= or EQ</td>
<td>equal to</td>
<td>a=3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Jones’ EQ ‘Jones’</td>
</tr>
<tr>
<td>^=, ¬=, ~=, or NE</td>
<td>not equal to</td>
<td>a ne 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘Jones’ NE ‘JONES’</td>
</tr>
<tr>
<td>&gt; or GT</td>
<td>greater than</td>
<td>num&gt;5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>‘CharlesJones’&gt;’C.Jones’</td>
</tr>
<tr>
<td>&lt; or LT</td>
<td>less than</td>
<td>num&lt;8</td>
</tr>
<tr>
<td>&gt;= or GE</td>
<td>greater than or equal to</td>
<td>sales&gt;=300</td>
</tr>
<tr>
<td>&lt;= or LE</td>
<td>less than or equal to</td>
<td>sales&lt;=100</td>
</tr>
<tr>
<td>IN</td>
<td>equal to one of a list</td>
<td>num in (3, 4, 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>state in (‘NY’,’NJ’,’PA’)</td>
</tr>
</tbody>
</table>

**Logical Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp; or AND</td>
<td>If both of the quantities linked by an AND are 1 (true), then the result of the AND operation is 1. Otherwise, the result is 0.</td>
<td>(a&gt;b &amp; c&gt;d)</td>
</tr>
<tr>
<td></td>
<td>or OR</td>
<td>If either of the quantities linked by an OR is 1 (true), then the result of the OR operation is 1 (true). Otherwise, the OR operation produces a 0.</td>
</tr>
<tr>
<td>! or OR</td>
<td></td>
<td>not(a&gt;b)</td>
</tr>
<tr>
<td>; or OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬ or NOT</td>
<td></td>
<td>not(a&gt;b)</td>
</tr>
<tr>
<td>° or NOT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Examples: Operators

Example: Subset Data Using a Comparison Operator

Example Code

This example demonstrates how comparison operators can be used in an IF-THEN/ELSE Statement to subset data.

```sas
data greencars;
  set sashelp.cars;
  MPG_AVG=(MPG_City + MPG_Highway)/2;
  if MPG_AVG>30 then Green_Rating=1; /* 1 */
  if MPG_AVG<30 and MPG_AVG>=25 then Green_Rating=2; /* 2 */
  else if MPG_AVG<=25 then delete; /* 3 */
proc print data=greencars;
  var Make Model MPG_City MPG_Highway MPG_AVG Green_Rating;
run;
```

1 Observations with an MPG_AVG value that is greater than 30 are given a Green_Rating of 1.
2 Observations with an MPG_AVG value that is less than 30 and greater than or equal to 25 are given a Green_Rating of 2.
3 Observations with an MPG_AVG value that is less than or equal to 25 are deleted.
Key Ideas

- Comparison operators express a condition.
- A numeric variable or expression can stand alone in a condition.
- You can use arithmetic operators in an Assignment Statement when creating new variables.
- Numeric comparisons yield a 1 if the result is true and a 0 if the result is false.

See Also

- “Summary of Ways to Use Operators Tables” on page 201
- You can use comparisons in expressions in Assignment statements. For example, see “Example: Create Variables Using Comparison Operators” on page 204.
- Missing numeric values have their own sort order. For more information, see “Missing Variable Values” on page 94.

Example: Create Variables Using Comparison Operators

Example Code

The following example shows how to use comparisons in the Assignment Statement to create a variable.

```r
data test;         /*1*/
x=6;               /*2*/
y=8;               /*3*/
c=5*(x<y)+12*(x>=y);   /*4*/
put c;              /*5*/
```
Assign values to the variables x and y so that x equals 6 and y equals 8.

Compare the values of x and y in an Assignment statement. Because the value for x (6) is less than the value for y (8), the expression (x<y) evaluates to 1 (true). Likewise, because 6 is not greater than or equal to 8, the expression (x>=y) evaluates to 0 (false). Therefore, c=(5*1)+(12*0)=5.

Print the value for c in the log.

```
data test;
  x=6;
  y=8;
  c=5*(x<y)+12*(x>=y);
  put c;
run;
```

Key Ideas

- Comparison operators express a condition.
- Comparisons yield a 1 if the result is true and a 0 if the result is false.
- SAS evaluates quantities inside parentheses before performing any operations.

See Also

- "Summary of Ways to Use Operators Tables" on page 201

Example: Search an Array of Numeric Variable Values Using the IN Operator

Example Code

This example shows how you can use the IN operator to search an array of numeric values. The following code creates an array a, defines a constant x, and uses the IN operator to search for the value of x in the array.

```
data test;
  array a{10} (2*1:5); /* 1 */
  x=99;               /* 2 */
  y= x in a;          /* 3 */
  put y=;
run;
```

Create an array named a that contains the following 10 values: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5.

Create a variable, x, and assign it the value 99.
The `x in a` part of the statement searches the array named `a` for the value 99. The variable `y` evaluates to 0, or false, because 99 does not exist in the array.

The output is below.

<table>
<thead>
<tr>
<th>Obs</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>a5</th>
<th>a6</th>
<th>a7</th>
<th>a8</th>
<th>a9</th>
<th>a10</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>99</td>
<td>0</td>
</tr>
</tbody>
</table>

The code above shows how to create the array and assign values. The code below shows how to assign values once you add `a{5}=98`.

```sas
data test;
    array a{10} (2*1:5);
    x=99;
    a{5} = 99;            /*1*/
    y = x in a;
    put y=;
run;
proc print data=test;
run;
```

The Assignment statement assigns the value 99 to the fifth element in the array, overwriting the original value of 5 that was assigned in the ARRAY statement.

The output is below.

<table>
<thead>
<tr>
<th>Obs</th>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>a5</th>
<th>a6</th>
<th>a7</th>
<th>a8</th>
<th>a9</th>
<th>a10</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>99</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>99</td>
<td>1</td>
</tr>
</tbody>
</table>

376 data test;
377    array a{10} (2*1:5);
378    x=99;
379    a{5} = 99;
380    y = x in a;
381    put y=;
382    run;

Key Ideas

- You can assign values to the variables in an array when you create it by specifying the values in parenthesis.
- PROC SQL does not support the `IN` operator.

See Also

- "IN Operator" on page 193
Example: Search an Array of Character Variable Values Using the IN Operator

Example Code

In this example, the array, a, defines the constant, x, and then uses the IN operator to search for x in the array.

```sas
data _null_
   array a{5} $ (5*'');
   x='b1';
   y = x in a;
   put y=;
   a{5} = 'b1';
   y = x in a;
   put y=;
run;
```

Example Code 9.1 Results from Using the IN Operator to Search an Array of Character Values (Partial Output)

```
data _null_
   array a{5} $ (5*'');
   x='b1';
   y = x in a;
   put y=;
   a{5} = 'b1';
   y = x in a;
   put y=;
run;
y=0
y=1
```

Key Ideas

- You can also use the IN operator to search an array of numeric values.

See Also

- “Why Use the IN Operator?” on page 193
Example: Compare a Specified Prefix of a Character Expression

Example Code

This example shows how you can use a colon (:) after the comparison operator to compare only a specified prefix of a character expression.

data restaurantratings;
length Action $10;
input Name $1-18 Eval_1 20-21 Eval_2 Eval_3 Status $;
  if Status='F' then Action='Contact';
  if Status='P' then Action='Print Card';
datalines;
Lilac and Lavender 97 95 99 Passed
The Salty Pearl 85 70 65 F
Taste the Range 90 92 90 Pass
The Underground 85 90 90 P
When Pigs Fly 70 70 67 Fail
Basil 85 90 90 Passed
;
proc print data=restaurantratings;
run;

Here is the output:

<table>
<thead>
<tr>
<th>Obs</th>
<th>Action</th>
<th>Name</th>
<th>Eval_1</th>
<th>Eval_2</th>
<th>Eval_3</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Print Card</td>
<td>Lilac and Lavender</td>
<td>97</td>
<td>95</td>
<td>99</td>
<td>Passed</td>
</tr>
<tr>
<td>2</td>
<td>Contact</td>
<td>The Salty Pearl</td>
<td>85</td>
<td>70</td>
<td>65</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td>Print Card</td>
<td>Taste the Range</td>
<td>90</td>
<td>92</td>
<td>90</td>
<td>Pass</td>
</tr>
<tr>
<td>4</td>
<td>Print Card</td>
<td>The Underground</td>
<td>85</td>
<td>90</td>
<td>90</td>
<td>P</td>
</tr>
<tr>
<td>5</td>
<td>Contact</td>
<td>When Pigs Fly</td>
<td>70</td>
<td>70</td>
<td>67</td>
<td>Fail</td>
</tr>
<tr>
<td>6</td>
<td>Print Card</td>
<td>Basil</td>
<td>85</td>
<td>90</td>
<td>90</td>
<td>Passed</td>
</tr>
</tbody>
</table>

Key Ideas

- In this example, the colon modifier after the equal sign tells SAS to select only the first character. But it has the capability to compare as many characters as are placed in quotation marks after the colon modifier.
- SAS truncates the longer value to the length of the shorter value during the comparison.
- If you compare a zero-length character value with any other character value in either an IN: comparison or an EQ: comparison, the two-character values are not considered equal. The result always evaluates to 0, or false.
Example: Compare Variables Using Boolean Operators

Example Code

This example shows how you can use Boolean operators to compare variables.

```sas
data inventory;
  length Restock $ 3;
  input ItemNum 1-4 Quantity 5-7 PriorityLev 8-10;
  if Quantity<=10 and PriorityLev>=5
    or Quantity<=10 then Restock=1;
  else Restock=0;
  datalines;
  6530 10 8
  8759 3 1
  4573 22 10
  9237 2 10
  4329 12 4
  9831 9 7
  9830 15 4
  9458 25 1
  5673 4 10
  7562 7 3
  3291 3 10
;
proc print data=inventory;
run;
```

<table>
<thead>
<tr>
<th>Obs</th>
<th>Restock</th>
<th>ItemNum</th>
<th>Quantity</th>
<th>PriorityLev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6530</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>8759</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>4573</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>9237</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>4329</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>9831</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>9830</td>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>9458</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>5673</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>7562</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>3291</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>
Key Ideas

- If both of the quantities linked by AND are 1 (true), then the result of the AND operation is 1. Otherwise, the result is 0.
- If either of the quantities linked by an OR is 1 (true), then the result of the OR operation is 1 (true). Otherwise, the OR operation produces a 0.

See Also


Example: Use the NOT Operator to Reverse the Logic of a Comparison

Example Code

This example shows how you can use the NOT operator with the IN operator to reverse the logic of a comparison.

```sas
data productreviews;
   length Category $ 10;
   input ProdID Satisfaction Quality Safety Usability;
   Avg= round((Satisfaction+Quality+Safety+Usability)/4,0.1);
   if (avg>=3) then Category='Pass'; /*1*/
   else if (avg) NOT IN (3,4,5) then Category='Fail';
datalines;
8954 5 3 2 5
9183 5 5 5 5
6839 1 1 1 1
3493 2 1 3 2
2908 3 2 3 3
5419 5 4 5 5
3759 3 4 3 3
5301 4 3 4 4
; run;
proc print data=productreviews;
   var ProdID avg Category;
run;
```

1 The variable category depends on the average of the values of satisfaction, quality, safety, and usability.
Key Ideas

- You can use the NOT operator with other operators to reverse the logic of the comparison.

See Also

- “NOT Operator” on page 196

Example: Remove Trailing Blanks Using the TRIM Function in a Concatenation Operation

Example Code

In this example, the TRIM function is used with the concatenation operator to remove the trailing blanks that are visible after the concatenation of the variables color and name.

```
data namegame;
    length color name $8 game $12;
    color='black';          /* 1 */
    name='jack';
    game=trim(color)||name; /* 2 */
    put game=;
run;
```

1. The length of the color variable is eight, so the value black has 3 trailing blanks.
2. The value of game is black jack. The TRIM function removes the spacing.

Key Ideas

- The concatenation operator does not trim leading or trailing blanks.
- You can use the CAT Functions to perform concatenation operations without needing to use the TRIM and PUT functions.
See Also

- SAS Functions and CALL Routines: Reference
Dates and Times

Dates, Times, and Intervals

SAS provides date, time, and datetime intervals for counting different periods of elapsed time. For more information, see “About SAS Date, Time, and Datetime Values” in SAS Language Reference: Concepts.
Component Objects

DATA Step Component Objects

A DATA step component object is an element in Base SAS that can be created and manipulated by using statements, attributes, methods, and operators in the DATA step.

SAS provides five predefined component objects for use in a DATA step: hash, hash iterator, logger, appender, and Java objects.

For more information, see SAS Component Objects: Reference
Definitions for SAS Libraries

A SAS library is a collection of one or more SAS files, including SAS data sets, that are referenced and stored as a unit. In directory-based environments, a SAS library is a group of SAS files that are stored in the same folder or directory.

Each SAS library is associated with a libref, an engine, a physical location, and options that are specific to the engine or environment.

The libref is a shortcut name or a nickname that you can use to reference the physical location of the data. For example, in the two-level name `mylib.myfile`, the libref is `mylib`, and the library member is `myfile`.

Library members can be any of the SAS file types in the following table. Other files can be stored in the directory, but only SAS files are recognized as part of the SAS library.

**Table 12.1 Most Common SAS Library Members**

<table>
<thead>
<tr>
<th>File</th>
<th>Member Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>data set or table</td>
<td>DATA</td>
<td>- “Example: Create and Print a V9 Engine Data Set” in SAS V9 LIBNAME Engine: Reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “Example: Access DBMS Data as a SAS Library”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “Example: Read and Write SAS Data in Hadoop by Using the SPD Engine”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- “Using an XMLMap to Import an XML Document as One SAS Data Set” in SAS XMLV2 and XML LIBNAME Engines: User’s Guide</td>
</tr>
</tbody>
</table>
In output from the DATASETS procedure that lists the contents of a library, you might notice that additional files are listed below a data set and have a different member type. These files are attributes of the data set and are stored in separate files. These member types are associated with a data set and cannot be specified in a MEMTYPE= option.

### Table 12.2 Data Set Attributes That Are Stored in Separate Files

<table>
<thead>
<tr>
<th>SAS File</th>
<th>Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>audit trail</td>
<td>AUDIT</td>
</tr>
<tr>
<td>extended attributes</td>
<td>XATTR</td>
</tr>
<tr>
<td>index</td>
<td>INDEX</td>
</tr>
</tbody>
</table>

Each SAS member type has a distinctive file extension. Therefore, a library can contain files with the same name but with different member types. File extensions vary, depending on the operating environment:
Elements of a Library Assignment

Introduction to Assigning Libraries

The following table summarizes common ways to assign a library. The assignment is valid only for the current SAS session unless you choose a method of persisting the assignment.

Table 12.3  Ways to Assign and Persist a SAS Library

<table>
<thead>
<tr>
<th>Method</th>
<th>Persistence Beyond the Current Session</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIBNAME statement (example)</td>
<td>Place the LIBNAME statement in an environment variable, the SAS Registry, an autoexec file, or the configuration file.</td>
</tr>
<tr>
<td>LIBNAME function (example)</td>
<td>New Library window</td>
</tr>
<tr>
<td></td>
<td>Check the <strong>Enable at start-up</strong> box in the SAS windowing environment or the <strong>Re-create this library at start-up</strong> box in SAS Studio.</td>
</tr>
<tr>
<td>Administrative interface such as SAS Management Console</td>
<td>A SAS administrator pre-assigns and persists the library.</td>
</tr>
<tr>
<td>Operating environment commands, such as JCL statements on z/OS</td>
<td>See information about deallocation in “Assigning SAS Libraries Externally” in <strong>SAS Companion for z/OS</strong>.</td>
</tr>
</tbody>
</table>

Library Name (Libref)

Rules for Naming a Libref

The libref is a required element in a library assignment:

```
libname libref engine 'location' options;
```
The **libref** is a shortcut name or a nickname that you can use to reference the physical location of the data. For example, in the two-level name `mylib.myfile`, the libref is `mylib`, and the library member is `myfile`.

Here are the rules for libref names:

- A libref can be no longer than eight bytes.
- The first character must be an English letter or underscore. Subsequent characters can be letters, numbers, or underscores. For details, see “Rules for Most SAS Names” on page 42.
- Do not use the reserved names Sashelp, Sasuser, or Work as librefs, except as intended. (See “SAS Default Libraries” on page 227.) Additional names are reserved for SAS under every operating environment, and should not be used for librefs:
  - “Librefs Assigned by SAS in UNIX Environments” in SAS Companion for UNIX Environments
  - “Assigning SAS Libraries Using the LIBNAME Statement or Function” in SAS Companion for Windows
  - “Reserved z/OS Ddnames” in SAS Companion for z/OS

**Rules for Using a Libref**

Here are some rules for using librefs:

- Specify the libref as the first element in a two-level name for a SAS file.
- A libref is valid only for the current SAS session unless you choose a method of persisting the library assignment.
- You can deassign (clear) a libref before the session ends.
- When a libref is deassigned or the session ends, the library members are not deleted from the physical storage location. (However, contents of the Work library are deleted when the session ends.)
- You can reference a libref repeatedly within a SAS session.
- SAS does not limit the number of librefs that you can assign during a session. However, your operating environment or administrator might set limitations.
- If you use a macro variable as a libref name, do not enclose the variable in quotation marks. Place a delimiter after the variable. For example, the two-level name `&mylib..mydata` references the `&mylib` variable to find the libref for the library where the `mydata` file is located. See “Creating a Period to Follow Resolved Text” in SAS Macro Language: Reference.

**Library Engine**

The engine name might not be required in a library assignment, but specifying the engine is a best practice:

```
libname libref engine 'location' options;
```

SAS provides a number of engines to access SAS files or files formatted by another application or DBMS. The shipped default Base SAS engine is BASE. In SAS®9 and
SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice. For more information, see Chapter 13, “SAS Engines,” on page 257. See also “Example: Set a Default Engine” on page 274.

Library Location

The physical location is required in a library assignment:

```sas
libname libref engine 'location' options;
```

Specify the path name of the physical location where you want to create or access data. If you specify a relative path name, it references your current working directory, which depends on the SAS interface and your deployment.

Enclose the physical location in single or double quotation marks. If you are concatenating libraries, the syntax for library location is slightly different. If the library’s physical location does not exist before you submit the LIBNAME statement, SAS writes a note to the log. In some cases, if you create the location after you submit such a LIBNAME statement, you can then successfully use the libref. However, in many processing modes and interfaces, you must resubmit the LIBNAME statement after you create the physical location.

You can set the DLCREATEDIR system option to automatically create a new subdirectory for the library.

Library Options

Library options might be required in a library assignment, depending on the engine and environment:

```sas
libname libref engine 'location' options;
```

When you access data that is stored in a DBMS or application other than SAS, usually you must specify connection information. You might need additional LIBNAME statement options that are specific to the engine.

In addition, some SAS data set options are also available as LIBNAME statement options. Note that LIBNAME statement options take precedence over system options. Data set options take precedence over LIBNAME statement options.

For LIBNAME statement options that are specific to an operating environment, see the SAS companions:

- “LIBNAME Statement: UNIX” in SAS Companion for UNIX Environments
- “LIBNAME Statement: Windows” in SAS Companion for Windows
- “LIBNAME Statement: z/OS” in SAS Companion for z/OS

For LIBNAME statement options that are specific to a SAS engine, see documents such as these:
Benefits of Using a Libref

Accessing data through a SAS library provides the following benefits:

- Specifying a libref in your SAS code is more convenient than specifying the path name, especially if you reference the file repeatedly in a program.
- You can specify options for an entire library. Accessing data that is stored in a DBMS or application other than SAS usually requires multiple options.
- If your data changes location, you can modify its path name once, in the library assignment, instead of multiple places in your code.
- Related files can be grouped in the same physical location.
- In addition to user interfaces, SAS provides programmatic ways to assign a library, to view information about a library, and to manage a library.

Accessing Data without Using a Libref

In some cases, a library assignment is not required:

- If you omit a libref and specify a one-level name, SAS uses the temporary Work library, which is typically deleted when the session ends. You can change this behavior. To set a default library to permanently store SAS files, assign a User library. If a User library is assigned, it takes precedence over the Work library.
- If you specify the file name only (no path name) in quotation marks, the default location is the current working directory. This location depends on the SAS interface and your deployment.
- In most language elements, you can omit a libref and refer to a file directly by specifying the physical location.
- For external unstructured data, such as raw data that is stored in a text file, you use a fileref instead of a libref.

Here are the rules for specifying a physical location instead of a libref:

- Enclose the path name and file name in quotation marks. You can omit the file extension if the file is a SAS data set.
- The quoted path name and file name must conform to the naming conventions of your operating environment.
- Specifying a libref is preferred to specifying the physical location, due to the benefits of library assignment.
- A quoted physical location is not supported for the following SAS features:
- engines other than the default Base SAS engine (V9)
- contexts that do not accept a libref, such as the SELECT statement of PROC COPY and most PROC DATASETS statements
- PROC SQL
- certain data set options
- SAS views
- stored compiled DATA step programs
- SAS catalogs
- MDDB and FDB references

Library Concatenation

Definition of SAS Library Concatenation

When you concatenate two or more SAS libraries, you logically join the contents of the libraries. Although the libraries are stored in different locations, you can reference them all with one libref.

How to Concatenate Libraries

In the LIBNAME statement or LIBNAME function, specify each library with its previously assigned libref or its physical location (full path name).
- You can use a combination of librefs and physical locations.
- If you specify a physical location, enclose it in single or double quotation marks.
- Separate each specification with either a blank or a comma.
- Enclose the entire list in parentheses.

Here is an example that concatenates three libraries. The specification includes two librefs and one physical location.

    libname year (quarter1 quarter2 'c:/quarter3/sales');

For a more detailed example, see "Example: Concatenate SAS Libraries" on page 232.

Rules for Library Concatenation

After you create a library concatenation, you can specify the libref in any context that accepts a libref. SAS applies the following rules:
If you specify any options or engines, they apply only to the libraries that you specified with the physical location, not to any library that you specified with a libref.

If you alter a libref after it has been assigned in a concatenation, it does not affect the concatenation.

When you logically concatenate two or more SAS libraries, you also concatenate any SAS catalogs. See “Catalog Concatenation” in SAS V9 LIBNAME Engine: Reference.

If any library in the concatenation is sequential, then all of the libraries are treated as sequential.

The order in which you list the libraries determines how SAS files are processed:

- When a data set is opened for input or update, the concatenated libraries are searched in the order in which they are listed in the concatenation. The first occurrence of the data set is used.
- When a data set is created, it is created in the first library that is listed in the concatenation, even if a file exists with the same name in another library in the concatenation.
- When you delete or rename a SAS file, only the first occurrence of the file is affected.
- When a list of SAS files is displayed, only the first occurrence of a file name is shown.
- A SAS file that is logically associated with another file (such as an index to a data set) is included in the concatenation only if the parent file resides in that same library. This rule is affected by the rule that only the first occurrence of a file is used. For example, two libraries are concatenated. Both libraries contain a data set that is named Mytable. In the second library, Mytable is indexed. The index is not included in the concatenation.
- The attributes of the first library determine the attributes of the concatenation. For example, if the first library is Read-Only, then the entire concatenated library is Read-Only.

SAS Default Libraries

Introduction to SAS Default Libraries

The following libraries are supplied by SAS:

- Work is assigned by default.
- User is assigned by the user or by an administrator.
- Sashelp is assigned by default.
- Sasuser is assigned by default.
Work Library (Temporary)

SAS assigns the Work library automatically. The Work library stores two types of temporary files:

- files that you create
- files that are created internally by SAS as part of processing

The Work library enables you to specify one-level names for temporary storage. Specifying a **one-level name** means that you omit a libref and specify the file name only, without quotation marks. See “Example: Use the Work Library for Temporary Data” on page 241.

By default, the Work library is deleted at the end of each SAS session if the session terminates normally. To change the default behavior for the Work library, see the WORK=, WORKINIT, and WORKTERM system options in *SAS System Options: Reference*.

In contrast to the Work library, most SAS libraries are permanent, not temporary.

User Library

The User library enables you to specify one-level names for permanent storage. If you assign a User library, then all files that are created with a one-level name are stored in the User library instead of the temporary Work library. When you refer to a file by a one-level name, SAS looks for the file in the User library. If you have not assigned a User library, then SAS looks for the file in the Work library.

You can assign the User library by using one of the methods in the User library example on page 242. See also “USER= System Option” in *SAS System Options: Reference*.

Files that are stored in the User library are not deleted by SAS when the session terminates. Data files that SAS creates internally are still stored in the Work library. After you assign the User library, if you want to create a temporary file in Work, you must specify Work in a two-level name, such as Work.MyFile.

For the SPD Engine, you can use the TEMP=YES LIBNAME statement option with the USER= system option to store temporary data sets that can be referenced with a one-level name. See “TEMP= LIBNAME Statement Option” in *SAS Scalable Performance Data Engine: Reference*.

Sashelp Library

SAS assigns the Sashelp library automatically. The Sashelp library contains the following items:

- sample data that is used for some examples in SAS documentation.
- catalogs, item stores, and other files that store SAS settings for your entire site.

The defaults in the Sashelp library can be customized by your on-site SAS
support personnel. (Many settings are stored in the SAS Registry, which is two item stores. See Chapter 35, “The SAS Registry,” on page 671.)

Use PROC DATASETS or PROC CATALOG to list the catalogs in a library. The SASHELP system option specifies the location of the Sashelp library. The option is set during the installation process and normally is not changed after installation.

### Sasuser Library

SAS assigns the Sasuser library automatically. The Sasuser library contains catalogs and other files that store your personal settings and customizations. For example, in Base SAS, you can store your defaults for function key settings or window attributes. These values are stored in a catalog named Sasuser.Profile. See “Sasuser.Profile Catalog” in SAS V9 LIBNAME Engine: Reference. (Many settings are stored in the SAS Registry, which is two item stores. See Chapter 35, “The SAS Registry,” on page 671.)

The SASUSER system option specifies the location of the Sasuser library.

The RSASUSER system option enables the system administrator to control the mode of access to the Sasuser library. RSASUSER is helpful for installations that have one Sasuser library for multiple users, to prevent those users from modifying it.

### Examples: Access Data by Using a Libref

#### Example: Assign a Libref by Using the LIBNAME Statement

**Example Code**

This example assigns a libref and then references it in a DATA step and a PROC step.

```sas
libname sales 'c:\myfiles'; /*1*/
data sales.quarter1; /*2*/
    length mileage 4;
    input account mileage;
datalines;
1 932
2 563
; proc print data=sales.quarter1; /*3*/
run;
```

1 The LIBNAME statement assigns the libref sales to the physical location c:\myfiles.

2 The DATA step creates the data set sales.quarter1 and stores it in the library's physical location.
The PROC PRINT step references the data set by its two-level name, sales.quarter1.

Operating Environment Information: Here are examples of the LIBNAME statement for different operating environments. The rules for assigning and using librefs differ across operating environments. See the SAS documentation for your operating environment for specific information. The V9 engine is specified in these examples.

Table 12.4 Syntax for Assigning a Libref

<table>
<thead>
<tr>
<th>Operating Environment</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOS, Windows</td>
<td>libname mylibref v9 'c:\myfiles\data;</td>
</tr>
<tr>
<td>UNIX</td>
<td>libname mylibref v9 '/myfiles/data;</td>
</tr>
<tr>
<td>z/OS</td>
<td>libname mylibref v9 'userid.myfiles.data;</td>
</tr>
<tr>
<td></td>
<td>libname mylibref v9 '/myfiles/data;</td>
</tr>
</tbody>
</table>

Key Ideas

- The LIBNAME statement is a commonly used way to assign a libref to a physical location for a library.
- Even if your libraries are assigned by an administrator, understanding the library concept is useful.

See Also

- “LIBNAME Statement: V9 Engine” in SAS V9 LIBNAME Engine: Reference
- “Elements of a Library Assignment” on page 222
- “Example: Assign the User Library for Permanent Data” on page 242

Example: Assign a Libref by Using a Function

Example Code

This example creates a macro named test that uses functions to assign a libref and verify the assignment.

```sas
%macro test;
```
The macro variable `mylibref` specifies the libref name, `new`.

The macro variable `mydirectory` specifies the library’s physical location, `c:\example`.

An IF-THEN-ELSE statement calls the `%SYSFUNC` macro function, which in turn calls the `LIBNAME` function to attempt the library assignment. If an error or warning occurs, the message is written to the SAS log. If no error or warning occurs, then `success` is written to the log. Note that in a macro statement, you do not enclose character strings in quotation marks.

Another IF-THEN-ELSE statement calls the `%SYSFUNC` macro function, which calls the `LIBREF` function to validate the library assignment. Again, if an error or warning occurs, the message is written to the SAS log. If no error or warning occurs, then an informative message is written to the log.

Here is the output in the log from running the macro `test`:

```
 12  %test
  success
library new is assigned to c:\example
```

Key Ideas

- Some programmers prefer using SAS functions rather than statements. The `LIBNAME` function can assign or deassign (clear) a library assignment. The `LIBREF` function can verify a library assignment.
- The behavior of the `LIBNAME` function depends on the number of arguments.
- Be aware of additional rules when you use DATA step functions within macro functions.

See Also

- “LIBNAME Function” in SAS Functions and CALL Routines: Reference
- “LIBREF Function” in SAS Functions and CALL Routines: Reference
- “%SYSFUNC, %QSYSFUNC Macro Functions” in SAS Macro Language: Reference
- “Using DATA Step Functions within Macro Functions” in SAS Functions and CALL Routines: Reference
Example: Concatenate SAS Libraries

Example Code

This LIBNAME statement concatenates two SAS libraries:

libname lib3 (lib1 lib2);

The following figure demonstrates the concatenation.

Output 12.1  Concatenation of Lib1 and Lib2

<table>
<thead>
<tr>
<th>Name</th>
<th>Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLES</td>
<td>DATA</td>
</tr>
<tr>
<td>FORMATS</td>
<td>CATALOG</td>
</tr>
<tr>
<td>PEARS</td>
<td>DATA</td>
</tr>
</tbody>
</table>

lib1 + lib2  =  lib3

<table>
<thead>
<tr>
<th>Name</th>
<th>Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPLES</td>
<td>DATA</td>
</tr>
<tr>
<td>FORMATS</td>
<td>CATALOG</td>
</tr>
<tr>
<td>FRUIT</td>
<td>CATALOG</td>
</tr>
<tr>
<td>ORANGES</td>
<td>DATA</td>
</tr>
<tr>
<td>ORANGES</td>
<td>INDEX</td>
</tr>
<tr>
<td>PEARS</td>
<td>DATA</td>
</tr>
</tbody>
</table>

Notice that the index for apples does not appear in the concatenation. The lib2.apples data set has an index. However, the lib1.apples data set does not have an index, and lib1 is listed first in the concatenation. SAS suppresses the index when its associated data set is not part of the concatenation.

If multiple catalogs have the same name, their entries are concatenated. The lib3.formats catalog combines the entries of the lib1.formats and lib2.formats catalogs. For details, see “Catalog Concatenation” in SAS V9 LIBNAME Engine: Reference.

Key Ideas

- Library concatenation enables you to reference multiple libraries that are stored in different physical locations.
- When a data set is opened for input or update, the concatenated libraries are searched and the first occurrence of the data set is used. When a data set is created, it is created in the first library that is listed in the concatenation, even if a file exists with the same name in another library in the concatenation. Unwanted behavior could occur if data sets exist with the same name in the different locations.

See Also

- “Library Concatenation” on page 226
- “Catalog Concatenation” in SAS V9 LIBNAME Engine: Reference
Example: Access a Remote SAS Library by Using SAS/CONNECT Software

Example Code

This example uses a SAS/CONNECT spawner to access a SAS library that is stored on a remote computer.

```sas
options comamid=tcp;                                    /* 1 */
%let myserver=host.name.com;                            /* 2 */
signon myserver.__1234 user=userid password='mypw';     /* 3 */
libname reports '/myremotedata' server=myserver.__1234; /* 4 */
proc datasets library=reports;                          /* 5 */
run;
quit;
signoff myserver.__1234;
```

1 The COMAMID= system option specifies TCP/IP as the communications access method.

2 The `myserver` macro variable is assigned to the host name of the remote SAS/CONNECT server.

3 The SIGNON statement references the `myserver` macro variable followed by the port number that the SAS/CONNECT spawner is listening on. If your port number or service name is defined in the macro variable, then omit it from the SIGNON statement.

4 In the LIBNAME statement, do not specify an engine. Specify the location of your data, and specify the `myserver` macro variable in the `SERVER=` option. Include the port number if it is specified in the SIGNON statement.

5 PROC DATASETS runs in the client. Although the client accesses data that is on the server, the data is not written to the client’s local disk.

The following output from PROC DATASETS shows the REMOTE engine is assigned to the directory. In this case, the SAS client is running on Windows, the spawner is running on z/OS, and the data is located on UNIX.
If you license SAS/CONNECT or SAS/SHARE software, you can use a LIBNAME statement to read, write, and update server (remote) data as if it were stored on the client's disk.

SAS processes the data in client memory, which is overwritten in subsequent client requests for server data.

Remote access can be useful for accessing data sets across computers that have different architectures. Some catalog entry types are available for Read-Only access.

### Key Ideas

- "Types of Sign-ons" in SAS/CONNECT User's Guide
- "COMAMID= System Option" in SAS/CONNECT User's Guide
Example: Access a Remote SAS Library on a WebDAV Server

Example Code

The following LIBNAME statement accesses a WebDAV server:

```sas
libname davdata v9 "http://www.webserver.com/users/mydir/datadir" /*1*/
   webdav user="mydir"
   pw="12345";                           /*2*/
```

1  The LIBNAME statement assigns the libref davdata to the URL location of a WebDAV server.
2  The WEBDAV option is required in order to access a WebDAV server.

Key Ideas

- WebDAV (Web Distributed Authoring and Versioning) is a protocol that enhances the HTTP protocol. It provides a standard infrastructure for collaborative authoring across the internet. WebDAV enables you to edit web documents, stores versions for later retrieval, and provides a locking mechanism to prevent overwriting.

- When you access files on a WebDAV server, SAS pulls the file from the server to your local disk for processing. The files are temporarily stored in the Work library, unless you use the LOCALCACHE= option in the LIBNAME statement, which specifies a different location for temporary storage. When you finish updating the file, SAS pushes the file back to the WebDAV server for storage and removes the file from the local disk.

- SAS supports the WebDAV protocol under the UNIX and Windows operating environments.

See Also

- "LIBNAME Statement: WebDAV Server Access" in SAS Global Statements: Reference
- "FILENAME Statement: WebDAV Access Method" in SAS Global Statements: Reference
Example: Access DBMS Data as a SAS Library

Example Code

In this example, a SAS DATA step creates a Teradata table. To run this example, you must license a SAS/ACCESS interface.

```sas
libname mytddata teradata server=mytera user=myid password=mypw; /* 1 */
data mytddata.grades; /* 2 */	ininput student $ test1 test2 final;
datalines;
Fred 66 80 70
Wilma 97 91 98
;
proc datasets library=mytddata; /* 3 */
run;
quit;
```

1 The LIBNAME statement specifies the `mytddata` libref and TERADATA, which is the engine nickname for SAS/ACCESS Interface to Teradata. The statement also specifies connection options for Teradata. Change these options to specify your SAS/ACCESS connection values and any other options you need.

2 The DATA step creates a table named `grades`. The table is in the Teradata DBMS and is not a SAS data set.

3 The PROC DATASETS output for the `mytddata` library shows that the engine is Teradata. For the `grades` table, the SAS member type is DATA and the DBMS member type is TABLE.

**Output 12.3** PROC DATASETS Output Showing the DBMS Directory Information

<table>
<thead>
<tr>
<th>Directory</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Libref</td>
<td>MYTDDATA</td>
</tr>
<tr>
<td>Engine</td>
<td>TERADATA</td>
</tr>
<tr>
<td>Physical Name</td>
<td>mytera</td>
</tr>
<tr>
<td>Schema/User</td>
<td>myid</td>
</tr>
</tbody>
</table>

**Output 12.4** PROC DATASETS Output Showing the Library Contents for DBMS Content

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Member Type</th>
<th>DBMS Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>grades</td>
<td>DATA</td>
<td>TABLE</td>
</tr>
</tbody>
</table>
Key Ideas

- If you license a SAS/ACCESS interface for your DBMS data, you can submit a LIBNAME statement in SAS to run SAS code against the DBMS data. SAS can create or process a DBMS table as if it were a SAS data set.
- Some SAS/ACCESS interfaces are case sensitive. You might need to change the case of table or column names to comply with the requirements of your DBMS.
- When you access data that is stored in a DBMS or application other than SAS, usually you must specify connection information. You might need additional LIBNAME statement options that are specific to the engine.

See Also

- SAS/ACCESS documentation
- Chapter 13, “SAS Engines,” on page 257

Example: Access DBMS Data to Create a SAS View

Example Code

This example creates a SAS view that references the Teradata table that was created in "Example: Access DBMS Data as a SAS Library" on page 236. This code creates a SAS view, not a native Teradata view.

```sas
libname target 'U:\mysasdata';                                   /* 1 */
libname mytddata teradata server=mytera user=myid password=mypw; /* 2 */
data target.highgrades / view=target.highgrades;                 /* 3 */
   set mytddata.grades;
   where final gt 80;                                            /* 4 */
run;
proc print data=target.highgrades;                               /* 5 */
run;
```

1 In the LIBNAME statement, the libref target is assigned with a Windows path name, not the Teradata server. No engine is specified, so SAS assigns the default V9 engine.
2 This is the same SAS/ACCESS LIBNAME statement as in "Example: Access DBMS Data as a SAS Library" on page 236.
3 The DATA step creates a SAS view named highgrades that references the Teradata table named grades.
4 The view includes rows where the final variable is greater than 80.
5 PROC PRINT executes the view. Be aware that DATA step views do not retain the LIBNAME statement. Therefore, when you reference this view, you must first
submit LIBNAME statements for the mytddata library as well as the target library.

The PROC PRINT output shows that Wilma’s final grade is greater than 80. Fred’s final grade is not greater than 80, so Fred is not included in the view.

Output 12.5  PROC PRINT of a SAS View That References a Teradata Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Student</th>
<th>test1</th>
<th>test2</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Wilma</td>
<td>97</td>
<td>91</td>
<td>98</td>
</tr>
</tbody>
</table>

If you run PROC DATASETS, you can see that the member type of highgrades is VIEW.

```sas
proc datasets library=target;
run;
quit;
```

Output 12.6  PROC DATASETS Output for the Target Library

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Member Type</th>
<th>File Size</th>
<th>Last Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HIGHGRADES</td>
<td>VIEW</td>
<td>5KB</td>
<td>03/19/2019 12:47 03</td>
</tr>
</tbody>
</table>

For comparison, you could create a native table and view in Teradata by using the Teradata BTEQ tool. Submit the CREATE TABLE and CREATE VIEW commands in BTEQ to create a table named gradessql and a view named gradessqlview.

If you run PROC DATASETS against the mytddata library, you can see that the SAS member types differ from the DBMS member types.

```sas
proc datasets library=mytddata;
run;
quit;
```

Output 12.7  PROC DATASETS Output for the Mytddata Library

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Member Type</th>
<th>DBMS Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>grades</td>
<td>DATA</td>
<td>TABLE</td>
</tr>
<tr>
<td>2</td>
<td>gradessql</td>
<td>DATA</td>
<td>TABLE</td>
</tr>
<tr>
<td>3</td>
<td>gradessqlview</td>
<td>DATA</td>
<td>VIEW</td>
</tr>
</tbody>
</table>
Key Ideas

- Many customers use a DBMS as a large, ongoing data store. Rather than process an entire DBMS table, you can create a SAS view to query a subset of the data.
- After you submit a SAS/ACCESS LIBNAME statement, you can create a DATA step view or a PROC SQL view. Another option is a PROC SQL view that uses the SQL pass-through facility, which submits DBMS-specific syntax.
- A SAS view reflects the current state of data in the underlying data source. Alternatively, if the underlying DBMS data is not expected to change, you could read the DBMS data to create a permanent SAS data set.

See Also

- “SAS Views of DBMS Data” in SAS/ACCESS for Relational Databases: Reference
- “SAS Views” in SAS V9 LIBNAME Engine: Reference
- Chapter 13, “SAS Engines,” on page 257

Example: Automatically Create a New Subdirectory for a Library

Example Code

This example sets the DLCREATEDIR system option in order to create a subdirectory in the file system for a library.

In the example, the directory c:\example exists in the file system, but the subdirectory project does not. Because the DLCREATEDIR system option is set, SAS creates project.

```sas
options dlcreatedir;
libname mynewlib 'c:\example\project';
```

The SAS log includes a note that the library was created:

```
NOTE: Library MYNEWLIB was created.
NOTE: Libref MYNEWLIB was successfully assigned as follows:
    Engine:       V9
    Physical Name: c:\example\project
```

z/OS Specifics: On z/OS, the shipped default is DLCREATEDIR.
Key Ideas

- You can set the DLCREATEDIR system option to create a subdirectory for the SAS library that is specified in the LIBNAME statement if that directory does not exist.
- SAS can create one new subdirectory of an existing directory. In other words, SAS creates only the final component in the path name.
- The shipped default is NODLCREATEDIR for all environments except z/OS.

See Also

- “DLCREATEDIR System Option” in SAS System Options: Reference
- “DLCREATEDIR System Option: z/OS” in SAS Companion for z/OS

Example: Deassign (Clear) a Libref

Example Code

The following statement deassigns the libref mylib from its physical location:

    libname mylib clear;

Use the _ALL_ keyword in the LIBNAME statement to deassign all library assignments (other than system libraries):

    libname _all_ clear;

You can also use the LIBNAME function. The following code deassigns the libref new, which was assigned in “Example: Assign a Libref by Using a Function” on page 230:

    %macro test;
    %if (%sysfunc(libname(new))) %then
       %put %sysfunc(sysmsg());
    %mend test;
    %test
Key Ideas

- Deassigning a libref can be useful for freeing up resources, especially for shared data.
- By default, SAS deassigns librefs automatically at the end of each SAS session.
- You can request to deassign a libref before the end of the session:
  - In the LIBNAME statement, specify the libref name and CLEAR to deassign a single libref. Specify _ALL_ and CLEAR to deassign all currently assigned librefs. System libraries such as Sashelp and Sasuser are not deassigned.
  - In the LIBNAME function, use the one-argument form to deassign the libref. In certain operating environments, you can deassign the libref by specifying a blank between quotation marks for the library location. However, in some operating environments, a blank for the library location assigns a libref to the current directory. Therefore, the one-argument form is recommended.
- If you use a method to persist a library assignment beyond the current session, you cannot permanently deassign the libref by using the LIBNAME statement or LIBNAME function. The libref is deassigned for the current session only and is reassigned when you start a new session.

See Also

- "LIBNAME Statement: V9 Engine" in SAS V9 LIBNAME Engine: Reference
- "LIBNAME Function" in SAS Functions and CALL Routines: Reference

Examples: Access Data without Using a Libref

Example: Use the Work Library for Temporary Data

Example Code

If you have not set a User library, the following code writes mytable to the Work library.

Notice the one-level name mytable does not have a libref. This code behaves the same if you specify work.mytable.

```sas
data mytable;
  x=1;
run;
proc contents data=mytable;
run;
```
Here is a portion of the PROC CONTENTS output, showing that the Work library is used.

**Output 12.8 Portion of PROC CONTENTS Output for a Data Set in Work**

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>WORK.MYTABLE</th>
<th>Observations</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>1</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>01/30/2019 14:25:51</td>
<td>Observation Length</td>
<td>8</td>
</tr>
<tr>
<td>Last Modified</td>
<td>01/30/2019 14:25:51</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td>Sorted</td>
<td>NO</td>
</tr>
<tr>
<td>Data Set Type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>WINDOWS_S4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>wlatin1 Western (Windows)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key Ideas**

- You can use the Work library for intermediate or temporary results.
- By default, if you specify a one-level name when you create a data set, the data set is stored temporarily in the Work library. If you want to specify a one-level name to create and use permanent files instead of temporary files, then assign a User library.
- The Work library is automatically defined by SAS at the beginning of each SAS session. Typically, files in the Work library are deleted at the end of each SAS session if the session terminates normally.

**See Also**

- “Work Library (Temporary)” on page 228
- “Example: Assign the User Library for Permanent Data” on page 242
- “Example: Access a Data Set by Specifying a Path Name in Quotation Marks” on page 244

**Example: Assign the User Library for Permanent Data**

**Example Code**

If you want to specify a one-level name to create and use permanent files instead of temporary files, then assign a User library. This example references the data set...
quarter1, which was created in “Example: Assign a Libref by Using the LIBNAME Statement” on page 229.

```sas
libname sales 'c:\myfiles'; /*1*/
options user=sales; /*2*/
proc print data=quarter1; /*3*/
run;
```

1 The LIBNAME statement assigns the libref sales to a physical location for the library.

2 The USER= system option specifies the sales library as the default for one-level names.

3 The PROC PRINT step references the data set by its one-level name, quarter1. The log output confirms the data is read from sales.quarter1, even though sales is not specified in the PROC PRINT:

```
1 libname sales 'c:\myfiles';
NOTE: Libref SALES was successfully assigned as follows:
  Engine:        V9
  Physical Name: c:\myfiles
1     !                             /*1*/
2   options user=sales; /*2*/
3
4 proc print data=quarter1; /*3*/
NOTE: Writing HTML Body file: sashtml.htm
 run;
NOTE: There were 2 observations read from the data set SALES.QUARTER1.
```

Instead of setting the USER= system option, you can use a LIBNAME statement or LIBNAME function to assign the User library. Try these two examples:

```sas
libname user 'c:\myfiles';
proc print data=quarter1;
run;

data _null_; x=libname ('user', 'c:\myfiles'); run;
proc print data=quarter1;
run;
```

The behavior is the same as setting the USER= system option, except the log shows the libref as user:

```
NOTE: There were 2 observations read from the data set USER.QUARTER1.
```
Key Ideas

- By default, if you specify a one-level name when you create a data set, the data set is stored temporarily in the Work library. If you want to specify a one-level name to create and use permanent files instead of temporary files, then assign a User library.
- You can use the USER= system option to set a User library. You can also use the common ways to assign libraries, such the LIBNAME statement. Specify a previously assigned libref or a physical location.
- After you set the User library, if you want to store a temporary data set in the Work library, you must specify the Work libref in a two-level name.

See Also

- “User Library” on page 228
- “USER= System Option” in SAS System Options: Reference
- “Example: Use the Work Library for Temporary Data” on page 241
- “Example: Access a Data Set by Specifying a Path Name in Quotation Marks” on page 244

Example: Access a Data Set by Specifying a Path Name in Quotation Marks

Example Code

This example prints a data set that is identified by its full path name and file name instead of a libref and data set name. The data set quarter1 is created in “Example: Assign a Libref by Using the LIBNAME Statement” on page 229. A Windows path name is used for this demonstration.

```sas
proc print data='c:\myfiles\quarter1.sas7bdat';
run;
```

Key Ideas

- Instead of using a two-level name libref.file-name, you can omit the libref and hardcode the full path name and file name, enclosed in quotation marks. You can omit the file extension if the file is a SAS data set.
- Many language elements do not accept a physical location. For the restrictions, see “Accessing Data without Using a Libref” on page 225.
Example: Specify a Fileref for a File That Is Not a Library Member

Example Code

This example uses a fileref to identify the location of raw data. The file sampdata.txt is an external text file.

```sas
filename test url "http://support.sas.com/publishing/cert/sampdata.txt" ; /*1*/
data credit; /*2*/
infile test firstobs=945 obs=954; /*3*/
input Account $ 1-4 Name $ 6-22 Type $ 24 Transaction $ 26-31;
run;
proc print data=credit; /*4*/
run;
```

1 The FILENAME statement specifies the location of a file. The fileref is test, the access method is URL, and the location is the full URL of the file. If you download sampdata.txt to a file system that is accessible from your SAS session (such as an NFS-mounted directory), then the URL access method is not necessary.

2 The DATA step creates a temporary data set named credit in the Work library.

3 The INFILE statement specifies the fileref test, which was assigned in the FILENAME statement. The FIRSTOBS= and OBS= data set options specify the lines to read from the external file. The file sampdata.txt contains many sample data sets and programs. This example uses lines 945–954 only, which is raw data that is delimited by spaces.

4 PROC PRINT verifies that the external data is imported as a SAS data set.
Output 12.9  PROC PRINT Output Showing Successful Import from the External File

<table>
<thead>
<tr>
<th>Obs</th>
<th>Account</th>
<th>Name</th>
<th>Type</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1118</td>
<td>ART CONTUCK</td>
<td>D</td>
<td>57.69</td>
</tr>
<tr>
<td>2</td>
<td>2287</td>
<td>MICHAEL WINSTONE</td>
<td>D</td>
<td>145.89</td>
</tr>
<tr>
<td>3</td>
<td>6201</td>
<td>MARY WATERS</td>
<td>C</td>
<td>45.00</td>
</tr>
<tr>
<td>4</td>
<td>7621</td>
<td>MICHELLE STANTON</td>
<td>A</td>
<td>304.45</td>
</tr>
<tr>
<td>5</td>
<td>6621</td>
<td>WALTER LUND</td>
<td>C</td>
<td>234.76</td>
</tr>
<tr>
<td>6</td>
<td>1066</td>
<td>KATHERINE MORRY</td>
<td>A</td>
<td>64.98</td>
</tr>
<tr>
<td>7</td>
<td>0556</td>
<td>LEE McDonald</td>
<td>D</td>
<td>70.82</td>
</tr>
<tr>
<td>8</td>
<td>7621</td>
<td>ELIZABETH WESTIN</td>
<td>C</td>
<td>188.23</td>
</tr>
<tr>
<td>9</td>
<td>0265</td>
<td>JEFFREY DONALDSON</td>
<td>C</td>
<td>78.90</td>
</tr>
<tr>
<td>10</td>
<td>1010</td>
<td>MARTIN LYNN</td>
<td>D</td>
<td>150.55</td>
</tr>
</tbody>
</table>

Key Ideas

- A text file that contains delimited data (also called raw data) is not a SAS library member. Therefore, you cannot use a libref to refer to the location of the file. Instead, use the FILENAME statement to assign a SAS fileref.
- The FILENAME statement can assign a fileref to an external file or an output device, deassign a fileref and external file, or list attributes of external files.
- If you do not need to reuse a fileref, and if you do not need an access method such as URL, you can omit the FILENAME statement. Instead, you can choose to specify the quoted path name and file name in the INFILE statement. However, filerefs can be helpful for many of the same reasons as librefs.

See Also

- Chapter 15, “Raw Data,” on page 315
- “FILENAME Statement: URL Access Method” in SAS Global Statements: Reference
Examples: View Information about a Library

Example: Print Information about a Library and Its Members

Example Code

The DATASETS procedure prints a list, or directory, of the members in a SAS library.

```
libname myfiles 'c:\example';
proc datasets library=myfiles;
run;
quit;
```

In the example output from PROC DATASETS, the directory information includes the libref, the physical location, and other attributes of the library. The second section of the output shows the name of each member and its member type, followed by any associated files. In the output, notice that the flowers data set has an index.

**Output 12.10**  PROC DATASETS Output Showing the Directory Information

<table>
<thead>
<tr>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libref</td>
</tr>
<tr>
<td>Engine</td>
</tr>
<tr>
<td>Physical Name</td>
</tr>
<tr>
<td>Filename</td>
</tr>
<tr>
<td>Owner Name</td>
</tr>
<tr>
<td>File Size</td>
</tr>
<tr>
<td>File Size (bytes)</td>
</tr>
</tbody>
</table>

**Output 12.11**  PROC DATASETS Output Showing the Library Members

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Member Type</th>
<th>File Size</th>
<th>Last Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONTAINERS</td>
<td>VIEW</td>
<td>5KB</td>
<td>07/19/2018 08:33:19</td>
</tr>
<tr>
<td>2</td>
<td>FLOWERS</td>
<td>DATA</td>
<td>192KB</td>
<td>01/18/2019 17:24:24</td>
</tr>
<tr>
<td></td>
<td>FLOWERS</td>
<td>INDEX</td>
<td>9KB</td>
<td>01/18/2019 17:24:24</td>
</tr>
<tr>
<td>3</td>
<td>FORMATS</td>
<td>CATALOG</td>
<td>17KB</td>
<td>07/03/2018 14:06:19</td>
</tr>
<tr>
<td>4</td>
<td>RESTOCK</td>
<td>DATA</td>
<td>128KB</td>
<td>11/09/2018 11:50:50</td>
</tr>
</tbody>
</table>
Key Ideas

- Without the CONTENTS statement, PROC DATASETS returns information about the directory. Add the CONTENTS statement if you want more information about a library member or members. Like many of the PROC DATASETS statements, CONTENTS is also available as a stand-alone procedure, PROC CONTENTS.

- Specify the DETAILS option to include information about the number of observations, number of variables, number of indexes, and data set labels. DETAILS is available as a PROC DATASETS option and as a system option.

- The returned information varies according to the operating environment and the engine.

- Any files that are not considered to be library members are not listed in the library directory. Some examples are internal SAS utility files or text files that contain SAS programs. You might see these files in the file system, but they are not listed in PROC DATASETS output.

See Also

- “DATASETS Procedure” in Base SAS Procedures Guide
- “DETAILS System Option” in SAS System Options: Reference

Example: Return Library Attributes by Using the LIBNAME Statement

Example Code

This example uses the LIST argument in the LIBNAME statement. The LIST argument prints the library’s attributes in the log.

```
libname myfiles 'c:\example';
libname myfiles list;
```

The following information is written to the log.

*Example Code 12.1  Library Information from SAS Log*

```
NOTE: Libref= MYFILES
Scope= DMS Process
Engine= V9
Physical Name= c:\example
Filename= c:\example
Owner Name= userid
FileSize= 8KB
FileSize (bytes)= 8192
```
Key Ideas

- Specify the libref name to list the attributes of a single SAS library. Specify _ALL_ to list the attributes of all SAS libraries that have librefs in your current session.
- If you specify _ALL_, then librefs that are defined as environment variables appear only if you have already used those librefs in a SAS statement.

See Also

- “LIBNAME Statement: V9 Engine” in SAS V9 LIBNAME Engine: Reference
- “Elements of a Library Assignment” on page 222
- “The SAS Log” on page 592

Example: Return Library Location by Using a Function

Example Code

This example uses the PATHNAME function to return the physical location that is assigned to the libref myfiles.

```sas
libname myfiles 'c:\example';
data _null_; length path $ 100; path=pathname('myfiles'); put path; run;
```

The PUT statement writes the path name to the SAS log:

```
c:\example
```

Key Ideas

- The PATHNAME function returns the physical location that is assigned to a libref. You can also view the physical location by using PROC DATASETS or by using the LIST argument in the LIBNAME statement.
- Several other SAS functions are available to return information about a SAS library or a library member.

See Also

- “ATTRC Function” in SAS Functions and CALL Routines: Reference
Examples: Manage SAS Libraries

Example: Copy a SAS Library

Example Code

The following `COPY` procedure example copies the entire `myfiles` library to the `target` library. PROC COPY has many useful options, but none are specified, so the default behavior such as CLONE is used.

```sas
libname myfiles 'c:\example';
libname target 'd:\new';
proc copy in=myfiles out=target;
run;
```

The SAS log shows the results of the copy:

```
NOTE: Copying MYFILES.CONTAINERS to TARGET.CONTAINERS (memtype=VIEW).
NOTE: Copying MYFILES.FLOWERS to TARGET.FLOWERS (memtype=DATA).
NOTE: Simple index plantname has been defined.
NOTE: There were 7 observations read from the data set MYFILES.FLOWERS.
NOTE: The data set TARGET.FLOWERS has 7 observations and 3 variables.
NOTE: Copying MYFILES.FORMATS to TARGET.FORMATS (memtype=CATALOG).
NOTE: Copying MYFILES.RESTOCK to TARGET.RESTOCK (memtype=DATA).
NOTE: There were 7 observations read from the data set MYFILES.RESTOCK.
NOTE: The data set TARGET.RESTOCK has 7 observations and 4 variables.
NOTE: PROCEDURE COPY used (Total process time):
      real time           5.70 seconds
      cpu time            0.87 seconds
```
Key Ideas

- SAS file management utilities such as PROC COPY have the following capabilities:
  - can operate on all library members or selected ones only
  - can copy, rename, or move a data set and its associated files, such as indexes, in most cases
  - are supported on all operating environments
- If you copy a library to a different operating environment, or to a different character encoding, specify the NOCLONE option for PROC COPY. The NOCLONE option changes the data representation and encoding to that of the OUT= library. In addition, if cross-environment data access (CEDA) is invoked, then you must be aware of CEDA restrictions.
- If you use SAS Studio, you can perform some file maintenance tasks in the navigation pane. If you use the SAS windowing environment, you can perform maintenance tasks in the SAS Explorer window.

See Also

- “COPY Procedure” in Base SAS Procedures Guide
- “DATASETS Procedure” in Base SAS Procedures Guide
- “CATALOG Procedure” in Base SAS Procedures Guide

Example: Migrate a SAS Library

Example Code

The following MIGRATE procedure example migrates members in a SAS library to take advantage of features that are provided in a newer SAS release. This example does not use a SAS/CONNECT or SAS/SHARE server, which is required in some cases.

Run this code in a session of the SAS release that you are migrating to.

```
libname myfiles 'c:\example';
libname target 'd:\new';
proc migrate in=myfiles out=target;
run;
```

The SAS log shows the results of the migration. Notice that PROC MIGRATE calls PROC CPORT to migrate catalogs.
NOTE: The BUFSIZE= option was not specified with the MIGRATE procedure. The migrated library members will use the current value for BUFSIZE. For more information, see the PROC MIGRATE documentation.

NOTE: Migrating MYFILES.CONTAINERS to TARGET.CONTAINERS (memtype=VIEW).
NOTE: Migrating MYFILES.FLOWERS to TARGET.FLOWERS (memtype=DATA).
NOTE: Simple index plantname has been defined.
NOTE: There were 7 observations read from the data set MYFILES.FLOWERS.
NOTE: The data set TARGET.FLOWERS has 7 observations and 3 variables.
NOTE: Migrating MYFILES.RESTOCK to TARGET.RESTOCK (memtype=DATA).
NOTE: There were 7 observations read from the data set MYFILES.RESTOCK.
NOTE: The data set TARGET.RESTOCK has 7 observations and 4 variables.

NOTE: PROC CPORT begins to transport catalog MYFILES.FORMATS
NOTE: Entry TESTFMT.FORMAT has been transported.
NOTE: PROCEDURE MIGRATE used (Total process time):
   real time           2.34 seconds
   cpu time            0.15 seconds

If you migrate to a different operating environment (for example, from Windows to UNIX), PROC MIGRATE has some limitations. For example, the following log note says that cross environment data access (CEDA) was used. This message informs you that performance can be slowed by CEDA.

NOTE: Data file MYFILES.FLOWERS.DATA is in a format that is native to another host, or the file encoding does not match the session encoding. Cross Environment Data Access will be used, which might require additional CPU resources and might reduce performance.

An error message like the following is also due to CEDA, and it indicates that the formats catalog was not migrated. To migrate catalogs with PROC MIGRATE to an incompatible operating environment, you must use a SAS/CONNECT or SAS/SHARE server to access the IN= library.

ERROR: File MYFILES.FORMATS.CATALOG was created for a different operating system.
WARNING: No data is available.
Key Ideas

- PROC MIGRATE is usually the best way to migrate members in a SAS library to the current SAS version. PROC MIGRATE is a one-step copy procedure that retains the data attributes that most users want in a data migration.

- If either of the following issues is present, then a SAS/CONNECT or SAS/SHARE server is required, and different syntax is used. See “Migrating from a SAS®9 Release by Using SAS/CONNECT” in Base SAS Procedures Guide.
  - if you do not have direct access to the source library from the target session via a Network File System (NFS)
  - if the source library contains catalogs and if the processing invokes CEDA on the target session

- Alternatively, if you have direct access to the source library through NFS, then you can use cross-environment data access (CEDA) instead of migrating. This Read-Only access is automatic and transparent, but you must be aware of the restrictions. See Chapter 33, “Cross-Environment Data Access,” on page 649.

- If you are changing to a different character encoding that uses more bytes to represent the characters, you might want to use the CVP engine as part of the copy or migration process. See “Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process” in SAS V9 LIBNAME Engine: Reference.

See Also

- “MIGRATE Procedure” in Base SAS Procedures Guide
- “Compatibility and Migration” in SAS V9 LIBNAME Engine: Reference

Example: Copy a SAS Library by Using a Transport File

Example Code

This example copies a SAS library across environments by using the CPORT and CIMPORT procedures. A multistep process is necessary:

1. In the source environment, use PROC CPORT to create a transport file.

2. Use communication software (such as FTP) or a storage device to move the transport file to the target environment. If you use FTP, transfer the file in binary mode.

3. In the target environment, use PROC CIMPORT to import the library from the transport file.

For step 1, PROC CPORT creates the transport file mytransfer, which is referenced by the fileref tranfile.

```
libname source 'c:\example';
filename tranfile 'c:\myfiles\mytransfer';
proc cport library=source file=tranfile;
```
run;

In the log, notice that containers is not ported, because it is a SAS view.

NOTE: Not porting SOURCE.CONTAINERS because memtype is not supported.

NOTE: PROC CPOR begins to transport data set SOURCE.FLOWERS
NOTE: The data set contains 3 variables and 7 observations.
    Logical record length is 24.
NOTE: Transporting data set index information.

NOTE: PROC CPOR begins to transport catalog SOURCE.FORMATS
NOTE: The catalog has 1 entries and its maximum logical record length is 120.
NOTE: Entry TESTFMT.FORMAT has been transported.

NOTE: PROC CPOR begins to transport data set SOURCE.RESTOCK
NOTE: The data set contains 4 variables and 7 observations.
    Logical record length is 40.
NOTE: PROCEDURE CPOR used (Total process time):
    real time           18.14 seconds
    cpu time            0.17 seconds

For step 2, the user copies the mytransfer file from their Windows environment to a UNIX environment.

For step 3, PROC CIMPORT creates the target library by importing the contents of mytransfer.

    libname target '/mydata/example';
    filename tranfile '/mydata/mytransfer';
    proc cimport library=target infile=tranfile;
    run;

The log indicates that the import was successful.

NOTE: PROC CIMPORT begins to create/update data set TARGET.FLOWERS
NOTE: The data set index plantname is defined.
NOTE: Data set contains 3 variables and 7 observations.
    Logical record length is 24

NOTE: PROC CIMPORT begins to create/update catalog TARGET.FORMATS
NOTE: Entry TESTFMT.FORMAT has been imported.
NOTE: Total number of entries processed in catalog TARGET.FORMATS: 1

NOTE: PROC CIMPORT begins to create/update data set TARGET.RESTOCK
NOTE: Data set contains 4 variables and 7 observations.
    Logical record length is 40

NOTE: PROCEDURE CIMPORT used (Total process time):
    real time           0.74 seconds
    cpu time            0.00 seconds
Key Ideas

- PROC CPORT and PROC CIMPORT have several limitations as compared to PROC MIGRATE. Only use this method for migration if PROC MIGRATE would require SAS/CONNECT or SAS/SHARE software, and you do not have access to that software. PROC MIGRATE migrates an entire library, including data sets, catalogs, and most other member types. See "Example: Migrate a SAS Library across Environments by Using SAS/CONNECT" in SAS V9 LIBNAME Engine: Reference.

- PROC CPORT supports SAS data sets and catalogs but not other member types.

- If you use FTP to move the transport file to the target environment, transfer the file in binary mode.

- When you are transcoding to a new encoding, truncation could occur. If truncation occurs, you must expand variable lengths. You can either use the CVP engine with PROC CPORT or use the EXTENDVAR= option with PROC CIMPORT.

- Transport files that are created by the CPORT procedure are not interchangeable with transport files that are created by the XPORT engine.

See Also

- “CPORT Procedure” in Base SAS Procedures Guide
- “CIMPORT Procedure” in Base SAS Procedures Guide
- “PROC CPORT and PROC CIMPORT” in Moving and Accessing SAS Files
- “Compatibility and Migration” in SAS V9 LIBNAME Engine: Reference
Definitions for SAS Engines

An *engine* is a component of SAS software that reads from and writes to a file in a particular file format. (Some engines are read-only.)

Most engines are referred to as *library engines*, because they access a group of SAS files that are used as a *SAS library*. For more information, see Chapter 12, “SAS Libraries,” on page 219.

The SAS *Multi Engine Architecture* enables you to access a variety of file formats:

- Certain engines process SAS data only. See “Examples: Use a SAS Engine to Process SAS Data” on page 266.
Other engines interpret data from other applications (for example, DBMS, XML, JSON, or Microsoft Excel). These engines apply a layer of abstraction so that SAS can process the external data as if it were a SAS data set or a SAS library of data sets. See “Examples: Use a SAS Engine to Process External Data” on page 275.

You specify an engine in a library assignment or in the ENGINE system option:

- A library assignment consists of a libref, an engine, a physical location, and options that are specific to the engine or environment. See “Elements of a Library Assignment” on page 222.
- A default engine is shipped with the software. To change the default, specify the ENGINE system option at invocation. See “Example: Set a Default Engine” on page 274.

How Engines Work with Files

The following figure shows how files are accessed through an engine. In the diagram, the shaded components represent external file types or applications.

Figure 13.1 How SAS Engines Access Data

1. A SAS data set or table is stored in one or more physical files, depending on the engine and attributes. If you have licensed the appropriate SAS engine, SAS can read and write data that is created by other applications, such as a DBMS. Base SAS can read some raw data. For example, the DATA step or the IMPORT procedure can read comma-separated data from a text file. The DATA step or procedure provides the data to the V9 engine for output to a SAS data set. See Chapter 15, “Raw Data,” on page 315.

2. When you specify a SAS data set name, the engine locates the stored file or files to obtain metadata. V9 engine data sets contain metadata (also known as
descriptor information) within the data set file. Other file types store metadata in a separate file. Although SAS can determine the metadata for many external file types, you might be required to provide additional instructions. The metadata provides information such as variable names and attributes, and whether the file has special processing characteristics such as indexes or compressed observations.

Note that more than one engine might be involved in processing. For example, in a DATA step, one engine could be used to read data, and a different engine used to write data.

3 The engine uses the metadata to organize the data in the standard logical form for SAS processing. This standard form is the SAS data set model. A SAS data set consists of data values that are organized into variables (columns) and observations (rows).

Similar to the SAS data set model, the SAS library model is a group of data sets and other library members that are organized in a logical form for processing. When files are accessed as a SAS library, you can use SAS utilities such as the DATASETS procedure to list their contents and to manage them.

4 SAS procedures and DATA step statements process the data in this logical form, the SAS data set model. During processing, the engine passes down whatever instructions are necessary to open and close physical files and to read and write data. Processing can occur in the SAS data set model without the data ever being physically stored as a SAS data set. If the data is stored in an external application, such as a DBMS, some SAS procedures can pass processing to that application.
## Engine Characteristics

### Summary of SAS Engines

*Table 13.1 Commonly Used SAS Engines*

<table>
<thead>
<tr>
<th>LIBNAME Engine Nickname</th>
<th>Uses</th>
<th>Examples and Documentation</th>
</tr>
</thead>
</table>
| V9 or BASE              | Shipped default Base SAS engine  
SAS data sets              | “Example: Assign the V9 Engine in a LIBNAME Statement” on page 266  
“Example: Read a Comma-Delimited File” on page 275  
“Example: Set a Default Engine” on page 274  
“Default Base SAS Engine (V9 Engine)” on page 262  
SAS V9 LIBNAME Engine: Reference |
| SPDE                    | Alternative Base SAS engine  
SPD Engine data sets  
Big data  
Multi-CPU threaded processing  
Optional Hadoop storage | “Example: Assign the SPD Engine in a LIBNAME Statement” on page 267  
“Example: Read and Write SAS Data in Hadoop by Using the SPD Engine” on page 268  
SAS Scalable Performance Data Engine: Reference  
SAS SPD Engine: Storing Data in the Hadoop Distributed File System |
| ORC                     | Import and export open-source file format in Base SAS  
Big data  
Cloud storage | “Example: Read an ORC Table” in SAS Viya Engine for ORC: Reference  
SAS Viya Engine for ORC: Reference |
<table>
<thead>
<tr>
<th>LIBNAME Engine Nickname</th>
<th>Uses</th>
<th>Examples and Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS/ACCESS engines (multiple)</td>
<td>External data from other applications</td>
<td>“Example: Read Microsoft Excel Data by Using a SAS/ACCESS Engine” on page 277</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Example: Create Data in a DBMS by Using a SAS/ACCESS Engine” on page 278</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Example: Embed a SAS/ACCESS LIBNAME Statement in a PROC SQL View” on page 279</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“Example: Access DBMS Data by Using the SQL Pass-Through Facility” on page 281</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAS/ACCESS for Relational Databases: Reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAS/ACCESS for Nonrelational Databases: Reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAS/ACCESS Interface to PC Files: Reference</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAS/ACCESS documentation</td>
</tr>
<tr>
<td>CAS</td>
<td>Big data</td>
<td>“Example: Load a SAS Data Set to a CAS Server” on page 272</td>
</tr>
<tr>
<td></td>
<td>Cloud computing</td>
<td></td>
</tr>
<tr>
<td>CVP</td>
<td>National language support</td>
<td>“Example: Avoid Truncation by Using the CVP Engine with the V9 Engine” on page 269</td>
</tr>
<tr>
<td>XMLV2</td>
<td>Import and export XML in Base SAS</td>
<td>“Example: Import XML Data by Using the XMLV2 Engine” on page 282</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAS XMLV2 and XML LIBNAME Engines: User’s Guide</td>
</tr>
<tr>
<td>LIBNAME Engine Nickname</td>
<td>Uses</td>
<td>Examples and Documentation</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>JSON</td>
<td>Import and export JSON in Base SAS</td>
<td>“Example: Import JSON Data by Using the JSON Engine” on page 284</td>
</tr>
<tr>
<td></td>
<td></td>
<td>“LIBNAME Statement: JSON Engine” in SAS Global Statements: Reference</td>
</tr>
<tr>
<td>SASIOLA and SASHDAT</td>
<td>SAS LASR Analytic Server Multi-threaded distributed processing Optional Hadoop storage</td>
<td>SAS LASR Analytic Server: Reference Guide</td>
</tr>
<tr>
<td>FEDSVR</td>
<td>External data or SAS data by using the SAS Federation Server</td>
<td>SAS LIBNAME Engine for SAS Federation Server: User’s Guide</td>
</tr>
<tr>
<td>JMP</td>
<td>JMP statistical discovery software</td>
<td>“LIBNAME Statement: JMP Engine” in SAS/ACCESS Interface to PC Files: Reference</td>
</tr>
<tr>
<td>INFOMAP</td>
<td>SAS Information Maps</td>
<td>Base SAS Guide to Information Maps</td>
</tr>
<tr>
<td>META</td>
<td>SAS Metadata Server</td>
<td>SAS Language Interfaces to Metadata</td>
</tr>
</tbody>
</table>

Default Base SAS Engine (V9 Engine)

The shipped default Base SAS engine is BASE. In SAS®9 and SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice. See “Example: Assign the V9 Engine in a LIBNAME Statement” on page 266. See also “Example: Set a Default Engine” on page 274.
The file format for SAS 9, SAS 8, and SAS 7 data sets is very similar, so SAS does not differentiate between them. See “Default Engine for a Mixed Library” in SAS V9 LIBNAME Engine: Reference. However, new file features can make a data set unusable in an earlier release. See “File Features That Are Not Supported in a Previous Release” in SAS V9 LIBNAME Engine: Reference.

If you see an unexpected engine name in the log or in output from PROC CONTENTS or PROC DATASETS, the engine was probably called internally from the V9 engine. These internal engines are not valid for a user to specify. For example, when you create a DATA step view, SAS calls the SASDSV engine. When you create a PROC SQL view, SAS calls SQLVIEW.

When you use SAS/CONNECT or SAS/SHARE software, SAS calls the REMOTE engine. Usually, you should not specify the REMOTE engine in a LIBNAME statement, because the client automatically determines which engine to use.

---

Legacy Engines

V6 Compatibility Engine

The SAS 6 compatibility engine can automatically support some processing of SAS 6 files in SAS 9 without requiring you to convert the file to the SAS 9 format. For more information, see the Migration Focus Area at support.sas.com/migration.

Tape Engines

The tape engines are sequential engines and are typically used for legacy data storage.

A tape engine processes SAS files on storage media that do not provide random access methods (for example, tape or sequential format on disk). The tape engines require less overhead than the V9 engine because sequential access is simpler than random access. The following tape engines are available:

- V9TAPE (alias TAPE) processes SAS 9, SAS 8, and SAS 7 files.
- V6TAPE processes SAS 6 files.

Before you store SAS libraries in sequential format, consider the following restrictions:

- DATA is the only member type that is useful for purposes other than backup and restore. Member types CATALOG, VIEW, and MDDB are supported for backup and restore purposes only.
- You cannot use random (direct) access with sequential SAS data sets. Some examples of direct access are the use of indexes, or the use of the POINT= or KEY= options in the SET or MODIFY statements.
- The DATASETS procedure is not supported.
- Update processing is not supported.
- Audit trails, extended attributes, generation data sets, indexes, integrity constraints, item stores, and AES-encrypted data are not supported.
- You cannot reference more than one data set from a sequential library in a single DATA step or PROC step. However:
You can access two or more SAS files in different sequential libraries or on different tapes at the same time, if enough tape drives are available.

You can access a SAS file during one DATA or PROC step. You can then access another SAS file in the same sequential library or on the same tape during a later DATA or PROC step.

You can use the OPEN=DEFER option of the SET statement in the DATA step to delay opening multiple data sets. OPEN=DEFER opens the first data set during the compilation phase and opens subsequent data sets during the execution phase. See the SET statement.

The tape engines are not supported on Windows.

Operating Environment Information: For more details about storing and accessing SAS files in sequential format, see the following documentation:

- SAS Companion for UNIX Environments
- SAS Companion for z/OS

Transport Engine

The XPORT engine creates files in transport format, which uses an environment-independent standard for character encoding and numeric representation. Transport files that are created by the XPORT engine can be transferred across operating environments. The transport file can be read on the target environment by using the XPORT engine with the DATA step or the COPY procedure (or the COPY statement of the DATASETS procedure).

The XPORT engine is not a best practice for moving SAS files across environments. For other methods, see “Strategies for Moving and Accessing SAS Files” in Moving and Accessing SAS Files. If you are migrating to a newer SAS release, see “MIGRATE Procedure” in Base SAS Procedures Guide.

SPSS Engine

The SPSS engine reads data that was created in the external application SPSS. The engine reads the SPSS portable file format, which has a .por extension. This file format is analogous to the transport format for SAS data sets. An SPSS portable file (also called an export file) must be created by using the SPSS EXPORT command. Under z/OS, the SPSS engine also reads SPSS Release 9 files and SPSS-X files in either compressed or uncompressed format. The SPSS engine is a sequential engine.

To read SPSS files in SAS, choose one of these methods:

- If you have a .por file, you can use the SPSS engine or the CONVERT procedure to convert from SPSS to a SAS data set. These methods are documented in the SAS documentation for your operating environment. See “Operating Environment Information” on page 266.

- If you have a .sav file and not a .por file, you must have a license to SAS/ACCESS to PC Files. Use the IMPORT procedure to convert from SPSS to a SAS data set. See SAS/ACCESS Interface to PC Files: Reference.

- For examples, see also Sample 34629: Coming to SAS from SPSS.
Access Patterns

SAS procedures and statements can read observations in SAS data sets in one of four general patterns:

- **sequential access**
  - processes observations one after the other, starting at the beginning of the file and continuing in sequence to the end of the file. For example, the XMLV2 and JSON engines perform sequential processing only.

- **random access**
  - processes observations according to the value of some indicator variable without processing previous observations. For example, the POINT= option in the SET statement requires random access to observations as well as the ability to calculate observation numbers from record identifiers within the file.

- **BY-group access**
  - groups and processes observations in order of the values of the variables that are specified in a BY statement.

- **multiple-pass**
  - performs two or more passes on data when required by SAS statements or procedures.

If a statement or procedure tries to access a data set whose engine does not support the required access pattern, SAS prints an appropriate error message in the SAS log.

Levels of Locking

When a SAS data set can be opened concurrently by more than one SAS session or by more than one statement or procedure within a single session, the level of locking is important. The level of locking determines how many sessions, procedures, or statements can read and write to the file at the same time. Some engines do not support locking at all, or do not support some of these levels. For details, see the engine documentation.

- **Library-level locking**
  - specifies that concurrent access is controlled at the library level. This locking restricts concurrent access to only one Update process to the library.

- **Member-level (data set) locking**
  - enables Read access to many sessions, statements, or procedures. This locking restricts all other access to the SAS data set when a session, statement, or procedure acquires Update or Output access.

- **Record-level (row) locking**
  - enables concurrent Read access and Update access to the SAS data set by more than one session, statement, or procedure. This locking prevents concurrent Update access to the same observation.

By default, SAS provides the greatest possible level of concurrent access, while guaranteeing the integrity of the data. Here are the ways of controlling the locking level:
Although controlling the access level yourself is usually not needed, the CNTLLEVE= data set option enables locking at the library, record, or member level.

- The LOCK statement acquires, lists, or releases an exclusive lock on an existing SAS file.

- Some SAS products, such as SAS/ACCESS and SAS/SHARE, contain engines that support enhanced session-management services and file-locking capabilities.

---

### Operating Environment Information

Engine availability and engine options are host dependent. See the SAS documentation for your operating environment:

- SAS Companion for UNIX Environments
- SAS Companion for Windows
- SAS Companion for z/OS

---

### Examples: Use a SAS Engine to Process SAS Data

#### Example: Assign the V9 Engine in a LIBNAME Statement

**Example Code**

This library assignment specifies the V9 engine.

```plaintext
libname myfiles v9 'c:\examples'; /*1*/
data myfiles.myclass; /*2*/
  set sashelp.class;
run;
```

1. The LIBNAME statement assigns the `myfiles` libref and the V9 engine to a physical location.
2. The DATA step creates the `myclass` data set in the `myfiles` library by copying the `class` data set in the `sashelp` library.
Example Code 13.1  SAS Log Showing a Successful V9 Library Assignment

```sas
1    libname myfiles v9 'c:\examples';
NOTE: Libref MYFILES was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\examples
2  !
3    data myfiles.myclass;
4       set sashelp.class;
5    run;
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set MYFILES.MYCLASS has 19 observations and 5 variables.
```

Key Ideas

- The LIBNAME statement is a common way to programmatically assign a library. A library assignment consists of a libref, an engine, a physical location, and options that are specific to the engine or environment.

- The shipped default Base SAS engine is BASE. In SAS® 9 and SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice.

See Also

- “Examples: Access Data by Using a Libref” on page 229
- “Elements of a Library Assignment” on page 222
- “Default Base SAS Engine (V9 Engine)” on page 262

Example: Assign the SPD Engine in a LIBNAME Statement

Example Code

The following LIBNAME statement for the SPD Engine is very similar to a LIBNAME statement for the V9 engine.

```sas
libname mylib spde 'c:\examples'; /*1*/
datapath=('d:\mydatadir'); /*2*/
indexpath=('e:\myindexdir'); /*3*/
```
This portion of the LIBNAME statement assigns the mylib libref and the SPD Engine to a primary path name. The first (and usually only) metadata file for a data set is always stored in the library's primary path.

Optionally, you can assign one or more path names in the DATAPATH= option to store data partitions. Otherwise, the data partition files are stored in the primary path.

Optionally, you can assign one or more path names in the INDEXPATH= option to store index files. Otherwise, the index files are stored in the primary path.

**Example Code 13.2** SAS Log Showing a Successful SPD Engine Library Assignment

```
libname mylib spde 'c:\examples'
  datapath=('d:\mydatadir')
  indexpath=('e:\myindexdir');
NOTE: Libref MYLIB was successfully assigned as follows:
  Engine:        SPDE
  Physical Name: c:\examples\
```

**Key Ideas**

- The SPD Engine is an alternative Base SAS engine.
- The SPD Engine is designed for high-speed processing of very large tables. The engine uses threads to read data very rapidly and in parallel, executing on multiple CPUs. Contributing to this performance is the partitioned file format, which can take advantage of distributed environments.
- Although SPD Engine stores a data set in multiple files, you can process an SPD Engine data set very similarly to a V9 engine data set. Most of the Base SAS language works very well with an SPD Engine data set. However, the engine supports some language elements that are specific to its processing and storage optimizations. For differences from V9 engine capabilities, see the documentation.

**See Also**

- *SAS Scalable Performance Data Engine: Reference*
- “Engine Characteristics” on page 260

---

**Example: Read and Write SAS Data in Hadoop by Using the SPD Engine**

**Example Code**

The following example assigns a Base SAS library to a Hadoop cluster.

```plaintext
options set=SAS_HADOOP_CONFIG_PATH='\myconfigpath';
```
The SET= system option defines environment variables for Hadoop. If these environment variables are already set (for example, in the SAS configuration file or SAS invocation), do not submit these lines of code. If these environment variables are not correctly set, then the following LIBNAME statement produces errors in the SAS log.

The LIBNAME statement assigns the mydata libref to the SPD Engine and a directory in the Hadoop cluster. The HDFS=YES argument specifies to connect to the Hadoop cluster that is defined in the Hadoop cluster configuration files. The ACCELWHERE=YES option requests that data subsetting be performed by a MapReduce program in the Hadoop cluster.

Key Ideas

- The SPD Engine is an alternative Base SAS engine that can read and write SAS data on a traditional file system or on Hadoop. The engine does not require you to license additional SAS products such as SAS/ACCESS, but you must be running a supported Hadoop distribution.
- Customers often choose Hadoop for low-cost storage of very large data. The distributed storage and processing of the SPD Engine works well with the Hadoop file system (HDFS). In addition, the engine can optimize most WHERE expressions by automatically submitting a MapReduce program in the Hadoop cluster.
- The engine can read SPD Engine data sets on Hadoop. After you use the engine to store a data set on Hadoop, you can use most of the Base SAS language for processing the data. However, the engine supports some language elements that are specific to its processing and storage on Hadoop.

See Also

- SAS SPD Engine: Storing Data in the Hadoop Distributed File System
- SAS Hadoop Configuration Guide for Base SAS and SAS/ACCESS

Example: Avoid Truncation by Using the CVP Engine with the V9 Engine

Example Code

To run this example, first create a data set named myclass as in "Example: Assign the V9 Engine in a LIBNAME Statement" on page 266. Run PROC CONTENTS to see the length of the variables:

```sas
libname myfiles v9 'c:\examples';
proc contents data=myfiles.myclass;
```
In the PROC CONTENTS output, notice the two character variables. Name has a length of 8, and Sex has a length of 1.

Output 13.1  PROC CONTENTS Showing Variable Lengths before Expansion

<table>
<thead>
<tr>
<th>Alphabetic List of Variables and Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

The example below uses the CVP engine with the V9 engine to expand the size of character variables. The CVP engine can help you avoid truncation if you copy a data set to an encoding that uses more bytes to represent the characters.

```
libname srclib cvp 'c:\examples' cvpengine=v9 cvpmult=2.5; /* 1 */
libname target v9 'c:\temp';                             /* 2 */
proc copy in=srclib out=target;                          /* 3 */
run;

proc contents data=target.myclass;                       /* 4 */
run;
```

1 This LIBNAME statement assigns the *srclib* library to the CVP engine and the location of the data that you want to copy. The CVPENGINE= option specifies the V9 engine as the underlying engine to process the data. The CVPMULT= option specifies a multiplication factor of 2.5 to expand all character variables.

2 This LIBNAME statement assigns the *target* library to contain the copied data.

3 The COPY procedure copies the *srclib* library to the *target* library. During the copy, the CVP engine expands the character variable lengths 2.5 times larger.

4 The CONTENTS procedure shows that the lengths of the character variables have been multiplied by 2.5:
   
   For Name, \( 8 \times 2.5 = 20 \).

   For Sex, \( 1 \times 2.5 = 2.5 \), which is 3 when rounded up to a whole number.
Key Ideas

- When you copy a data set to an encoding that uses more bytes to represent the characters, truncation might occur if the column length does not accommodate the larger character size. For example, a character might be represented in wlatin1 encoding as one byte but in UTF-8 as two bytes.
- If an error in the log states character data was lost during transcoding, it usually indicates that truncation has occurred. You can troubleshoot the error by using the CVP engine to expand the length of character variables.
- The CVPMULTIPLIER=2.5 value is usually sufficient to avoid truncation. If disk space is a concern, try a smaller value such as 1.5.
- The default CVPFORMATWIDTH=YES option expands the length for formats but does not affect user-defined formats. For user-defined formats, see “Example: Avoid Truncation in Formats When Migrating Catalogs” in SAS V9 LIBNAME Engine: Reference.
- If you copy a library to a different operating environment, or to a different character encoding, you probably want to specify options such as NOCLONE for PROC COPY. NOCLONE does not copy certain data set attributes, such as data representation and character encoding, so that the library is compatible in the target environment.
- As an alternative to the NOCLONE option, you can use the OVERRIDE= option to specify ENCODING= and OUTREP= options. This method keeps other data set attributes from the source library, so make sure that you want those attributes.
- If the library contains indexes or integrity constraints, then cross-environment data access (CEDA) is an important issue. If you have experienced truncation across environments, then SAS is probably using CEDA to access the files. Check the log for a CEDA message.
- PROC COPY does not copy indexes or integrity constraints under CEDA processing, but PROC MIGRATE does. However, PROC MIGRATE does not support the CVP engine to expand character column lengths. Therefore, if you want to migrate indexes or integrity constraints, you must copy the library with the CVP engine first and then migrate (in other words, a two-step process). See “Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process” in SAS V9 LIBNAME Engine: Reference.
- PROC COPY or the COPY statement of the DATASETS procedure do not preserve an audit trail.
Example: Load a SAS Data Set to a CAS Server

Example Code

The following example uses the DATA step to load a SAS data set into memory as a SAS Cloud Analytic Services (CAS) table.

```sas
cas casauto host="cloud.example.com" port=5570; /*1*/
libname mycas cas;                              /*2*/
data mycas.cars (promote=yes);                  /*3*/
  set sashelp.cars;
run;
proc contents data=mycas.cars;                  /*4*/
run;
```

1 The CAS statement starts a CAS session and specifies `casauto` as the CAS session name. Use your connection information in the `PORT=` and `HOST=` options.

2 The LIBNAME statement assigns the `mycas` libref to the CAS engine. The `SESSREF=` LIBNAME option is not specified, so the engine uses the `casauto` session.

3 The DATA step copies the SAS data set `sashelp.cars` to the CAS session. The `PROMOTE=YES` data set option promotes the table with global scope.

4 PROC CONTENTS shows the `mycas.cars` table is available on the CAS server for the duration of the session. After data is loaded into memory, subsequent steps can process the data in memory. Loading and processing are done in separate steps.
Key Ideas

- You can submit a LIBNAME statement that uses the CAS engine to connect your SAS session to a CAS session. You must have access to a CAS server and an existing CAS session.

- The CAS LIBNAME engine with the DATA step is one way of loading SAS data to the CAS server as an in-memory table. Other methods might be more efficient for large tables.

- After you load data to the CAS server, you can execute SAS procedures or the DATA step from your SAS session by referencing the SAS libref and table name. You do not process the table in memory in the same DATA step that you use to load the table into memory. Loading and processing are done in separate steps.

- Tables are not automatically saved when they are loaded to a caslib. You can use the CASUTIL procedure to save tables. Native CAS tables have the file extension .sashdat.

See Also

- “The DATA Step and CAS” in SAS Cloud Analytic Services: DATA Step Programming
- “Example: Load SAS Data to CAS by Using the CASUTIL Procedure” in SAS V9 LIBNAME Engine: Reference
Example: Set a Default Engine

Example Code

The following invocation command is an example for the Windows file system. The path to the executable might be different in your deployment.

In this example, the ENGINE system option assigns the SPD Engine as the default for new data sets.

```
"c:\program files\SASHome\SASFoundation\9.4\sas.exe" -engine spde
```

Key Ideas

- The ENGINE system option cannot be set interactively. You must specify ENGINE in a configuration file, at invocation, or in an environment variable.
- The shipped default Base SAS engine is BASE. In SAS®9 and SAS® Viya®, the BASE engine is an alias for the V9 engine. If you do not specify an engine name when you create a new library, and if you have not specified the ENGINE system option, then the V9 engine is automatically selected. If the library location already contains SAS files, then SAS might be able to assign the correct engine based on those files. For example, if the location contains V9 data sets only, then SAS assigns the V9 engine. However, if a library location contains a mix of different engine files, then SAS might not assign the engine you want. Therefore, specifying the engine is a best practice.

See Also

- “ENGINE= System Option” in SAS System Options: Reference
- “Default Base SAS Engine (V9 Engine)” on page 262
- “Default Engine for a Mixed Library” in SAS V9 LIBNAME Engine: Reference
- “Operating Environment Information” on page 266
Examples: Use a SAS Engine to Process External Data

Example: Read a Comma-Delimited File

Example Code

In this example, the IMPORT procedure imports a file that contains comma-separated values.

```sas
filename chol temp;                 /* 1 */
proc http                           /* 2 */
  url="http://support.sas.com/documentation/onlinedoc/viya/
      exampledatasets/cholesterol.csv"
  out=chol;
quit;
proc import datafile=chol           /* 3 */
  out=work.mycholesterol
  dbms=csv
  replace;
run;
proc print data=work.mycholesterol; /* 4 */
run;
```

1. The FILENAME statement assigns the chol fileref. The TEMP option specifies that the file is temporary, so a path name is not necessary.
2. The HTTP procedure specifies the URL of the cholesterol.csv input file. The data is written to the chol fileref.
3. PROC IMPORT reads the comma-delimited data and creates the mycholesterol data set.
4. The PRINT procedure prints the data set. The output shows that the file was imported correctly, including the column names.

The output shows that the file was imported correctly, including the column names.
Base SAS software can import and export some external text files without using a SAS/ACCESS engine. These files are usually referred to as raw data or delimited data. The DATA step or PROC IMPORT can read data from a text file and provide that data to the V9 engine for output to a SAS data set.

If you want Base SAS to read or write a Microsoft Excel file, the file must have a .csv extension. To import a file that has an Excel file extension of .xls or .xlsx, you must have a license to SAS/ACCESS Interface to PC Files. In Excel, you can save an Excel file as a .csv file.

If you create a SAS data set from an Excel file, the data is static and does not change to reflect the underlying Excel data. If you want to keep data in Excel as an ongoing data store that you can query from SAS, then use the XLSX or EXCEL engine. You must have a license to SAS/ACCESS Interface to PC Files.

See Also

- “Default Base SAS Engine (V9 Engine)” on page 262
- Chapter 15, “Raw Data,” on page 315
- “IMPORT Procedure” in Base SAS Procedures Guide
- “Comparing SAS LIBNAME Engines for PC Files Data” in SAS/ACCESS Interface to PC Files: Reference
Example: Read Microsoft Excel Data by Using a SAS/ACCESS Engine

Example Code

This example uses the XLSX engine and PROC IMPORT to import Excel data. To run this example, start Microsoft Excel and open the cholesterol.csv file that you used in “Example: Read a Comma-Delimited File” on page 275. (The cholesterol.csv file can be downloaded from http://support.sas.com/documentation/onlinedoc/viya/exampledatasets/cholesterol.csv.) In Excel, save the file with an .xlsx extension.

```
options validvarname=v7;                 /* 1 */
proc import datafile='C:\examples\cholesterol.xlsx' /* 2 */
   dbms=xlsx                                    /* 3 */
   out=work.mycholesterol                        /* 4 */
   replace;                                       /* 5 */
run;
```

1 The VALIDVARNAME=V7 statement forces SAS to convert spaces to underscores when it converts column names to variable names.

2 The DATAFILE= option specifies the path for the input file.

3 The DBMS= option specifies the type of data to import. When you import an Excel workbook, specify DBMS=XLSX. PROC IMPORT calls the XLSX engine, so you do not need to submit a LIBNAME statement.

4 The OUT= option identifies the output SAS data set.

5 The REPLACE option overwrites an existing SAS data set.

Key Ideas

- The XLSX engine enables you to read data directly from Excel .xlsx files. You must have a license to SAS/ACCESS Interface to PC Files. The XLSX engine supports connections to Microsoft Excel 2007, 2010, and later files.

- SAS/ACCESS Interface to PC Files also provides the EXCEL engine for earlier releases of Excel.

- Excel has different naming conventions than SAS. A best practice for reading Excel data is to set VALIDVARNAME=V7. This option converts spaces to underscores, and truncates names greater than 32 characters.

- Date values are automatically converted to numeric SAS date values.

See Also

- SAS/ACCESS Interface to PC Files: Reference
Example: Create Data in a DBMS by Using a SAS/ACCESS Engine

Example Code

In this example, a SAS DATA step creates a Teradata table. To run this example, you must license a SAS/ACCESS interface.

```
libname mytdata teradata server=mytera user=myid password=mypw; /*1*/
data mytdata.grades;                                            /*2*/
   input student $ test1 test2 final;
   datalines;
   Fred  66  80  70
   Wilma 97  91  98
;
proc datasets library=mytdata;                                  /*3*/
run;
quit;
```

1 The LIBNAME statement specifies the `mytdata` libref and `TERADATA`, which is the engine nickname for SAS/ACCESS Interface to Teradata. The statement also specifies connection options for Teradata. Change these options to specify your SAS/ACCESS connection values and any other options you need.

2 The DATA step creates a table named `grades`. The table is in the Teradata DBMS and is not a SAS data set.

3 The PROC DATASETS output for the `mytdata` library shows that the engine is Teradata. For the `grades` table, the SAS member type is `DATA` and the DBMS member type is `TABLE`.

Output 13.5 PROC DATASETS Output Showing the DBMS Directory Information

<table>
<thead>
<tr>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libref</td>
</tr>
<tr>
<td>Engine</td>
</tr>
<tr>
<td>Physical Name</td>
</tr>
<tr>
<td>Schema/User</td>
</tr>
</tbody>
</table>

Output 13.6 PROC DATASETS Output Showing the Library Contents for DBMS Content

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Member Type</th>
<th>DBMS Member Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>grades</td>
<td>DATA</td>
<td>TABLE</td>
</tr>
</tbody>
</table>
Key Ideas

- If you license a SAS/ACCESS interface for your DBMS data, you can submit a LIBNAME statement in SAS to run SAS code against the DBMS data. SAS can create or process a DBMS table as if it were a SAS data set.
- Most SAS/ACCESS engines fully support the Base SAS language. A few Base SAS features, such as catalogs, are specific to the V9 engine and are not available in SAS/ACCESS engines. The SAS/ACCESS engines support some language elements that are specific to the data source. For details, see the documentation.
- Some SAS/ACCESS interfaces are case sensitive. You might need to change the case of table or column names to comply with the requirements of your DBMS.

See Also

- SAS/ACCESS documentation
- “DATA Statement” in SAS DATA Step Statements: Reference

Example: Embed a SAS/ACCESS LIBNAME Statement in a PROC SQL View

Example Code

The following example embeds a SAS/ACCESS LIBNAME statement in a PROC SQL view.

```sas
libname viewlib v9 '/mydata';        /* A */
proc sql;
  create view viewlib.mygrades as   /* B */
    select *                         /* C */
      from mytdda.data grades        /* C */
      using libname mytdda teradata  /* C */
      server=mytera
      user=myid password=mypw;       /* D */
quit;
proc print data=viewlib.mygrades noobs;  /* E */
run;
```

1. The V9 engine LIBNAME statement assigns the viewlib libref to the location where the SAS view will be stored.
2. The CREATE VIEW statement in the SQL procedure creates the mygrades view in the viewlib library.
3. The mytdda.data grades table is referenced in the view.
4. The USING argument embeds the LIBNAME statement for the SAS/ACCESS Interface to Teradata engine.
The PRINT procedure executes the `viewlib.mygrades` view, which references the `mytddata.grades` table by using the embedded LIBNAME statement.

Here is the PROC PRINT output.

**Output 13.7 PROC PRINT of viewlib.mygrades**

<table>
<thead>
<tr>
<th>student</th>
<th>test1</th>
<th>test2</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilma</td>
<td>97</td>
<td>91</td>
<td>99</td>
</tr>
<tr>
<td>Fred</td>
<td>66</td>
<td>80</td>
<td>70</td>
</tr>
</tbody>
</table>

**Key Ideas**

- Most SAS/ACCESS engines require several connection options. Embedding the LIBNAME statement enables you to store DBMS connection information in the view for ease of use. However, if the connection information is expected to change (for example, a different user ID), then this practice is not recommended.
- To embed a LIBNAME statement in a PROC SQL view, specify the USING clause of the CREATE VIEW statement.
- When PROC SQL executes the SAS view, the SELECT statement assigns the libref and establishes the connection to the DBMS. The scope of the libref is local to the SAS view and does not conflict with identically named librefs that might exist in the SAS session. When the query finishes, the connection is terminated and the libref is deassigned.
- When you create a V9 engine view of a DBMS table, the data stays in the DBMS. You can run a procedure or DATA step against a view at any time to get the latest data from the underlying data source.

**See Also**

- "LIBNAME Statement Syntax for Relational Databases Statement" in SAS/ACCESS for Relational Databases: Reference
- "CREATE VIEW Statement" in SAS SQL Procedure User’s Guide
- "SAS Views of DBMS Data" in SAS/ACCESS for Relational Databases: Reference
- "SAS Views" in SAS V9 LIBNAME Engine: Reference
- SAS/ACCESS documentation
Example: Access DBMS Data by Using the SQL Pass-Through Facility

Example Code

This PROC SQL example uses the SQL pass-through facility to send a query to a Teradata table.

```sql
proc sql;
   connect to teradata as myconn (server=mytera user=myid password=mypw); /*1*/
   select *
   from connection to myconn /*2*/
       (select *
           from grades
           where final gt 90);
   disconnect from myconn; /*3*/
quit;
```

1 The CONNECT statement establishes a connection to the DBMS.
2 The CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement retrieves data directly from a DBMS. The DBMS-specific query is enclosed in parentheses.
3 The DISCONNECT statement terminates the connection to the DBMS.

Here is the result of the PROC SQL query.

**Output 13.8  Output of PROC SQL Pass-Through Facility**

<table>
<thead>
<tr>
<th>student</th>
<th>test1</th>
<th>test2</th>
<th>final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wilma</td>
<td>97</td>
<td>91</td>
<td>98</td>
</tr>
</tbody>
</table>

Key Ideas

- The SQL pass-through facility is an extension of PROC SQL that enables you to use DBMS-specific SQL syntax instead of SAS SQL syntax.
- PROC SQL provides several statements such as CONNECT that are specific to the SQL pass-through facility.
- You can use SQL pass-through facility statements in a PROC SQL query or store them in a PROC SQL view.

See Also

- “SQL Pass-Through Facility” in SAS/ACCESS for Relational Databases: Reference
Example: Import XML Data by Using the XMLV2 Engine

Prerequisites
To run the XMLV2 example code, first create a file named nhl.xml that contains the following XML data. Store this file in a location that is accessible by your SAS session.

```xml
<?xml version="1.0" encoding="iso-8859-1" ?>
<NHL>
  <CONFERENCE> Eastern
    <DIVISION> Southeast
      <TEAM name="Thrashers" abbrev="ATL" />
      <TEAM name="Hurricanes" abbrev="CAR" />
      <TEAM name="Panthers" abbrev="FLA" />
      <TEAM name="Lightning" abbrev="TB" />
      <TEAM name="Capitals" abbrev="WSH" />
    </DIVISION>
  </CONFERENCE>

  <CONFERENCE> Western
    <DIVISION> Pacific
      <TEAM name="Stars" abbrev="DAL" />
      <TEAM name="Kings" abbrev="LA" />
      <TEAM name="Ducks" abbrev="ANA" />
      <TEAM name="Coyotes" abbrev="PHX" />
      <TEAM name="Sharks" abbrev="SJ" />
    </DIVISION>
  </CONFERENCE>
</NHL>
```

Example Code
The following example uses the XMLV2 engine to import XML data. You can process XML data as SAS data sets in memory, without creating permanent SAS data sets.

```sas
filename nhl 'c:\example\nhl.xml';            /* 1 */
filename map 'c:\output\nhlgenerate.map';     /* 2 */
libname nhl xmlv2 automap=replace xmlmap=map; /* 3 */
proc print data=nhl.team noobs;               /* 4 */
  var TEAM_name TEAM_abbrev;
run;
```
1 The first FILENAME statement assigns the file reference nhl to the physical location of the XML document nhl.xml that will be imported.

2 The second FILENAME statement assigns the file reference map to a physical location to store the XMLMap nhlgenerate.map that will be generated by SAS.

3 The LIBNAME statement assigns the nhl library to the XMLV2 engine. The AUTOMAP=REPLACE option specifies to automatically generate an XMLMap and to replace the XMLMap if it already exists. The XMLMAP= option specifies the fileref of the XMLMap.

4 PROC PRINT produces output, verifying that the import was successful.

Here is the output from PROC PRINT.

**Output 13.9**  PROC PRINT of Data Set Imported from nhl.xml

<table>
<thead>
<tr>
<th>TEAM_name</th>
<th>TEAM_abbrev</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thrashers</td>
<td>ATL</td>
</tr>
<tr>
<td>Hurricanes</td>
<td>CAR</td>
</tr>
<tr>
<td>Panthers</td>
<td>FLA</td>
</tr>
<tr>
<td>Lightning</td>
<td>TB</td>
</tr>
<tr>
<td>Capitals</td>
<td>WSH</td>
</tr>
<tr>
<td>Stars</td>
<td>DAL</td>
</tr>
<tr>
<td>Kings</td>
<td>LA</td>
</tr>
<tr>
<td>Ducks</td>
<td>ANA</td>
</tr>
<tr>
<td>Coyotes</td>
<td>PHX</td>
</tr>
<tr>
<td>Sharks</td>
<td>SJ</td>
</tr>
</tbody>
</table>

**Key Ideas**

- The Base SAS XMLV2 engine provides sequential access to read (import) XML data. The engine can also write (export) SAS data sets as XML data.
- The engine creates temporary data sets in memory. To create permanent data sets, reference the temporary data sets in the DATA step or in PROC COPY.
- One XML file can contain multiple related data sets. See the documentation for an example that imports an XML document as multiple SAS data sets.
- The engine can automatically create an XMLMap of the data in an XML file. To save the map, specify the MAP= option and AUTOMAP=REPLACE in the LIBNAME statement. Then you can open the map in a text editor to customize it. After you edit a map, specify the MAP= option and AUTOMAP=REUSE to avoid overwriting your customizations.
- You can also generate and customize an XMLMap by using SAS XML Mapper, a graphical interface.
See Also

- SAS XMLV2 and XML LIBNAME Engines: User’s Guide
- “FILENAME Statement” in SAS Global Statements: Reference

Example: Import JSON Data by Using the JSON Engine

Example Code

The following example uses the SAS JSON engine to read JSON data. The example creates temporary, in-memory SAS data sets for analysis.

To run this example, first copy the JSON code for example.json from "LIBNAME Statement: JSON Engine" in SAS Global Statements: Reference. Create the file in a location that is accessible by your SAS session.

libname mydata json 'c:\examples\example.json';
proc datasets lib=mydata;
run;
quit;

Here is the PROC DATASETS output, showing the SAS data sets imported from example.json.

Output 13.10  PROC DATASETS Output of SAS Data Sets Imported from JSON
Key Ideas

- The Base SAS JSON engine provides read-only, sequential access to JSON data.
- The engine creates temporary data sets in memory. To create permanent data sets, reference the temporary data sets in the DATA step or in PROC COPY.
- One JSON file can contain multiple related data sets. See the documentation for an example that merges the imported data sets into a single data set.
- The engine automatically creates a map of the data in a JSON file. To save the map, specify the MAP= option and AUTOMAP=CREATE in the LIBNAME statement. Then you can open the map in a text editor to customize it. After you edit a map, specify the MAP= option and AUTOMAP=REUSE to avoid overwriting your customizations.
- To export a JSON file from a SAS data set, use the JSON procedure.

See Also

- “LIBNAME Statement: JSON Engine” in SAS Global Statements: Reference
- “JSON Procedure” in Base SAS Procedures Guide
Definitions for SAS Data Sets

Definition of a SAS Data Set

A SAS data set is a tabular set of data whose contents are in a SAS file format. A SAS data set contains the data and information about the data (metadata).

For more information about SAS views, see “Definitions for SAS Views” on page 353.
Parts of a SAS Data Set

When you create a SAS data set using the V9 engine, the data values and the metadata are stored in a single file. When you create a SAS data set using the SAS Scalable Performance Data Engine, the data values and the metadata are stored in multiple separate files. Metadata is also called descriptor information. For specific information about how data is stored based on the engine, see the following documentation:

- For the V9 engine, see Parts of a SAS data set.
- For the SPD engine, see “Differences between the Default Base SAS Engine Data Sets and the SPD Engine Data Sets” in SAS Scalable Performance Data Engine: Reference and “Organizing SAS Data Using the SPD Engine” in SAS Scalable Performance Data Engine: Reference.

To view the descriptor information (metadata) about a SAS data set, you can use the CONTENTS procedure or the CONTENTS Statement in the DATASETS procedure:

```sas
proc contents data=sashelp.air;
run;
```

If a data set is sorted, then additional sort information is added to the data set’s metadata. For more information about sorting SAS data sets, see BY-Group Processing on page 363.

Managing SAS Data Sets

Ways to Read and Create SAS Data Sets

The following table contains some common tasks for managing SAS data sets. This table contains a small sampling of ways that you can manipulate and manage SAS data sets. For more information about SAS engines, see Chapter 13, “SAS Engines,” on page 257.

**Table 14.1  Ways to Manage Data Sets**

<table>
<thead>
<tr>
<th>Task</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read and Create a SAS data set on page 294</td>
<td>V9 Engine LIBNAME statement</td>
</tr>
<tr>
<td>Copy a SAS data set</td>
<td>DATASETS procedure</td>
</tr>
<tr>
<td>Get information about a SAS data set</td>
<td></td>
</tr>
</tbody>
</table>
Many of the methods that you use to manage data sets are identical to the ones that you use to manage SAS libraries on page 250.

Ways to Manage Variables in SAS Data Sets

By default, SAS writes all the variables and all the rows from the input data set to the output data set. You can control which variables and rows that SAS reads and writes by using any of the following language elements:

Table 14.2  Ways to Control the Reading and Writing of Variables and Rows

<table>
<thead>
<tr>
<th>Category</th>
<th>Task</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control variables</td>
<td>Drop variables from output</td>
<td>DROP statement and DROP= data set option</td>
</tr>
<tr>
<td>(columns)</td>
<td>Drop variables from input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Keep variables in the output data set</td>
<td>KEEP statement and KEEP= data set option</td>
</tr>
<tr>
<td></td>
<td>Keep variables in the input data set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rename variables</td>
<td>RENAME statement and RENAME= data set option</td>
</tr>
<tr>
<td></td>
<td>Rename variables in the output data set output</td>
<td></td>
</tr>
<tr>
<td>Control rows</td>
<td>Conditionally select rows by using the WHERE</td>
<td>WHERE statement</td>
</tr>
<tr>
<td></td>
<td>statement</td>
<td>WHERE= data set option</td>
</tr>
<tr>
<td></td>
<td>Conditionally select rows by using the IF</td>
<td>subsetting IF statement</td>
</tr>
<tr>
<td></td>
<td>statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delete rows based on a condition</td>
<td>DELETE statement</td>
</tr>
<tr>
<td></td>
<td>Remove a row</td>
<td>REMOVE statement</td>
</tr>
<tr>
<td></td>
<td>Write the current row to the output data set</td>
<td>OUTPUT statement</td>
</tr>
<tr>
<td></td>
<td>Write the current row when a specified condition is true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create multiple rows from one line of input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create one row from multiple lines of input</td>
<td></td>
</tr>
</tbody>
</table>
### Table 14.3  Ways to Control the Reading and Writing of Variables and Rows

<table>
<thead>
<tr>
<th>Category</th>
<th>Task</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directly access rows</td>
<td>Modify rows based on row number</td>
<td>POINT= statement option</td>
</tr>
<tr>
<td></td>
<td>“Example: Read an Observation Directly Using the POINT= Option” on page 308</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modify rows based on row number</td>
<td>KEY= statement option</td>
</tr>
<tr>
<td></td>
<td>Read a range of rows using the FIRSTOBS= and OBS= data set options</td>
<td>FIRSTOBS= data set option (V9 engine only)</td>
</tr>
<tr>
<td></td>
<td>Process a specific row first in the data set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specify when to stop processing rows</td>
<td>OBS= data set option (V9 engine only)</td>
</tr>
<tr>
<td>Control variables</td>
<td>Drop variables from output</td>
<td>DROP statement and DROP= data set option</td>
</tr>
<tr>
<td></td>
<td>Drop variables from input</td>
<td>V9 Engine LIBNAME statement</td>
</tr>
<tr>
<td></td>
<td>Keep variables in the output data set</td>
<td>KEEP statement and KEEP= data set option</td>
</tr>
<tr>
<td></td>
<td>Keep variables in the input data set</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Rename variables</td>
<td>RENAME statement and RENAME= data set option</td>
</tr>
<tr>
<td></td>
<td>Rename variables in the output data set output</td>
<td></td>
</tr>
</tbody>
</table>
### Ways to Read Rows in SAS Data Sets

You can access rows in data sets either sequentially or directly.

**Sequential access**
reads rows sequentially, in the order in which they appear in the physical file. The SET, MERGE, UPDATE, and MODIFY statements read rows sequentially by

------------
<table>
<thead>
<tr>
<th>Category</th>
<th>Task</th>
<th>More Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control rows</td>
<td>Conditionally select rows by using the WHERE statement</td>
<td>WHERE statement and subsetting IF statement</td>
</tr>
<tr>
<td></td>
<td>Conditionally select rows by using the IF statement</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Process conditionally selected data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Delete rows based on a condition</td>
<td>DELETE statement</td>
</tr>
<tr>
<td></td>
<td>Remove a row</td>
<td>REMOVE statement</td>
</tr>
<tr>
<td></td>
<td>Write the current row to the output data set</td>
<td>OUTPUT statement</td>
</tr>
<tr>
<td></td>
<td>Write the current row when a specified condition is true</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create multiple rows from one line of input</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Create one row from multiple lines of input</td>
<td></td>
</tr>
<tr>
<td>Directly access rows</td>
<td>Modify rows based on row number</td>
<td>POINT= statement option</td>
</tr>
<tr>
<td></td>
<td>Modify rows based on row number</td>
<td>KEY= statement option</td>
</tr>
<tr>
<td></td>
<td>Read a range of rows using the FIRSTOBS= and OBS= data set options</td>
<td>FIRSTOBS= data set option</td>
</tr>
<tr>
<td></td>
<td>Process a specific row first in the data set</td>
<td>FIRSTOBS= system option</td>
</tr>
<tr>
<td></td>
<td>Specify when to stop processing rows</td>
<td>OBS= data set option</td>
</tr>
</tbody>
</table>

**Directly access rows**

**Modify rows based on row number**

**Modify rows based on row number**

**Read a range of rows using the FIRSTOBS= and OBS= data set options**

**Process a specific row first in the data set**

**Specify when to stop processing rows**

**Specify when to stop processing rows**

### Ways to Read Rows in SAS Data Sets

You can access rows in data sets either sequentially or directly.

**Sequential access**
reads rows sequentially, in the order in which they appear in the physical file. The SET, MERGE, UPDATE, and MODIFY statements read rows sequentially by...
default. The SAS functions OPEN, FETCH, and FETCHOBS also read rows sequentially.

**Direct access**

**by row number**

In the DATA step, to access rows directly by their row number, use the \texttt{POINT=} option in the \texttt{SET} or \texttt{MODIFY} statements. The \texttt{POINT=} option names a temporary variable whose current value determines which row that a \texttt{SET} or \texttt{MODIFY} statement reads.

You can subset rows from one data set and combine them with rows from another data set by using direct access methods, as follows:

```sas
data south;
  set revenue;
  if region=4;
  set expense point=_n_;
run;
```

**by index**

To directly access rows that are based on the values of one or more specified variables, you must first create an index for the variables. An index is a separate structure that contains the data values of the key variable or variables, paired with a location identifier for the rows that contain the value.

Once the index is created, you can then use the DATA step with the \texttt{KEY=} option in the \texttt{SET} or \texttt{MODIFY} statement to directly access rows based on the values of the indexed variable.

For example, suppose that you need to match information in one data set with a specific value in a second data set. If the second data set is properly indexed, you can use the \texttt{KEY=} option in the \texttt{SET} statement to perform a “table lookup” on the indexed table to combine only those rows with the first data set.

```sas
data combine;
  set inventory(keep=partno instock price);
  set partcode(keep=partno desc) key=partno;
run;
```

For another example, see “Modifying Observations Located by an Index” in *SAS DATA Step Statements: Reference*.

Here are some methods for creating indexes:

**Table 14.4** Ways to Create Indexes

<table>
<thead>
<tr>
<th>Language Element Used</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{INDEX=} data set option</td>
<td>Create an Index Using the \texttt{INDEX=} Data Set Option</td>
</tr>
<tr>
<td>\texttt{INDEX CREATE} statement (\texttt{DATASETS} procedure)</td>
<td>“Modifying SAS Data Sets” in <em>Base SAS Procedures Guide</em></td>
</tr>
<tr>
<td>\texttt{CREATE INDEX} statement (\texttt{SQL} procedure)</td>
<td>“Creating an Index” in <em>SAS SQL Procedure User’s Guide</em></td>
</tr>
</tbody>
</table>
Indexes can be created on SAS data sets that are created using the Base SAS V9 Engine. For more information about creating indexes on SAS data sets, see “Indexes” in SAS V9 LIBNAME Engine: Reference.

Rules for Naming SAS Data Sets

- “Data Set Names” on page 53
- Summary of Rules for Naming SAS Data Sets on page 57
- “Special Data Set Names” on page 55
- “Example: Create a SAS Data Set Name Containing a Special Character” on page 60 and “Example: Create a Two-Level Data Set Name” on page 65

Data Set Name Lists

A data set name list is a shorthand way of specifying multiple SAS data set names in a statement or procedure.

A data set name list can be either a numbered range list or a name prefix list.

Numbered range list

data combine;
  set sales0 sales1 sales2 sales3 sales4 sales5;
run;

data combine;
  set sales0:sales5;
run;

- data set names in a numbered range list have the same name except for the last character or characters, which are consecutive numbers.

- data set names in a numbered range list can begin with any number and end with any number as long as the numbers are consecutive. For example, the following SET statements refer to the same data sets:

  proc datasets;
  copy in=work out=mysas;
  select d1-d3;
  run; quit;

  Note: If the numeric suffix contains leading zeros, the number of digits in the suffix of the last data set name must be greater than or equal to the number of digits in the first data set name. For example, the data set lists sales001–sales99 causes an error. The data set list sales001–sales999 is valid.

Name prefix list (colon list)

A name prefix list is a shorthand way of specifying multiple data set names by using a single name prefix followed by a colon. The shorthand (prefix) name must begin with the same character or characters as the names that it
A colon (:) is used to designate the end of the character string prefix. For example, the following two DATA steps refer to the same data set:

```
data combine;
  set sales5 sales3 sales2 sal sal_weekly sales_monthly;
run;
```
```
data combine;
  set sal:
run;
```

Data set name lists are typically used in these SAS language elements:

**Statements in PROC DATASETS:**
- COPY statement (“Copying Selected SAS Files” in Base SAS Procedures Guide)
- SELECT statement (“Selecting Several Like-Named Files” in Base SAS Procedures Guide)
- EXCLUDE statement (“Excluding Several Like-Named Files” in Base SAS Procedures Guide)
- DELETE statement
- REPAIR statement
- SORTEDBY option in the MODIFY statement

**DATA step statements:**
- MERGE statement, (Using Data Set Lists with MERGE)
- SET statement

See “SAS Variable Lists” on page 91 for more information about creating character and numeric variable lists in SAS code.

---

**See Also**

“Definitions for SAS Data Sets” on page 287

---

### Examples: Create and Read SAS Data Sets

#### Example: Read a Single SAS Data Set

**Example Code**

This example reads a SAS data set from the Sashelp library and writes the output to the SAS Work library.

The SET statement reads the Sashelp.shoes data set into the DATA step where it is processed by the WHERE statement. The WHERE statement selects only those observations that contain a value greater than 500,000 for the variable sales. The
DATA step then writes the output to the data set that is specified in the DATA statement (work.shoes).

```sas
data work.shoes;
set sashelp.shoes;
where sales>500000;
run;
proc print data=shoes; run;
```

Output 14.1  PROC PRINT Output for the Work.shoes Data Set Read from the Sashelp Library

<table>
<thead>
<tr>
<th>Obs</th>
<th>Region</th>
<th>Product</th>
<th>Subsidiary</th>
<th>Stores</th>
<th>Sales</th>
<th>Inventory</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Canada</td>
<td>Men's Dress</td>
<td>Vancouver</td>
<td>28</td>
<td>$757,798</td>
<td>$1,847,559</td>
<td>$16,833</td>
</tr>
<tr>
<td>2</td>
<td>Canada</td>
<td>Slipper</td>
<td>Vancouver</td>
<td>27</td>
<td>$706,513</td>
<td>$2,520,685</td>
<td>$21,247</td>
</tr>
<tr>
<td>3</td>
<td>Canada</td>
<td>Women's Dress</td>
<td>Vancouver</td>
<td>21</td>
<td>$756,347</td>
<td>$2,503,387</td>
<td>$19,378</td>
</tr>
<tr>
<td>4</td>
<td>Central America/Caribbean</td>
<td>Men's Casual</td>
<td>Kingston</td>
<td>28</td>
<td>$576,112</td>
<td>$1,159,556</td>
<td>$20,065</td>
</tr>
<tr>
<td>5</td>
<td>Middle East</td>
<td>Men's Casual</td>
<td>Tel Aviv</td>
<td>11</td>
<td>$1,290,717</td>
<td>$2,881,605</td>
<td>$57,362</td>
</tr>
<tr>
<td>6</td>
<td>Western Europe</td>
<td>Women's Casual</td>
<td>Copenhagen</td>
<td>26</td>
<td>$502,636</td>
<td>$1,110,412</td>
<td>$17,448</td>
</tr>
</tbody>
</table>

Key Ideas

- The SET statement reads SAS data sets into the DATA step for processing. You can also use the MERGE statement, the MODIFY statement, and the UPDATE statement to read SAS data sets into a DATA step.

- The DATA statement writes out SAS data sets that have been processed by the DATA step.

- If you do not specify a location for the output data set, the DATA statement automatically writes the output to the SAS Work library. Data sets written to the Work library are saved only for the duration of the current SAS session. To specify a permanent output location, you must create a SAS library (libref) by using the LIBNAME statement. See Chapter 12, “SAS Libraries,” on page 219 for more information about libraries in SAS.

- By default, the DATA step reads observations from a SAS data set using sequential access. For more information about sequential and direct data access, see “Ways to Read Rows in SAS Data Sets” on page 291.

See Also

- SET statement, DATA statement, and “WHERE Statement” in SAS DATA Step Statements: Reference

- “Sashelp Library” on page 228 in SAS Programmer’s Guide: Essentials

- “Ways to Read Rows in SAS Data Sets” on page 291
Example: Read Multiple SAS Data Sets

Example Code

This example reads in three data sets from the Sashelp library and then concatenates them into a single output data set named `concat`. Since a SAS library or output location is not specified, the output data set, `concat`, is temporarily saved in the SAS Work library.

```sas
data concat;
   set sashelp.nvst1 sashelp.nvst2 sashelp.nvst3;
run;
proc print data=concat; run;
```

The output data set consists of observations from all three data sets. The order in which the data sets are concatenated in the output data set is based on how the data sets are listed in the SET statement. Observations from `sashelp.nvst1` are first, followed by observations from `sashelp.nvst2`, followed by observations from `sashelp.nvst3`.

Here is the PROC PRINT output for the data set `concat`, annotated to show how the DATA step concatenates multiple input data sets:
**Output 14.2**  PROC PRINT Output for Data Set Concat

<table>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01JAN1997</td>
<td>-30000</td>
</tr>
<tr>
<td>2</td>
<td>01JAN1998</td>
<td>7500</td>
</tr>
<tr>
<td>3</td>
<td>01JAN1999</td>
<td>7500</td>
</tr>
<tr>
<td>4</td>
<td>01JAN2000</td>
<td>7500</td>
</tr>
<tr>
<td>5</td>
<td>01JAN2001</td>
<td>7500</td>
</tr>
<tr>
<td>6</td>
<td>01JAN2002</td>
<td>7500</td>
</tr>
<tr>
<td>7</td>
<td>01JAN1997</td>
<td>-60000</td>
</tr>
<tr>
<td>8</td>
<td>01JAN1998</td>
<td>13755</td>
</tr>
<tr>
<td>9</td>
<td>01JAN1999</td>
<td>13755</td>
</tr>
<tr>
<td>10</td>
<td>01JAN2000</td>
<td>13755</td>
</tr>
<tr>
<td>11</td>
<td>01JAN2001</td>
<td>13755</td>
</tr>
<tr>
<td>12</td>
<td>01JAN2002</td>
<td>23755</td>
</tr>
<tr>
<td>13</td>
<td>01JAN1997</td>
<td>-20000</td>
</tr>
<tr>
<td>14</td>
<td>01JAN1998</td>
<td>5000</td>
</tr>
<tr>
<td>15</td>
<td>01JAN1999</td>
<td>5000</td>
</tr>
<tr>
<td>16</td>
<td>01JAN2000</td>
<td>5000</td>
</tr>
<tr>
<td>17</td>
<td>01JAN2001</td>
<td>5000</td>
</tr>
<tr>
<td>18</td>
<td>01JAN2002</td>
<td>5000</td>
</tr>
</tbody>
</table>

Here is the log output:

**Output 14.3**  Log Output for Reading Multiple SAS Data Sets

NOTE: There were 6 observations read from the data set SASHELP.NVST1.
NOTE: There were 6 observations read from the data set SASHELP.NVST2.
NOTE: There were 6 observations read from the data set SASHELP.NVST3.
NOTE: The data set WORK.CONCAT has 18 observations and 2 variables.
Key Ideas

- You can read from multiple SAS data sets and combine and modify data in different ways. See “Summary of Ways to Combine SAS Data Sets” on page 406 for more information.
- The SET statement reads SAS data sets into the DATA step for processing. You can also use the MERGE statement, the MODIFY statement, and the UPDATE statement to read SAS data sets into a DATA step.
- The DATA statement writes out SAS data sets that have been processed by the DATA step.
- If you do not specify a location for the output data set, the DATA statement automatically writes the output to the SAS Work library. Data sets written to the Work library are saved only for the duration of the current SAS session. To specify a permanent output location, you must create a SAS library (libref) by using the LIBNAME statement. See Chapter 12, “SAS Libraries,” on page 219 for more information about libraries in SAS.
- By default, the DATA step reads observations from a SAS data set using sequential access. For more information about sequential and direct data access, see “Ways to Read Rows in SAS Data Sets” on page 291.

See Also

- “SET Statement” in SAS DATA Step Statements: Reference
- “WHERE Statement” in SAS DATA Step Statements: Reference
- “Sashelp Library” on page 228

Example: Read a SAS Data Set from a User-defined Library

Example Code

This example reads the permanent data set, quakes, from the user-defined library, mysas, and then writes it out as quakes_mag in the same library.

To set up this example, first use the LIBNAME statement to create a user-defined library (libref), mysas. Then, create the data set sashelp.quakes and specify mysas.quakes as the output data set.

```sas
libname mysas "<path-to-file>"; /*1*/
data mysas.quakes; /*2*/
  set sashelp.quakes;
run;
```
proc sort data=mysas.quakes;
   by Magnitude;
run;                                             /* 3 */

data mysas.quakes_mag;                           /* 4 */
   set mysas.quakes;                              /* 5 */
   by Magnitude;                                 /* 6 */
run;
proc print data=mysas.quakes_mag(obs=10);        /* 7 */
   title "Earthquakes by Magnitude";             /* 7 */
run;

1 The **LIBNAME statement** creates the libref, mysas. The path to the library is specified in single or double quotation marks.

2 To create the permanent data set for the example, the **SET statement** reads the quakes data set from the Sashelp library. The DATA statement writes the data set quakes to the mysas library, where it is saved to disc.

3 The **SORT procedure** sorts the mysas.quakes data set by the values of the variable Magnitude.

4 The **DATA statement** writes the quakes data set to the data set quakes_mag in the same library.

5 The **SET statement** reads the data set quakes from the mysas library.

6 The **BY statement** groups and orders the observations by the values of the variable Magnitude.

7 The **PRINT procedure** prints the results for the quakes_mag data set. The **OBS= data set option** in the PRINT statement causes the PRINT procedure to display only the first 10 observations of the output data set.

The following PROC PRINT output shows the results:

**Output 14.4  PROC PRINT Output for Mysas.Quakes by Magnitude**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Depth</th>
<th>Magnitude</th>
<th>dNearestStation</th>
<th>RootMeanSquareTime</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36.8481</td>
<td>-97.675</td>
<td>4.2510</td>
<td>2.5</td>
<td>.</td>
<td>0.4610</td>
<td>earthquake</td>
</tr>
<tr>
<td>2</td>
<td>36.0210</td>
<td>-97.037</td>
<td>4.8420</td>
<td>2.5</td>
<td>.</td>
<td>0.5300</td>
<td>earthquake</td>
</tr>
<tr>
<td>3</td>
<td>36.7038</td>
<td>-90.041</td>
<td>12.7600</td>
<td>2.5</td>
<td>0.2030</td>
<td>0.2500</td>
<td>earthquake</td>
</tr>
<tr>
<td>4</td>
<td>36.7564</td>
<td>-97.556</td>
<td>6.2220</td>
<td>2.5</td>
<td>.</td>
<td>0.7600</td>
<td>earthquake</td>
</tr>
<tr>
<td>5</td>
<td>41.9035</td>
<td>-119.662</td>
<td>1.7742</td>
<td>2.5</td>
<td>0.4770</td>
<td>0.2545</td>
<td>earthquake</td>
</tr>
<tr>
<td>6</td>
<td>36.3510</td>
<td>-84.898</td>
<td>20.7000</td>
<td>2.5</td>
<td>0.3413</td>
<td>0.2000</td>
<td>earthquake</td>
</tr>
<tr>
<td>7</td>
<td>35.7967</td>
<td>-97.444</td>
<td>5.4500</td>
<td>2.5</td>
<td>.</td>
<td>0.3500</td>
<td>earthquake</td>
</tr>
<tr>
<td>8</td>
<td>36.7484</td>
<td>-97.649</td>
<td>3.3550</td>
<td>2.5</td>
<td>.</td>
<td>0.5000</td>
<td>earthquake</td>
</tr>
<tr>
<td>9</td>
<td>41.8692</td>
<td>-119.615</td>
<td>5.0000</td>
<td>2.5</td>
<td>0.6557</td>
<td>0.6500</td>
<td>earthquake</td>
</tr>
<tr>
<td>10</td>
<td>41.8552</td>
<td>-119.673</td>
<td>0.2000</td>
<td>2.5</td>
<td>0.45712</td>
<td>0.4500</td>
<td>earthquake</td>
</tr>
</tbody>
</table>

---

**Note:** The output is only a portion of the output data set. The **OBS= data set option** in the PROC PRINT statement limits the number of observations that are displayed.
Key Ideas

- If you do not specify a location for the output data set, the DATA statement automatically writes the output to the SAS Work library. Data sets written to the Work library are saved only for the duration of the current SAS session. To specify a permanent output location, you must create a SAS library (libref) by using the LIBNAME statement. See Chapter 12, “SAS Libraries,” on page 219 for more information about libraries in SAS.

- The SET statement reads SAS data sets into the DATA step for processing. You can also use the MERGE statement, the MODIFY statement, and the UPDATE statement to read SAS data sets into a DATA step.

- The DATA statement writes out SAS data sets that have been processed by the DATA step.

- By default, the DATA step reads observations from a SAS data set using sequential access. For more information about sequential and direct data access, see “Ways to Read Rows in SAS Data Sets” on page 291.

See Also

- Chapter 12, “SAS Libraries,” on page 219
- “BY Statement” in SAS DATA Step Statements: Reference

Example: Read a Permanent SAS Data Set Without Using a Libref

Example Code

This example reads a permanent SAS data set that has been saved to the Windows directory, demo, and writes it out to a new data set in a different directory (the demo2 directory). Instead of using a libref, the pathname to the SAS data set is specified in quotation marks in the DATA statement.

The SET statement reads the data set, shoesales, in the folder demo. The DATA statement uses the same syntax to write the output to shoesales2, in the folder demo2.

```sas
data "c:\Users\demo\shoesales2.sas7bdat";
set "c:\Users\demo\shoesales.sas7bdat";
run;

data "c:\Users\Jdoe\courses2.sas7bdat";
set "c:\Users\Jdoe\sasuser\courses.sas7bdat";
run;
proc print data="c:\Users\Jdoe\courses2.sas7bdat"; run;
```

The following shows the log output:
Key Ideas

- If you do not specify a location for the output data set, the DATA statement automatically writes the output to the SAS Work library. Data sets written to the Work library are saved only for the duration of the current SAS session. To specify a permanent output location, you must create a SAS library (libref) by using the LIBNAME statement. See Chapter 12, “SAS Libraries,” on page 219 for more information about libraries in SAS.

- The SET statement reads SAS data sets into the DATA step for processing. You can also use the MERGE statement, the MODIFY statement, and the UPDATE statement to read SAS data sets into a DATA step.

- The DATA statement writes out SAS data sets that have been processed by the DATA step.

- By default, the DATA step reads observations from a SAS data set using sequential access. For more information about sequential and direct data access, see “Ways to Read Rows in SAS Data Sets” on page 291.

See Also

- Chapter 12, “SAS Libraries,” on page 219
- “SAS Default Libraries” on page 227
Examples: Control Variables and Observations in Data Sets

Example: Keep Specific Variables in a Data Set

Example Code

This example uses the **KEEP statement** to control which variables are written to the output data set.

The **LIBNAME statement** specifies the location for writing the output data set, and it associates that location with the name `mysas`. The **SET statement** reads the input data set, `Sashelp.Cars` and the **DATA statement** writes the results to the output data set `mysas.cars`.

This example uses the **KEEP statement** to include only the variables `Make`, `Mpg`, and `MSRP` in the output data set. SAS reads all the variables from the `Sashelp.Cars` data set into memory and then removes the unwanted variables when it creates the output data set. The variables `MPG_City` and `MPG_Highway` are read into memory by the DATA step and are used to calculate the weighted average MPG, but they are not included in the output.

```sas
libname mysas "c:\Users\demo";
data cars;
set sashelp.cars;
  keep make mpg MSRP;
  Mpg=(MPG_City*.45)+(MPG_Highway*.55)/2;
run;
proc print data=cars(obs=10); run;
```

**Output 14.6** *PROC PRINT* Output for the Mysas.Cars Data Set Showing Only the Variables Specified in the KEEP Statement

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>MSRP</th>
<th>Mpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acura</td>
<td>$36,045</td>
<td>13.075</td>
</tr>
<tr>
<td>2</td>
<td>Acura</td>
<td>$23,320</td>
<td>10.225</td>
</tr>
<tr>
<td>3</td>
<td>Acura</td>
<td>$26,000</td>
<td>17.875</td>
</tr>
<tr>
<td>4</td>
<td>Acura</td>
<td>$33,195</td>
<td>18.700</td>
</tr>
<tr>
<td>5</td>
<td>Acura</td>
<td>$43,755</td>
<td>14.700</td>
</tr>
<tr>
<td>6</td>
<td>Acura</td>
<td>$46,100</td>
<td>14.700</td>
</tr>
<tr>
<td>7</td>
<td>Acura</td>
<td>$56,705</td>
<td>14.250</td>
</tr>
<tr>
<td>8</td>
<td>Audi</td>
<td>$25,940</td>
<td>19.425</td>
</tr>
<tr>
<td>9</td>
<td>Audi</td>
<td>$35,040</td>
<td>19.800</td>
</tr>
<tr>
<td>10</td>
<td>Audi</td>
<td>$31,340</td>
<td>18.700</td>
</tr>
</tbody>
</table>
Note: The output is only a portion of the output data set. The OBS= data set option in the PROC PRINT statement limits the number of observations that are displayed.

An alternate way to control the selection of variables is to use the KEEP= data set option. The KEEP= data set option in the input data set in the SET statement controls which variables are read and written to the output data set. The KEEP= data set option in the output data set in the DATA statement controls which variables are written to the output data set.

Key Ideas

- If you do not instruct it to do otherwise, SAS writes all variables and all observations from input data sets to output data sets.
- You can control which variables and observations that you want to read and write by using SAS statements, data set options, and functions. See “Ways to Manage Variables in SAS Data Sets” on page 289 for a list of these language elements.
- The DROP statement and DROP= data set option work the same way that the KEEP= statement and KEEP= data set option work except that the selected variables are dropped rather than kept.

See Also

- Comparing the DROP= data set option and the DROP statement
- Comparing the KEEP= data set option and the KEEP statement
- “Excluding Variables from Input ” in SAS Data Set Options: Reference
- “Processing Variables without Writing Them to a Data Set” in SAS Data Set Options: Reference

Example: Control Which Observations are Read Using the WHERE Statement

Example Code

This example uses the WHERE statement to select observations that are based on the values of the variables age and height.

```sas
data class;
  set sashelp.class;
  where age>12 and height>=67;
run;
proc print data=class; run;
```
When the DATA step runs this program, it does not read every row in the input data set, as the following log shows:

Output 14.7 Log Output

```
NOTE: There were 3 observations read from the data set SASHELP.CLASS.
WHERE (age>12) and (height>=67);
```

Using the WHERE statement can improve the efficiency of your SAS program when the DATA step has to read fewer observations.

You can also use the WHERE= data set option to conditionally select observations in either the input data set or the output data set.

```
data class;
  set sashelp.class(where=(age>12 and height >=67));
run;
```

Like the WHERE statement, the WHERE= data set option, when specified in the input data set, does not require that the DATA step reads all the observations. Here is the log output when the WHERE= data set option is specified in the SET statement, in the input data set:

Output 14.8 Log Output

```
NOTE: There were 3 observations read from the data set SASHELP.CLASS.
WHERE (age>12) and (height>=67);
```

When the WHERE= data set option is specified in the DATA statement, in the output data set, the DATA step reads all the rows of the input data set and then writes only those observations that meet the criteria to the output data set:

```
data class(where=(age>12 and height >=67));
  set sashelp.class;
run;
```

Output 14.9 Log Output

```
NOTE: There were 19 observations read from the data set SASHELP.CLASS.
NOTE: The data set WORK.CLASS has 3 observations and 5 variables.
```
Key Ideas

- If you do not instruct it to do otherwise, SAS writes all variables and all observations from input data sets to output data sets.
- You can control which variables and observations that you want to read and write by using SAS statements, data set options, and functions. See "Controlling the Reading and Writing of Variables and Observations" in SAS Language Reference: Concepts for a list of these language elements.
- The WHERE statement controls which observations are read by the SET statement based on the value of a variable.
- When the WHERE statement is used in a DATA step, the DATA step does not read every observation in the input data set. Therefore, using the WHERE statement can improve the efficiency of your SAS programs.
- The WHERE= data set option in the DATA statement controls which observations are written to the output data set by the DATA statement. When the WHERE= data set option is specified in the DATA statement, the DATA step reads all the observations in the input data set.

See Also

- "WHERE Statement" in SAS DATA Step Statements: Reference and "Specify the WHERE Statement in a SAS DATA Step" in SAS DATA Step Statements: Reference
- "WHERE= Data Set Option" in SAS Data Set Options: Reference

Example: Read a Specific Observation First (FIRSTOBS)

Example Code

In this example, the DATA step directly accesses a specific observation in the input data set and writes the output to a new data set starting with that specified observation. The FIRSTOBS= data set option specifies that the observations are written to the output data set, quakes2, beginning with observation number 5 of the input data set, Sashelp.quakes.

To create an input data set for this example, the following DATA step is used to create a subset of the Sashelp.Quakes data set. The DATA step creates a subset by conditionally selecting observations in which the values of the variable Magnitude are greater than 6.0.

```sas
data quakes;
  set sashelp.quakes(where=(Magnitude>6.0));
  keep Depth Type Magnitude;
run;
proc print data=quakes; run;
```
Output 14.10  PROC PRINT Output for the Quakes Data Set

<table>
<thead>
<tr>
<th>Obs</th>
<th>Depth</th>
<th>Magnitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.25</td>
<td>6.02</td>
<td>earthquake</td>
</tr>
<tr>
<td>2</td>
<td>9.45</td>
<td>6.60</td>
<td>earthquake</td>
</tr>
<tr>
<td>3</td>
<td>13.00</td>
<td>6.00</td>
<td>earthquake</td>
</tr>
<tr>
<td>4</td>
<td>13.00</td>
<td>7.00</td>
<td>earthquake</td>
</tr>
<tr>
<td>5</td>
<td>4.00</td>
<td>7.20</td>
<td>earthquake</td>
</tr>
<tr>
<td>6</td>
<td>10.00</td>
<td>6.20</td>
<td>earthquake</td>
</tr>
<tr>
<td>7</td>
<td>10.00</td>
<td>6.60</td>
<td>earthquake</td>
</tr>
<tr>
<td>8</td>
<td>14.00</td>
<td>6.60</td>
<td>earthquake</td>
</tr>
</tbody>
</table>

In the next DATA step, the FIRSTOBS= data set option is specified in the SET statement to create a new output data set that begins with observation 5 from the input data set. The output data set contains all of the remaining observations from the input data set.

```sas
data quakes2;
  set quakes(firstobs=5);
run;
proc print data=quakes2; run;
```

Output 14.11  PROC PRINT Output Showing Row 5 from the Quakes Data Set as the First Row in the Quakes2 Data Set

<table>
<thead>
<tr>
<th>Obs</th>
<th>Depth</th>
<th>Magnitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>7.2</td>
<td>earthquake</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>6.2</td>
<td>earthquake</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>6.8</td>
<td>earthquake</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>6.8</td>
<td>earthquake</td>
</tr>
</tbody>
</table>

You can also use the following functions to access observations directly by observation number: the NOTE function, the CUROBS function, the POINT function, and the FETCHOBS function.

Key Ideas

- When the OBS= data set option specifies an ending point for processing, the FIRSTOBS= data set option specifies a starting point. The two options are often used together to define a range of observations to be processed.
- The OBS= data set option enables you to select observations from SAS data sets. You can select observations to be read from external data files by using the OBS= option in the INFILE statement.

See Also

- “FIRSTOBS= System Option” in SAS System Options: Reference and “OBS= System Option” in SAS System Options: Reference
- “FIRSTOBS= Data Set Option” in SAS Data Set Options: Reference and “OBS= Data Set Option” in SAS Data Set Options: Reference


Example: Read a Range of Observations (FIRSTOBS and OBS)

Example Code

In this example, the DATA step accesses a range of observations in a data set by using the FIRSTOBS= and OBS= data set options together. The FIRSTOBS= data set option specifies that the observations are written to the output data set, quakes2, beginning with observation number 2 from the input data set, Sashelp.quakes. The OBS= data set option tells SAS to stop processing observations after reading a specified number of observations.

SAS uses the following formula to determine the number of observations to read when using the OBS= and FIRSTOBS= data set options together: \( (\text{obs} - \text{firstobs}) + 1 = \text{number of rows} \).

To create the input data set for this example, the first DATA step creates a subset of the Sashelp.Quakes data set. The second DATA step creates a range of observations.

```
data quakes;
  set sashelp.quakes(where=(Magnitude>6.0));
  keep Depth Type Magnitude;
run;
proc print data=quakes; run;

data quakes2;
  set quakes(firstobs=2 obs=4);
run;
proc print data=quakes2; run;
```

Output 14.12  PROC PRINT Output Showing a Range of Observations Selected from Quakes and Written to Quakes2

<table>
<thead>
<tr>
<th>Obs</th>
<th>Depth</th>
<th>Magnitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11.25</td>
<td>8.02</td>
<td>earthquake</td>
</tr>
<tr>
<td>2</td>
<td>0.45</td>
<td>8.60</td>
<td>earthquake</td>
</tr>
<tr>
<td>3</td>
<td>15.00</td>
<td>8.30</td>
<td>earthquake</td>
</tr>
<tr>
<td>4</td>
<td>13.00</td>
<td>7.00</td>
<td>earthquake</td>
</tr>
<tr>
<td>5</td>
<td>4.00</td>
<td>7.20</td>
<td>earthquake</td>
</tr>
<tr>
<td>6</td>
<td>10.00</td>
<td>8.20</td>
<td>earthquake</td>
</tr>
<tr>
<td>7</td>
<td>10.00</td>
<td>8.00</td>
<td>earthquake</td>
</tr>
<tr>
<td>8</td>
<td>14.00</td>
<td>8.80</td>
<td>earthquake</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>Depth</th>
<th>Magnitude</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.45</td>
<td>6.00</td>
<td>earthquake</td>
</tr>
<tr>
<td>2</td>
<td>13.00</td>
<td>6.30</td>
<td>earthquake</td>
</tr>
<tr>
<td>3</td>
<td>12.00</td>
<td>7.00</td>
<td>earthquake</td>
</tr>
</tbody>
</table>

You can also use the following functions to access observations directly by observation number: the NOTE function, the CUROBS Function, the POINT function, and the FETCHOBS function.
Key Ideas

- When the OBS= data set option specifies an ending point for processing, the FIRSTOBS= data set option specifies a starting point. The two options are often used together to define a range of observations to be processed.
- The OBS= data set option enables you to select observations from SAS data sets. You can select observations to be read from external data files by using the OBS= option in the INFILE statement.

See Also

- “FIRSTOBS= System Option” in SAS System Options: Reference and “OBS= System Option” in SAS System Options: Reference
- “FIRSTOBS= Data Set Option” in SAS Data Set Options: Reference and “OBS= Data Set Option” in SAS Data Set Options: Reference
- “Applying FIRSTOBS= and OBS= Options” on page 396 (using the FIRSTOBS= and OBS= data set options with the WHERE statement to segment data)

Example: Read an Observation Directly Using the POINT= Option

Example Code

In this example, the DATA step directly accesses the third row in the Sashelp.Comet data set.

A temporary numeric variable, num, is created to hold the value of the observation to be directly accessed from the Sashelp.Comet data set. The assignment statement creates the variable and assigns a value of 3, which represents the third observation in the Sashelp.Comet data set. Then, the POINT= option is specified in the SET statement and is set equal to the temporary variable num. The OUTPUT statement writes the current observation to the output data set Comet. The STOP statement is used to prevent continuous processing of the DATA step.

The CALL SYMPUT routine assigns the value for the row number to a macro variable that can be used in the TITLE statement to denote which row is being accessed.

The first PRINT procedure below is used to show the first few rows of the input data set in which the third row is directly accessed by the DATA step.

```sas
proc print data=sashelp.comet(obs=5);
title "Sashelp.Comet Data Set";
run;
```
data comet;
  num=3;
  set sashelp.comet point=num;
  call symput('num',num);
  output;
  stop;
run;

proc print data=comet;
  title "Row &num from Sashelp.Comet Data Set";
run;

Output 14.13  Partial PROC PRINT Output for the Sashelp.Comet Data Set (for Comparison)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Dose</th>
<th>Rat</th>
<th>Sample</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>15.3027</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>15.1820</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>14.0078</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>12.4431</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>12.0031</td>
</tr>
</tbody>
</table>

Output 14.14  PROC Print Output for the Comet Data Set Showing the Directly Accessed Row 3

<table>
<thead>
<tr>
<th>Obs</th>
<th>Dose</th>
<th>Rat</th>
<th>Sample</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>14.0078</td>
</tr>
</tbody>
</table>

Key Ideas

- If you do not instruct it to do otherwise, SAS writes all variables and all observations from input data sets to output data sets.
- You can control which variables and observations that you want to read and write by using SAS statements, data set options, and functions. See “Controlling the Reading and Writing of Variables and Observations” in SAS Language Reference: Concepts for a list of these language elements.
- Because SAS does not detect an end-of-file when directly accessing an observation using the POINT= option, you must specify the STOP statement with the POINT= option to prevent continuous processing of the DATA step.
- The WHERE statement controls which observations are read by the SET statement based on the value of a variable.

See Also

- POINT= option
- STOP statement
- OUTPUT statement
Examples: View Descriptor and Sort Information for Data Sets

Example: View Descriptor Information for a Data Set

Example Code

In this example, the CONTENTS procedure displays information about the SAS data set Sashelp.Snacks.

```sas
proc contents data=sashelp.snacks;
run;
```
**Key Ideas**

- The descriptor information is the part of a SAS data set that contains information about the contents of the data set.
- Descriptor information includes the number of observations, the observation length, the date that the data set was last modified, and other facts.
- Descriptor information for individual variables includes attributes such as name, type, length, format, label, and whether the variable is indexed.
- When a data set has been sorted, the Sort Information table shows the BY variable that was used to sort the data.
Example: View Sort Information for a Data Set

Example Code

In this example, the CONTENTS procedure is used to view information about the Sashelp.Air data set. The Sorted field in the PROC CONTENTS output indicates that the data set is not sorted. Therefore, there is no additional Sort Indicator table in the output.

```
proc contents data=sashelp.air; run;
```

**Output 14.16  PROC CONTENTS Output Showing That the Data Set Sashelp.Air is Not Sorted**

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>SASHELP.AIR</th>
<th>Observations</th>
<th>144</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>2</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>06/03/2018 21:41:44</td>
<td>Observation Length</td>
<td>16</td>
</tr>
<tr>
<td>Last Modified</td>
<td>06/03/2018 21:41:44</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td></td>
<td>Sorted</td>
<td>NO</td>
</tr>
<tr>
<td>Label</td>
<td>air line data (monthly); JAN18-DEC50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>WINDOWS64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the DATA step below, data set air is created from the Sashelp.Air data set and it is sorted by the variable air. The CONTENTS procedure is used again to view the sort information.

```
data air(sortedby=air);
   set sashelp.air;
run;
```

```
proc contents data=air; run;
```

The PROC CONTENTS output indicates that the data set was sorted using the SORTEDBY=data set option. The Sort Information table (the Sort Indicator) is included now. Notice that the Sorted field is set to NO. This is because the SORTEDBY=data set option was used to sort the data set. Sort information indicates a valid sort only when the data set is sorted using either PROC SORT or PROC SQL.
Next, the SORT procedure is used to sort the data set by the values of the variable **air**, in descending order.

```
proc sort data=air; by descending air; run;
proc contents data=air; run;
```

**Output 14.18**  Partial PROC CONTENTS Output Showing Validated Sort
The PROC CONTENTS output now shows that the data set is sorted and that the sort is validated. YES in the Validated field indicates that the data was sorted by SAS using PROC SORT or PROC SQL.

Key Ideas

- The sort indicator is set when a data set is sorted by any of the following methods:
  - SORT procedure
  - ORDER BY clause in PROC SQL
  - MODIFY statement in PROC DATASETS
  - SORTEDBY= data set option in the DATA step DATA statement
- PROC SORT and PROC SQL generate validated sorts, which can be seen in the Validated field of the descriptor information.
- The SORTEDBY= data set option and the SORTEDBY= option in the DATASETS procedure generate sorts that are not validated.
- SAS procedures that require data to be sorted read sort indicator field in the the descriptor information for the data set to determine if the data set is sorted.
- You can specify the SORTVALIDATE system option in the OPTIONS statement to validate sorts and to ensure that the data set is sorted according to the variables in the BY statement. If the data set is not sorted correctly, SAS sorts the data set.

See Also

- “CONTENTS Procedure” in Base SAS Procedures Guide
Raw Data

Definitions for Raw Data

raw data
unprocessed data that has not been read into a SAS data set. You can use the IMPORT procedure or the SAS DATA step to read raw data into a SAS data set. Here are the types of raw data that SAS can read:

- instream data
- external files

Note: “Definition of SAS/ACCESS Software” on page 345.

Raw data does not include Database Management System (DBMS) files. You must license SAS/ACCESS software to access data stored in DBMS files. For more information about SAS/ACCESS features, see "About SAS/ACCESS Software" in SAS Language Reference: Concepts.
Reading Raw Data

Ways to Read Raw Data

You can read raw data by using one of the following items:

- **IMPORT procedure**

  See the following examples:
  - “Importing a Delimited File” in *Base SAS Procedures Guide*
  - “Importing a Specific Delimited File Using a Fileref” in *Base SAS Procedures Guide*
  - “Importing a Tab-Delimited File” in *Base SAS Procedures Guide*

- **DATA Step Statements on page 318**

- **SAS I/O functions, such as the FOPEN, FGET, and FCLOSE functions.**

  For a description of available functions, see the SAS File I/O and External File categories in “SAS Functions and CALL Routines by Category” in *SAS Functions and CALL Routines: Reference*. See “Using Functions to Manipulate Files” in *SAS Functions and CALL Routines: Reference* for information about how statements and functions manipulate files.

- **External File Interface (EFI)**

  If your operating environment supports a graphical user interface, you can use the EFI or the Import Wizard to read raw data. The EFI is a point-and-click graphical interface that you can use to read and write data that is not in SAS software’s internal format. By using EFI, you can read data from an external file and write it to a SAS data set. You can also read data from a SAS data set and write it to an external file. For more information about EFI, see *SAS/ACCESS Interface to PC Files: Reference*.

- **Import Wizard**

  The Import Wizard guides you through the steps to read data from an external data source and write it to a SAS data set. As a wizard, it is a series of windows that present simple choices to guide you through a process. For more information about the wizard, see *SAS/ACCESS Interface to PC Files: Reference*.

---

**Note:** If the data file that you are passing to EFI is password protected, you are prompted multiple times for your logon ID and password.

**Operating Environment Information:** Using external files with your SAS jobs requires that you specify filenames with syntax that is appropriate to your operating environment. For information about operating system documentation, see “Operating Environment Information” on page 14.
Reading Raw Instream Data

Definition for Instream Data

**Instream data**
rows of raw data records that are contained within a SAS program. Instream data begins with the **DATALINES statement** and ends with a semicolon on a new line in a SAS DATA step.

Reading Instream Data

This example reads in instream data using the **INPUT** and **DATALINES** statements. The **INPUT** statement specifies the variable (column) names and the **DATALINES** statement indicates the raw, instream data to follow:

```sas
data weight;
  input PatientID $ State $;
  loss=Week1-Week16;
  datalines;
2477 TX
2431 TX
2456 NC
2412 NC
;
```

**Note:** A semicolon appearing alone on the line immediately following the last data line is the convention that is used in this example. However, a PROC statement, DATA statement, or a global statement ending in a semicolon on the line immediately following the last data line also submits the previous DATA step.

Reading Instream Data Containing Semicolons

This example reads in instream data containing semicolons:

```sas
data weight;
  input PatientID $ Week1 Week8 Week16;
  loss=Week1-Week16;
  datalines4;
24;77 195 177 163
24;31 220 213 198
24;56 173 166 155
24;12 135 125 116
```

```sas
; ; ; ;
```
The SAS DATA step reads raw data from instream data lines or from external files. The INPUT statement in the DATA step reads the data from these sources into a SAS data set.

Choosing an Input Style

When you read raw data with a DATA step, you can use a combination of the INPUT statement, DATALINES statement, and INFILE statement. You can use the following different input styles, depending on the layout of data values in the records:

- List input
- Formatted input
- Column input
- Named input

You can also combine styles of input in a single INPUT statement. For details about the styles of input, see the INPUT statement.

List Input

List input uses a scanning method for locating data values. Data values are not required to be aligned in columns but must be separated by at least one blank (or other defined delimiter). List input requires only that you specify the variable names and a dollar sign ($), if defining a character variable. You do not have to specify the location of the data fields.

An example of reading raw instream data using list input follows:

```sas
data scores;
  length name $ 12;
  input name $ score1 score2;

datalines;
  Riley 1132 1187
  Henderson 1015 1102
;
```

For more examples, see "Reading Unaligned Data with Simple List Input" in SAS DATA Step Statements: Reference.

List input has several restrictions on the type of data that it can read:

- Input values must be separated by at least one blank (the default delimiter) or by the delimiter specified with the DLM= or DLMSTR= option in the INFILE statement. If you want SAS to read consecutive delimiters as if there is a missing value between them, specify the DSD option in the INFILE statement.
- Blanks cannot represent missing values. A real value, such as a period, must be used instead.
- To read and store a character value that is longer than 8 bytes, you must explicitly define its length. You can define a variable’s length by using either the LENGTH statement, the INFORMAT statement, or the ATTRIB statement. When
you define a character variable using either of these statements, you must place the statement that defines the variable’s length first in the DATA step, before any other references to that variable. You can also specify the variable’s length by using modified list input, which consists of an informat and the colon modifier in the INPUT statement.

- Character values cannot contain embedded blanks when the file is delimited by blanks.
- Fields must be read in order.
- Data must be in standard numeric or character format.

**Note:** Nonstandard numeric values, such as packed decimal data, must use the formatted style of input. For more information, see “Formatted Input” on page 321.

---

**Modified List Input**

A more flexible version of list input, called modified list input, includes format modifiers. The following format modifiers enable you to use list input to read nonstandard data by using SAS informat:

- & (ampersand) format modifier enables you to read character values that contain one or more embedded blanks with list input and to specify a character informat. SAS reads until it encounters two consecutive blanks, the defined length of the variable, or the end of the input line, whichever comes first.

- : (colon) format modifier enables you to use list input but also to specify an informat after a variable name, whether character or numeric. SAS reads until it encounters a blank column, the defined length of the variable (character only), or the end of the data line, whichever comes first.

- ~ (tilde) format modifier enables you to read and retain single quotation marks, double quotation marks, and delimiters within character values.

Here is an example of the : and ~ format modifiers. You must use the DSD option in the INFILE statement. Otherwise, the INPUT statement ignores the ~ format modifier. This example reads raw instream data using modified list input:

```sas
data scores;
  infile datalines dsd;
  input Name : $9. Score1-Score3 Team ~ $25. Div $;

datalines;
  Smith,12,22,46,"Green Hornets, Atlanta",AAA
  Mitchel,23,19,25,"High Volts, Portland",AAA
  Jones,09,17,54,"Vulcans, Las Vegas",AA
;
proc print data=scores;
```

Reading Raw Data 319
Output 15.1  Output from Example with Format Modifiers

The SAS System

<table>
<thead>
<tr>
<th>Name</th>
<th>Score1</th>
<th>Score2</th>
<th>Score3</th>
<th>Team</th>
<th>Div</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>12</td>
<td>22</td>
<td>46</td>
<td>&quot;Green Hornets, Atlanta&quot;</td>
<td>AAA</td>
</tr>
<tr>
<td>Mitchel</td>
<td>23</td>
<td>19</td>
<td>25</td>
<td>&quot;High Volts, Portland&quot;</td>
<td>AAA</td>
</tr>
<tr>
<td>Jones</td>
<td>9</td>
<td>17</td>
<td>54</td>
<td>&quot;Vulcans, Las Vegas&quot;</td>
<td>AA</td>
</tr>
</tbody>
</table>

For another example, see Reading Delimited Data with Modified List Input.

Column Input

Column input enables you to read standard data values that are aligned in columns in the data records. Specify the variable name, followed by a dollar sign ($) if it is a character variable, and specify the columns in which the data values are located in each record:

```sas
data scores;
  infile datalines truncover;
  input name $ 1-12 score2 17-20 score1 27-30;
  
  datalines;
  Riley         1132       987
  Henderson     1015      1102

;```

For more examples, see "Read Input Records with Column Input" in SAS DATA Step Statements: Reference.

Note: Use the TRUNCOVER option in the INFILE statement to ensure that SAS handles data values of varying lengths appropriately.

To use column input, data values must be:
- in the same field on all the input lines
- in standard numeric or character form

Note: You cannot use an informat with column input.

Here are some features of column input:
- Character values can contain embedded blanks.
- Character values can be from 1 to 32,767 characters long.
- Placeholders, such as a single period (.), are not required for missing data.
- Input values can be read in any order, regardless of their position in the record.
- Values or parts of values can be reread.
- Both leading and trailing blanks within the field are ignored.
Values do not need to be separated by blanks or other delimiters.

**CAUTION**
If you insert tabs while entering data in the DATALINES statement in column format, you might get unexpected results. This issue exists when you use the SAS Enhanced Editor or SAS Program Editor. To avoid the issue, do one of the following actions:

- Replace all tabs in the data with single spaces using another editor outside of SAS.
- Specify the `%INCLUDE` statement from the SAS editor to submit your code.
- If you are using the SAS Enhanced Editor, select **Tools** ⇒ **Options** ⇒ **Enhanced Editor** to change the tab size from 4 to 1.

**Formatted Input**

Formatted input combines the flexibility of using informats with many of the features of column input. By using formatted input, you can read nonstandard data for which SAS requires additional instructions. Formatted input is typically used with pointer controls that enable you to control the position of the input pointer in the input buffer when you read data.

The INPUT statement in the following DATA step uses formatted input and pointer controls to read the raw, instream data listed in the DATALINES statement.

Note that `$12.$` and `COMMA5.$` are informats; `+4` and `+6` are column pointer controls.

```sas
data scores;
  input name $12. +4 score1 comma5. +6 score2 comma5.;
  datalines;
  Riley           1,132      1,187
  Henderson       1,015      1,102
;```

For more examples, see "Formatted Input with Pointer Controls" in *SAS DATA Step Statements: Reference*.

**Note:** You can also use informats to read data that is not aligned in columns. See "Modified List Input" in *SAS Language Reference: Concepts* for more information.

Here are some features of formatted input:

- Characters values can contain embedded blanks.
- Character values can be from 1 to 32,767 characters long.
- Placeholders, such as a single period (.) are not required for missing data.
- With the use of pointer controls to position the pointer, input values can be read in any order, regardless of their positions in the record.
- Values or parts of values can be reread.
- Formatted input enables you to read data stored in nonstandard form, such as packed decimal or numbers with commas.
Named Input

Named input enables you to read records in which data values are preceded by the name of the variable and an equal sign (=). The following INPUT statement reads the data lines containing equal signs.

```sas
data games;
  input name=$ score1= score2=;

datalines;
  name=riley score1=1132 score2=1187
;
```

For more examples, see “Using List and Named Input” in SAS DATA Step Statements: Reference.

Note: When an equal sign follows a variable in an INPUT statement, SAS expects that data remaining on the input line contains only named input values. You cannot switch to another form of input in the same INPUT statement after using named input. Also, note that any variable that exists in the input data but is not defined in the INPUT statement generates a note in the SAS log indicating a missing field.

Additional Data-Reading Features

In addition to different styles of input, there are many tools to meet the needs of different data-reading situations. You can use options in the INFILE statement in combination with the INPUT statement to give you additional control over the reading of data records. The following table lists common data-reading tasks and the appropriate features available in the INPUT and INFILE statements.

Table 15.1 Additional Data-Reading Features

<table>
<thead>
<tr>
<th>Input</th>
<th>Goal</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiple records</td>
<td>create a single observation</td>
<td>#n or / line pointer control in the INPUT statement, with a DO loop</td>
</tr>
<tr>
<td>a single record</td>
<td>create multiple observations</td>
<td>trailing @@ in the INPUT statement, trailing @ with multiple INPUT and OUTPUT statements (Example)</td>
</tr>
<tr>
<td>variable-length data fields</td>
<td>read delimited data</td>
<td>list input with or without a format modifier in the INPUT statement and the TRUNCOVER, DLM=, DLMSTR=, or DSD options in the INFILE statement (Examples)</td>
</tr>
<tr>
<td>and records</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>Goal</td>
<td>Use</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>read non-delimited data</td>
<td>$VARYINGw. informat in the INPUT statement and the LENGTH= and TRUNCVER options in the INFILE statement</td>
<td>(Example)</td>
</tr>
<tr>
<td>a file with varying record layouts</td>
<td>IF-THEN statements with multiple INPUT statements, using trailing @ or @@ as necessary</td>
<td></td>
</tr>
<tr>
<td>hierarchical files</td>
<td>IF-THEN statements with multiple INPUT statements, using trailing @ as necessary</td>
<td></td>
</tr>
<tr>
<td>more than one input file or to control the program flow at EOF</td>
<td>EOF= option or END= option in an INFILE statement</td>
<td>(Example)</td>
</tr>
<tr>
<td></td>
<td>multiple INFILE and INPUT statements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FILEVAR= in an INFILE statement.</td>
<td>(Example)</td>
</tr>
<tr>
<td></td>
<td>FILENAME statement with concatenation, wildcard, or piping</td>
<td></td>
</tr>
<tr>
<td>only part of each record</td>
<td>LINESIZE= option in an INFILE statement</td>
<td></td>
</tr>
<tr>
<td>some but not all records in the file</td>
<td>FIRSTOBS= and OBS= options in an INFILE statement; FIRSTOBS= and OBS= system options; #n line pointer control</td>
<td>(Example)</td>
</tr>
<tr>
<td>instream data lines</td>
<td>control the reading with special options</td>
<td>INFILe statement with DATALINES and appropriate options</td>
</tr>
<tr>
<td>starting at a particular column</td>
<td>@ column pointer controls</td>
<td></td>
</tr>
<tr>
<td>leading blanks</td>
<td>maintain them</td>
<td>$CHARw. informat in an INPUT statement</td>
</tr>
<tr>
<td>a delimiter other than blanks (with list input or modified list input with the colon modifier)</td>
<td>DLM= option or the DSD option both the DLM= option and the DSD option in an INFILE statement</td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>Goal</td>
<td>Use</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>the standard tab character</td>
<td></td>
<td>DLM= or in an INFILE statement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>both the DLM= and the DLMSTR= options</td>
</tr>
<tr>
<td></td>
<td></td>
<td>EXPANDTABS option in an INFILE statement</td>
</tr>
<tr>
<td>missing values (with list input or</td>
<td>create observations without</td>
<td>TRUNCOVER option in an INFILE statement</td>
</tr>
<tr>
<td>modified list input with the colon modifier)</td>
<td>compromising data integrity</td>
<td>DLM=, DLMSTR=, or , or DSD might also be needed</td>
</tr>
<tr>
<td></td>
<td>protect data integrity by overriding the default behavior</td>
<td></td>
</tr>
</tbody>
</table>

For more information about data-reading features, see the INPUT and INFILE statements in *SAS DATA Step Statements: Reference*.

Types of Raw Data

Definitions for Types of Raw Data

data values
- are character or numeric values.

numeric value
- contains only numbers, and sometimes a decimal point, a minus sign, or both. When they are read into a SAS data set, numeric values are stored in the floating-point format native to the operating environment. Nonstandard numeric values can contain other characters as numbers; you can use formatted input to enable SAS to read them.

character value
- is a sequence of characters.

standard data
- are character or numeric values that can be read with list, column, formatted, or named input. Examples of standard data include:
  - ARKANSAS
  - 1166.42

nonstandard data
- is data that can be read only with the aid of informats. Examples of nonstandard data include numeric values that contain commas, dollar signs, or blanks; date and time values; and hexadecimal and binary values.
### Numeric Data

Numeric data can be represented in several ways. SAS can read standard numeric values without any special instructions. To read nonstandard values, SAS requires special instructions in the form of informats. “Reading Nonstandard Numeric Data” in *SAS Language Reference: Concepts* shows standard, nonstandard, and invalid numeric data values and the special tools, if any, that are required to read them. For complete descriptions of all SAS informats, see *SAS Formats and Informats: Reference*.

#### Table 15.2  Reading Standard Numeric Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>input right aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>23</td>
<td>input not aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>23</td>
<td>input left aligned</td>
<td>None needed</td>
</tr>
<tr>
<td>00023</td>
<td>input with leading zeros</td>
<td>None needed</td>
</tr>
<tr>
<td>23.0</td>
<td>input with decimal point</td>
<td>None needed</td>
</tr>
<tr>
<td>2.3E1</td>
<td>in E notation, 2.30 (ss1)</td>
<td>None needed</td>
</tr>
<tr>
<td>230E-1</td>
<td>in E notation, 230x10 (ss-1)</td>
<td>None needed</td>
</tr>
<tr>
<td>-23</td>
<td>minus sign for negative numbers</td>
<td>None needed</td>
</tr>
</tbody>
</table>

#### Table 15.3  Reading Nonstandard Numeric Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 3</td>
<td>embedded blank</td>
<td>COMMA or BZ. informat</td>
</tr>
<tr>
<td>- 23</td>
<td>embedded blank</td>
<td>COMMA or BZ. informat</td>
</tr>
<tr>
<td>2,341</td>
<td>comma</td>
<td>COMMA informat</td>
</tr>
<tr>
<td>(23)</td>
<td>parentheses</td>
<td>COMMA informat</td>
</tr>
<tr>
<td>C4A2</td>
<td>hexadecimal</td>
<td>HEX informat</td>
</tr>
<tr>
<td>1MAR90</td>
<td>date value</td>
<td>DATE informat</td>
</tr>
</tbody>
</table>
Table 15.4  Reading Invalid Numeric Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Description</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>minus sign follows number</td>
<td>Put minus sign before number or solve programmatically. It might be possible to use the S370FZDTw.d informat, but positive values require the trailing plus sign (+).</td>
</tr>
<tr>
<td>.</td>
<td>double instead of single periods</td>
<td>Code missing values as a single period or use the ?? modifier in the INPUT statement to code any invalid input value as a missing value.</td>
</tr>
<tr>
<td>J23</td>
<td>not a number</td>
<td>Read as a character value, or edit the raw data to change it to a valid number.</td>
</tr>
</tbody>
</table>

Remember the following rules for reading numeric data:

- Parentheses or a minus sign preceding the number (without an intervening blank) indicates a negative value.
- Leading zeros and the placement of a value in the input field do not affect the value assigned to the variable. Leading zeros and leading and trailing blanks are not stored with the value. Unlike some languages, SAS does not read trailing blanks as zeros by default. To cause trailing blanks to be read as zeros, use the BZ. informat described in SAS Formats and Informats: Reference.
- Numeric data can have leading and trailing blanks but cannot have embedded blanks (unless they are read with a COMMA. or BZ. informat).
- To read decimal values from input lines that do not contain explicit decimal points, indicate where the decimal point belongs by using a decimal parameter with column input or an informat with formatted input. See the full description of the INPUT statement in SAS Formats and Informats: Reference for more information. An explicit decimal point in the input data overrides any decimal specification in the INPUT statement.

Character Data

A value that is read with an INPUT statement is assumed to be a character value if one of the following conditions is true:

- A dollar sign ($) follows the variable name in the INPUT statement.
- A character informat is used.
- The variable has been previously defined as character. For example, a value is assumed to be a character value if the variable has been previously defined as...
character in a LENGTH statement, in the RETAIN statement, by an assignment statement, or in an expression.

Input data that you want to store in a character variable can include any character. Use the guidelines in the following table when your raw data includes leading blanks and semicolons.

<table>
<thead>
<tr>
<th>Table 15.5  Reading Instream Data and External Files Containing Leading Blanks and Semicolons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Characters in the Data</strong></td>
</tr>
<tr>
<td>leading or trailing blanks that you want to preserve</td>
</tr>
<tr>
<td>semicolons in instream data</td>
</tr>
<tr>
<td>delimiters, blank characters, or quoted strings</td>
</tr>
</tbody>
</table>

Remember the following facts when reading character data:

- In a DATA step, when you place a dollar sign ($) after a variable name in the INPUT statement, character data that is read from data lines remains in its original case. If you want SAS to read data from data lines as uppercase, use the CAPS system option or the $UPCASE informat.
- If the value is shorter than the length of the variable, SAS adds blanks to the end of the value to give the value the specified length. This process is known as padding the value with blanks.

How SAS Handles Invalid Data

An input value is invalid if it has any of the following characteristics:

- It requires an informat that is not specified.
It does not conform to the informat specified.

It does not match the input style used. An example is if it is read as standard numeric data (no dollar sign or informat) but it does not conform to the rules for standard SAS numbers.

It is out of range (too large or too small).

**Operating Environment Information:** The range for numeric values is operating environment-specific. See “Operating Environment Information” on page 14 for information about operating system documentation.

If SAS reads a data value that is incompatible with the type specified for that variable, SAS tries to convert the value to the specified type. If conversion is not possible, an error occurs, and SAS performs the following actions:

- sets the value of the variable being read to missing or to the value specified with the INVALIDDATA= system option.
- prints an invalid data note in the SAS log.
- sets the automatic variable _ERROR_ to 1 for the current observation.
- prints the input line and column number containing the invalid value in the SAS log. If a line contains unprintable characters, it is printed in hexadecimal form. A scale is printed above the input line to help determine column numbers.

---

### How SAS Handles Missing Values

#### Representing Missing Values in Input Data

Many collections of data contain some missing values. SAS can recognize these values as missing when it reads them. You use the following characters to represent missing values when reading raw data:

**numeric missing values**
- are represented by a single decimal point (.). All input styles except list input also allow a blank to represent a missing numeric value.

**character missing values**
- are represented by a blank, with one exception: list input requires that you use a period (.) to represent a missing numeric value.

**special numeric missing values**
- are represented by two characters: a decimal point (.) followed by either a letter or an underscore (_).

For more information about missing values, see “Reading Column-Binary Data” on page 338.

#### Special Missing Values in Numeric Input Data

SAS enables you to differentiate among classes of missing values in numeric data. For numeric variables, you can designate up to 27 special missing values by using the letters A through Z, in either uppercase or lowercase, and the underscore character (_).
The following example shows how to code missing values by using a MISSING statement in a DATA step:

```sas
data test_results;
  missing a b c;
  input name $8. Answer1 Answer2 Answer3;

datalines;
Smith    2 5 9
Jones    4 b 8
Carter   a 4 7
Reed     3 5 c
;
```

Note that you must use a period when you specify a special missing numeric value in an expression or assignment statement, as in the following:

```sas
x=.d;
```

However, you do not need to specify each special missing numeric data value with a period in your input data. For example, the following DATA step, which uses periods in the input data for special missing values, produces the same result as the input data without periods:

```sas
data test_results;
  missing a b c;
  input name $8. Answer1 Answer2 Answer3;

datalines;
Smith    2 5 9
Jones    4 .b 8
Carter   .a 4 7
Reed     3 5 .c
;
```

```sas
proc print;
run;
```

Output 15.2  Output of Data with Special Missing Numeric Values

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>Answer1</th>
<th>Answer2</th>
<th>Answer3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smith</td>
<td>2</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Jones</td>
<td>4</td>
<td>B</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Carter</td>
<td>A</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Reed</td>
<td>3</td>
<td>5</td>
<td>C</td>
</tr>
</tbody>
</table>

Note: SAS is displayed and prints special missing values that use letters in uppercase.
Definitions for External Files

**binary data**
numeric data that is stored in binary form in an external file. Binary numbers have a base of two and are represented with the digits 0 and 1.

**column-binary data storage**
an older form of data storage that is no longer widely used and is not needed by most SAS users. Column-binary data storage compresses data so that more than 80 items of data can be stored on a single “virtual” punched card. The advantage is that this method enables you to store more data in the same amount of space. Because card-image data sets remain in existence, SAS provides informats for reading column-binary data. See “Reading Column-Binary Data” on page 338 for more information about column-binary data storage.

**external files**
files that are managed and maintained by your operating system, not by SAS. They contain data or text as input to SAS jobs, or they are

External files as input to a SAS session include the following:
- records of raw data in external files that you want to read into SAS as input, including data in the form of plain ASCII text files and binary files.
- programming statements in external files that you want to submit to SAS for execution.

External files as output from a SAS session include the following:
- files in which you want to store data or text, including data written out as records in plain ASCII text files or data written out in binary form.
- files created as the result of running a SAS program, including SAS log files and results from SAS procedures. For example, the `PRINTTO` procedure enables you to direct procedure output to an external file. Every SAS job creates at least one external file, the SAS log. SAS catalog files and ODS output destinations are also examples of external files.

**packed decimal data**
binary decimal numbers that are encoded by using each byte to represent two decimal digits. Packed decimal representation stores decimal data with exact precision; the fractional part of the number must be determined by using an informat or format because there is no separate mantissa and exponent.

**zoned decimal data**
binary decimal numbers that are encoded so that each digit requires one byte of storage. The last byte contains the number’s sign as well as the last digit. Zoned decimal data produces a printable representation.

Operating Environment Information: Using external files with your SAS jobs entails significant operating-environment-specific information.

See “Operating Environment Information” on page 14 for information about operating system documentation specific to your operating environment.
Note: External files do not include database management system (DBMS) files or PC files. DBMS and PC files are a special category of files that can be read with SAS/ACCESS software.

- For information about DBMS files, see Chapter 16, "Database and PC Files," on page 345 and SAS/ACCESS for Relational Databases: Reference.
- For information about access to PC files, see Chapter 16, "Database and PC Files," on page 345 and SAS/ACCESS Interface to PC Files: Reference

---

Reading External Files

Reading an External File Using the DATA Step

The following example shows how to read in raw data from an external file using the INFILE and INPUT statements. This example is similar to formatted input example in that it uses formatted input with a SAS informat to read in the raw data. But, for this example, notice how the INFILE statement is required instead of the DATALINES statement because the source data is in an external text file.

```sas
data weight;
  infile '<fileref>' or '<path-name>';  
  input PatientID $ Week1 Week8 Week16;
  loss=Week1-Week16;
run;
```

---

Reading and Writing to External Files Directly

To reference a file directly in a SAS statement or command, specify in quotation marks its physical name. This is the name by which the operating environment recognizes it, as shown in the following table:

<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
</table>
| Specify the file that contains input data. | INFILE statement | data weight;
  infile '/path/weight.csv';
  input idno $ week1 week16;
  loss=week1-week16;
  run; |
Table 15.7  Writing Data to an External File Directly

<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
</table>
| Identify the file that the PUT statement writes to. | FILE statement | file 'mydata.txt';
| | | if loss ge 5 and loss le 9 then
| | | put idno loss 'AWARD STATUS=3';
| | | else if loss ge 10 and loss le 14 then
| | | put idno loss 'AWARD STATUS=2';
| | | else if loss ge 15 then
| | | put idno loss 'AWARD STATUS=1';
| | | run; |

Table 15.8  Including an External File That Contains Programming Statements

<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bring statements or raw data from another file into your SAS job and execute them.</td>
<td>%INCLUDE statement</td>
<td>%include 'myprogram.txt';</td>
</tr>
</tbody>
</table>

Referencing External Files Indirectly By Using a Fileref

If you want to reference a file in only one place in a program so that you can easily change it for another job or a later run, you can reference a filename indirectly. Use a FILENAME statement, the FILENAME function, or an appropriate operating system command to assign a fileref or nickname to a file.¹ Note that you can assign a fileref to a SAS catalog that is an external file, or to an output device, as shown in the following table.

---

¹ In some operating environments, you can also use the command ‘&’ to assign a fileref.
<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
</table>
| Assign a fileref to an external file that contains input data. | FILENAME statement | /* Using PROC IMPORT */
| | | filename myinput 'c:\Users\sasuser\lake.txt';
| | | proc import
datafile=myinput
| | | out=lake
| | | dbms=dlm replace;
delimiter=' '; run;
| | /* Using the DATA step */
| | | filename myinput 'c:\Users\sasuser\lake.txt';
data lake;
infile myinput;
| | | input Width Length Depth; run; |
| Assign a fileref to an external file for output data. | FILENAME statement | filename myoutput 'c:\Users\sasuser\lake.txt';
data _null_; set sashelp.lake;
| | | file myoutput;
| | | put Width Length Depth; run; |
| Assign a fileref to an external file that contains program statements. | FILENAME statement | filename mypgm 'c:\Users\sasuser\myprogram.txt'; |
| Assign a fileref to an output device. | FILENAME statement | filename myprinter <device-type>
| | | <host-options>; |
| Specify the file that contains input data. | INFILE statement | filename myinput 'c:\Users\sasuser\lake.txt';
data lake;
infile myinput;
| | | input Width Length Depth; run; |
| Specify the file that the PUT statement writes values to. | FILE statement | filename myoutput 'c:\Users\sasuser\lake.txt';
data _null_; set sashelp.lake;
| | | file myoutput;
| | | put Width Length Depth; run; |
| Bring statements or raw data from another file into your SAS job and execute them. | %INCLUDE statement | %include mypgm; |
Referencing Many External Files Efficiently

When you use many files from a single aggregate storage location, such as a directory or partitioned data set (PDS or MACLIB), you can use a single fileref, followed by a filename enclosed in parentheses, to access the individual files. This saves time by eliminating the need to enter a long file storage location name repeatedly. It also makes changing the program easier later if you change the file storage location. The following table shows an example of assigning a fileref to an aggregate storage location:

<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign a fileref to aggregate storage location.</td>
<td>FILENAME statement</td>
<td><code>filename mydir 'directory-or-PDS-name';</code></td>
</tr>
</tbody>
</table>
| Specify the file that contains input data. | INFILE statement | `data weight;`  
`  infile mydir(qrt1.data);`  
`  input idno $ week1 week16;`  
`  loss=week1-week16;` |
| Specify the file that the PUT statement writes to. SAS creates a file that is named with the appropriate extension for your operating environment. | FILE statement | `file mydir(awards);`  
`  if loss ge 5 then`  
`    put idno loss 'AWARD STATUS=3';`  
`  else if loss ge 10 then`  
`    put idno loss 'AWARD STATUS=2';`  
`  else if loss ge 15 then`  
`    put idno loss 'AWARD STATUS=1';`  
`  run;` |
| Bring statements or raw data from another file into your SAS job and execute them. | `%include mydir(whole.program);`  
`%include mydir(whole.program);` |

Referencing External Files with Other Access Methods

You can assign filerefs to external files that you access with the following FILENAME access methods:
### Table 15.11 Referencing External Files with Other Access Methods

<table>
<thead>
<tr>
<th>Task</th>
<th>Language Element</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign a fileref to a SAS catalog that is an aggregate storage location.</td>
<td>FILENAME statement with CATALOG specifier</td>
<td>filename mycat catalog 'catalog' &lt;catalog-options&gt;;</td>
</tr>
<tr>
<td>Assign a fileref to an external file accessed by a data URL.</td>
<td>FILENAME statement with DATAURL specifier</td>
<td>filename myfile dataurl 'external-file' &lt;dataurl-options&gt;;</td>
</tr>
<tr>
<td>Assign a fileref to an external file accessed with FTP.</td>
<td>FILENAME statement with FTP specifier</td>
<td>filename myfile FTP 'external-file' &lt;ftp-options&gt;;</td>
</tr>
<tr>
<td>Assign a fileref to an external file accessed on a Hadoop Distributed File System.</td>
<td>FILENAME statement with Hadoop specifier</td>
<td>filename myfile hadoop 'external-file' &lt;hadoop-options&gt;;</td>
</tr>
<tr>
<td>Assign a fileref to an external file accessed with SFTP.</td>
<td>FILENAME statement with SFTP specifier</td>
<td>filename myfile SFTP 'external-file' &lt;sftp-options&gt;;</td>
</tr>
</tbody>
</table>
| Assign a fileref to an external file accessed by TCP/IP SOCKET in either client or server mode. | FILENAME statement with SOCKET specifier | filename myfile SOCKET 'hostname: portno' <tcpip-options>; 
  or 
  filename myfile SOCKET ':portno' SERVER <tcpip-options>; |
| Assign a fileref to an external file accessed by URL. | FILENAME statement with URL specifier | filename myfile URL 'external-file' <url-options>; |
| Assign a fileref to an external file accessed on a WebDAV server. | FILENAME statement with WebDAV specifier | filename myfile WEBDAV 'external-file' <webdav-options>; |
Assign a fileref to a ZIP file accessed by using Zlib services.

**FILENAME** statement with **ZIP** specifier

```
filename myfile ZIP 'external-file'
<zip-options>;
```

See *SAS DATA Step Statements: Reference* for detailed information about each of these statements.

---

**Reading Binary Data**

**Reading Binary Data Using SAS Informats**

SAS can read binary data with the special instructions supplied by SAS informats. You can use formatted input and specify the informat in the INPUT statement. The informat that you choose is determined by the following factors:

- the type of number being read: binary, packed decimal, zoned decimal, or a variation of one of these
- the type of system on which the data was created
- the type of system that you use to read the data

Different computer platforms store numeric binary data in different forms. The ordering of bytes differs by platforms that are referred to as either “big endian” or “little endian.” For more information, see “Byte Ordering for Integer Binary Data on Big Endian and Little Endian Platforms” in *SAS Formats and Informats: Reference*.

SAS provides a number of informats for reading binary data and corresponding formats for writing binary data. Some of these informats read data in native mode, that is, by using the byte-ordering system that is standard for the system on which SAS is running. Other informats force the data to be read by the IBM 370 standard, regardless of the native mode of the system on which SAS is running. The informats that read in native or IBM 370 mode are listed in the following table.

**Table 15.12 Informats for Native or IBM 370 Mode**

<table>
<thead>
<tr>
<th>Description</th>
<th>Native Mode Informats</th>
<th>IBM 370 Mode Informats</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII Character</td>
<td>$w.$</td>
<td>$ASCIIw.$</td>
</tr>
<tr>
<td>ASCII Numeric</td>
<td>$w.d$</td>
<td>$ASCIIw.$</td>
</tr>
<tr>
<td>EBCDIC Character</td>
<td>$w.$</td>
<td>$EBCDICw.$</td>
</tr>
<tr>
<td>EBCDIC Numeric (Standard)</td>
<td>$w.d$</td>
<td>S370FFw.d</td>
</tr>
<tr>
<td>Integer Binary</td>
<td>IBw.d</td>
<td>S370FIBw.d</td>
</tr>
</tbody>
</table>
If you write a SAS program that reads binary data and that is run on only one type of system, you can use the native mode informats and formats. However, if you want to write SAS programs that can be run on multiple systems that use different byte-storage systems, use the IBM 370 informats. The IBM 370 informats enable you to write SAS programs that can read data in this format and that can be run in any SAS environment, regardless of the standard for storing numeric data.

For example, using the IBM 370 informats, you could download data that contain binary integers from a mainframe to a PC and then use the S370FIB informats to read the data.

### Table: Native Mode Informats and IBM 370 Mode Informats

<table>
<thead>
<tr>
<th>Description</th>
<th>Native Mode Informats</th>
<th>IBM 370 Mode Informats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive Integer Binary</td>
<td>PIBw.d</td>
<td>S370FPIBw.d</td>
</tr>
<tr>
<td>Real Binary</td>
<td>RBw.d</td>
<td>S370FRBw.d</td>
</tr>
<tr>
<td>Unsigned Integer Binary</td>
<td>PIBw.d, S370FPIBw.d</td>
<td></td>
</tr>
<tr>
<td>Packed Decimal</td>
<td>PDw.d</td>
<td>S370FPDw.d</td>
</tr>
<tr>
<td>Unsigned Packed Decimal</td>
<td>PKw.d, S370FPDUw.d</td>
<td></td>
</tr>
<tr>
<td>Zoned Decimal</td>
<td>ZDw.d</td>
<td>S370FZDw.d</td>
</tr>
<tr>
<td>Zoned Decimal Leading Sign</td>
<td>S370FZDLw.d</td>
<td>S370FZDLw.d</td>
</tr>
<tr>
<td>Zoned Decimal Separate Leading Sign</td>
<td>S370FZDSw.d</td>
<td>S370FZDSw.d</td>
</tr>
<tr>
<td>Zoned Decimal Separate Trailing Sign</td>
<td>S370FZDTw.d</td>
<td>S370FZDTw.d</td>
</tr>
<tr>
<td>Unsigned Zoned Decimal</td>
<td>ZDw.d, S370FZDUw.d</td>
<td></td>
</tr>
</tbody>
</table>

Note: Anytime a text file originates from anywhere other than the local encoding environment, it might be necessary to specify the ENCODING= option on either EBCDIC or ASCII systems. When you read an EBCDIC text file on an ASCII platform, it is recommended that you specify the ENCODING= option in the FILENAME or INFILE statement. However, if you use the DSD and the DLM= or DLMSTR= options in the INFILE statement, the ENCODING= option is a requirement because these options require certain characters in the session encoding (such as quotation marks, commas, and blanks). Reserve encoding-specific informats for use with true binary files that contain both character and non-character fields.

For complete descriptions of all SAS formats and informats, including how numeric binary data is written, see [SAS Formats and Informats: Reference](#).
Reading Column-Binary Data

To read column-binary data with SAS, you need to know:

- how to select the appropriate SAS column-binary informat
- how to set the RECFM= and LRECL= options in the INFILE statement
- how to use pointer controls

The following table lists and describes SAS column-binary informats.

**Table 15.13  SAS Informats for Reading Column-Binary Data**

<table>
<thead>
<tr>
<th>Informat Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CBw.</td>
<td>reads standard character data from column-binary files</td>
</tr>
<tr>
<td>CBw.</td>
<td>reads standard numeric data from column-binary files</td>
</tr>
<tr>
<td>PUNCH.d</td>
<td>reads whether a row is punched</td>
</tr>
<tr>
<td>ROWw.d</td>
<td>reads a column-binary field down a card column</td>
</tr>
</tbody>
</table>

To read column-binary data, you must set two options in the INFILE statement:

- Set RECFM= to F for fixed.
- Set the LRECL= to 160, because each card column of column-binary data is expanded to two bytes before the fields are read.

For example, to read column-binary data from a file, use an INFILE statement in the following form before the INPUT statement that reads the data:

```sas
data out;
infile file-specification or path-name recfm=f lrecl=160;
input var1;
run;
```

**Note:** The expansion of each column of column-binary data into two bytes does not affect the position of the column pointer. You use the absolute column pointer control @, as usual, because the informats automatically compute the true location on the doubled record. If a value is in column 23, use the pointer control @23 to move the pointer there.
Example: Read a Comma-Delimited File

Example Code

In this example, the IMPORT procedure imports a file that contains comma-separated values.

```sas
filename chol temp;    /*1*/
proc http                           /*2*/
  url="http://support.sas.com/documentation/onlinedoc/viya/
  exampledatasets/cholesterol.csv"
  out=chol;
quit;
proc import datafile=chol           /*3*/
  out=work.mycholesterol
dbms=csv
  replace;
run;
proc print data=work.mycholesterol; /*4*/
run;
```

1 The FILENAME statement assigns the chol fileref. The TEMP option specifies that the file is temporary, so a path name is not necessary.

2 The HTTP procedure specifies the URL of the cholesterol.csv input file. The data is written to the chol fileref.

3 PROC IMPORT reads the comma-delimited data and creates the mycholesterol data set.

4 The PRINT procedure prints the data set. The output shows that the file was imported correctly, including the column names.

The output shows that the file was imported correctly, including the column names.
Key Ideas

- Base SAS software can import and export some external text files without using a SAS/ACCESS engine. These files are usually referred to as raw data or delimited data. The DATA step or PROC IMPORT can read data from a text file and provide that data to the V9 engine for output to a SAS data set.

- If you want Base SAS to read or write a Microsoft Excel file, the file must have a .csv extension. To import a file that has an Excel file extension of .xls or .xlsx, you must have a license to SAS/ACCESS Interface to PC Files. In Excel, you can save an Excel file as a .csv file.

- If you create a SAS data set from an Excel file, the data is static and does not change to reflect the underlying Excel data. If you want to keep data in Excel as an ongoing data store that you can query from SAS, then use the XLSX or EXCEL engine. You must have a license to SAS/ACCESS Interface to PC Files.

See Also

- “Default Base SAS Engine (V9 Engine)” on page 262
- Chapter 15, “Raw Data,” on page 315
- “IMPORT Procedure” in Base SAS Procedures Guide
- “Comparing SAS LIBNAME Engines for PC Files Data” in SAS/ACCESS Interface to PC Files: Reference
Example: Read and Write a Space-delimited File

Example Code

The following example shows how to write data from a Sashelp data set to an external, space-delimited text file. It then reads the data from the external text file into a SAS data set named air.

```sas
proc export data=sashelp.air
    outfile="c:\Users\sasuser\air.txt" /*1*/
    dbms=' ' /*2*/
    replace; /*3*/
run;

/* read the file into a SAS data set */
proc import datafile='C:\Users\sasuser\txt\air.txt /*4*/
    out=air                                /*5*/
    dbms=dlm                               /*6*/
    replace;
    delimiter=' '; run;

proc print data=cholesterol;                      /*7*/
run;
```

1 Write the contents of the Sashelp.air data set to an external, space-delimited text file named air.txt. Specify the name and where to store the file. Include the full path and file name quotation marks in the OUTFILE= option in PROC EXPORT.

2 Specify how the file is to be delimited by specifying dlm in the DBMS= option in PROC EXPORT.

3 Replace the file if it already exists by specifying the REPLACE option.

4 Read the external file back into SAS by specifying the IMPORT procedure.

5 Specify the name of the output data set (air) in the OUT= option.

6 Specify how the file is to be delimited by specifying dlm in the DBMS= option in PROC EXPORT.

7 Replace the file if it already exists by specifying the REPLACE option.

PROC IMPORT builds a DATA step to read the external file and writes the DATA step code to the SAS log:
Output 15.4 Partial PROC PRINT Output for Reading a Space-Delimited Text File Using PROC IMPORT

<table>
<thead>
<tr>
<th>Obs</th>
<th>DATE</th>
<th>AIR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAN1949</td>
<td>112</td>
</tr>
<tr>
<td>2</td>
<td>FEB1949</td>
<td>118</td>
</tr>
<tr>
<td>3</td>
<td>MAR1949</td>
<td>132</td>
</tr>
<tr>
<td>4</td>
<td>APR1949</td>
<td>129</td>
</tr>
<tr>
<td>5</td>
<td>MAY1949</td>
<td>121</td>
</tr>
<tr>
<td>6</td>
<td>JUN1949</td>
<td>135</td>
</tr>
<tr>
<td>7</td>
<td>JUL1949</td>
<td>148</td>
</tr>
<tr>
<td>8</td>
<td>AUG1949</td>
<td>148</td>
</tr>
<tr>
<td>9</td>
<td>SEP1949</td>
<td>136</td>
</tr>
<tr>
<td>10</td>
<td>OCT1949</td>
<td>119</td>
</tr>
<tr>
<td>11</td>
<td>NOV1949</td>
<td>104</td>
</tr>
</tbody>
</table>
PROC IMPORT automatically generates this DATA step and prints the step to the SAS Log.

The **MISSOVER** option prevents the INPUT statement from reading a new input data record if it does not find values in the current input line for all the variables in the statement.

The **DSD** option specifies that when data values are enclosed in quotation marks, delimiters within the value are treated as character data.

**FIRSTOBS=** data set option specifies the first observation to be processed in the data set.

**MONYYw.** informat reads month and year date values in the form monyy.

The **MMYYxw.** format writes date values in the form mm<yy>yy or mm-<yy>yy, where the x in the format name is a character that represents the special character that separates the month and the year.

The **INPUT statement** assigns input values to the corresponding SAS variables.

**Key Ideas**

- You can use the **DATAROW statement** to specify which row to begin reading data from. For example, you specify **DATAROW=1** for external files that do not contain column names.
See Also

- “PROC IMPORT Statement” in *Base SAS Procedures Guide*
- “DELIMITER Statement” in *Base SAS Procedures Guide*
- “Importing a Tab-Delimited File” in *Base SAS Procedures Guide*
Definition of SAS/ACCESS Software

SAS/ACCESS software enables you to read and write data to and from other vendors' database management systems (DBMS), as well as from some PC file formats. Depending on your DBMS, a SAS/ACCESS product might provide one or more of the following:

- a dynamic LIBNAME engine
- the SQL pass-through facility
- the ACCESS procedure and interface view engine
- the DBLOAD procedure
- an interface DATA step engine

These interfaces are described in this section. Each SAS/ACCESS product provides one or more of these interfaces for each supported DBMS. See Chapter 13, “SAS Engines,” on page 257 for more information about SAS engines.

Note: To use the SAS/ACCESS features described in this section, you must license SAS/ACCESS software. See the SAS/ACCESS documentation for your DBMS for full documentation of the features described in this section.
Dynamic LIBNAME Engine

SAS/ACCESS LIBNAME Statement

Beginning in SAS 7, you can associate a SAS libref directly with a database, schema, server, or group of tables and SAS views, depending on your DBMS. To assign a libref to DBMS data, you must use the SAS/ACCESS LIBNAME statement, which has syntax and options that are different from the Base SAS LIBNAME statement. For example, to connect to an ORACLE database, you might use the following SAS/ACCESS LIBNAME statement:

```
libname mydblib oracle user=smith password=secret path='myoracleserver';
```

This LIBNAME statement connects to ORACLE by specifying the ORACLE connection options: USER=, PASSWORD=, and PATH=. In addition to the connection options, you can specify SAS/ACCESS LIBNAME options that control the type of database connection that is made. You can use additional options to control how your data is processed.

You can use a DATA step, SAS procedures, or the Explorer window to view and update the DBMS data associated with the libref, or use the DATASETS and CONTENTS procedures to view information about the DBMS objects.

See your SAS/ACCESS documentation for a full listing of the SAS/ACCESS LIBNAME options that can be used with librefs that refer to DBMS data.

Using Data Set Options with SAS/ACCESS Librefs

After you have assigned a libref to your DBMS data, you can use SAS/ACCESS data set options, and some of the Base SAS data set options, on the data. The following example associates a libref with DB2 data and uses the SQL procedure to query the data:

```
libname mydb2lib db2;

proc sql;
  select *  
    from mydb2lib.employees(drop=salary)  
    where dept='Accounting';
quit;
```

The LIBNAME statement connects to DB2. You can reference a DBMS object, in this case, a DB2 table, by specifying a two-level name that consists of the libref and the DBMS object name. The DROP= data set option causes the SALARY column of the EMPLOYEES table on DB2 to be excluded from the data that is returned by the query.

See your SAS/ACCESS documentation for a full listing of the SAS/ACCESS data set options and the Base SAS data set options that can be used on data sets that refer to DBMS data.
Embedding a SAS/ACCESS LIBNAME Statement in a PROC SQL View

You can issue a SAS/ACCESS LIBNAME statement by itself, as shown in the previous examples, or as part of a CREATE VIEW statement in PROC SQL. The USING clause of the CREATE VIEW statement enables you to store DBMS connection information in a SAS view by embedding a SAS/ACCESS LIBNAME statement inside the SAS view. The following example uses an embedded SAS/ACCESS LIBNAME statement:

```
libname viewlib 'SAS-library';
proc sql;
  create view viewlib.emp_view as
    select *
    from mydblib.employees
    using libname mydblib oracle user=smith password=secret
    path='myoraclepath';
quit;
```

When PROC SQL executes the SAS view, the SELECT statement assigns the libref and establishes the connection to the DBMS. The scope of the libref is local to the SAS view and does not conflict with identically named librefs that might exist in the SAS session. When the query finishes, the connection is terminated and the libref is unassigned.

Note: You can also embed a Base SAS LIBNAME statement in a PROC SQL view.

SQL Procedure Pass-Through Facility

The SQL Procedure pass-through facility is an extension of the SQL procedure that enables you to send DBMS-specific statements to a DBMS and to retrieve DBMS data. You specify DBMS SQL syntax instead of SAS SQL syntax when you use the pass-through facility. You can use pass-through facility statements in a PROC SQL query or store them in a PROC SQL view.

The pass-through facility consists of three statements and one component:

- The CONNECT statement establishes a connection to the DBMS.
- The EXECUTE statement sends dynamic, non-query DBMS-specific SQL statements to the DBMS.
- The CONNECTION TO component in the FROM clause of a PROC SQL SELECT statement retrieves data directly from a DBMS.
- The DISCONNECT statement terminates the connection to the DBMS.

The following pass-through facility example sends a query to an ORACLE database for processing:
proc sql;
   connect to oracle as myconn (user=smith password=secret
       path='myoracleserver');

   select *
       from connection to myconn
           (select empid, lastname, firstname, salary
                from employees
                where salary>75000);

   disconnect from myconn;
quit;

The example uses the pass-through CONNECT statement to establish a connection with an ORACLE database with the specified values for the USER=, PASSWORD=, and PATH= arguments. The CONNECTION TO component in the FROM clause of the SELECT statement enables data to be retrieved from the database. The DBMS-specific statement that is sent to ORACLE is enclosed in parentheses. The DISCONNECT statement terminates the connection to ORACLE.

To store the same query in an SQL procedure, use the CREATE VIEW statement:

libname viewlib
    'SAS-library';

proc sql;
   connect to oracle as myconn (user=smith password=secret
       path='myoracleserver');

   create view viewlib.salary as
       select *
           from connection to myconn
               (select empid, lastname, firstname, salary
                    from employees
                    where salary>75000);

   disconnect from myconn;
quit;

ACCESS Procedure and Interface View Engine

The ACCESS procedure enables you to create access descriptors, which are SAS files of member type ACCESS. They describe data that is stored in a DBMS in a format that SAS can understand. Access descriptors enable you to create SAS/ACCESS views, called view descriptors. View descriptors are files of member type VIEW that function in the same way as SAS views that are created with PROC SQL, as described in “SAS Views” in SAS V9 LIBNAME Engine: Reference.

Note: If a dynamic LIBNAME engine is available for your DBMS, it is recommended that you use the SAS/ACCESS LIBNAME statement to access your DBMS data instead of access descriptors and view descriptors. However, descriptors continue to work in SAS software if they were available for your DBMS in SAS 6. Some new
SAS features, such as long variable names, are not supported when you use descriptors.

The following example creates an access descriptor and a view descriptor in the same PROC step to retrieve data from a DB2 table:

```sas
libname adlib 'SAS-library';
libname vlib 'SAS-library';

proc access dbms=db2;
  create adlib.order.access;
  table=sasdemo.orders;
  assign=no;
  list all;

  create vlib.custord.view;
  select ordernum stocknum shipto;
  format ordernum 5.
    stocknum 4.;
run;

proc print data=vlib.custord;
run;
```

When you want to use access descriptors and view descriptors, both types of descriptors must be created before you can retrieve your DBMS data. The first step, creating the access descriptor, enables SAS to store information about the specific DBMS table that you want to query.

After you have created the access descriptor, the second step is to create one or more view descriptors to retrieve some or all of the DBMS data described by the access descriptor. In the view descriptor, you select variables and apply formats to manipulate the data for viewing, printing, or storing in SAS. You use only the view descriptors, and not the access descriptors, in your SAS programs.

The interface view engine enables you to reference your SAS view with a two-level SAS name in a DATA or PROC step, such as the PROC PRINT step in the example.

See “SAS Views” in SAS V9 LIBNAME Engine: Reference for more information about SAS views. See the SAS/ACCESS documentation for your DBMS for more detailed information about creating and using access descriptors and SAS/ACCESS views.

---

**DBLOAD Procedure**

The DBLOAD procedure enables you to create and load data into a DBMS table from a SAS data set, data file, SAS view, or another DBMS table, or to append rows to an existing table. It also enables you to submit non-query DBMS-specific SQL statements to the DBMS from your SAS session.

**Note:** If a dynamic LIBNAME engine is available for your DBMS, it is recommended that you use the SAS/ACCESS LIBNAME statement to create your DBMS data instead of the DBLOAD procedure. However, DBLOAD continues to work in SAS
software if it was available for your DBMS in SAS 6. Some new SAS features, such as long variable names, are not supported when you use the DBLOAD procedure.

The following example appends data from a previously created SAS data set named INVDATA into a table in an ORACLE database named INVOICE:

```sas
proc dbload dbms=oracle data=invdata append;
  user=smith;
  password=secret;
  path='myoracleserver';
  table=invoice;
  load;
run;
```

See the SAS/ACCESS documentation for your DBMS for more detailed information about the DBLOAD procedure.

### Interface DATA Step Engine

Some SAS/ACCESS software products support a DATA step interface. This support enables you to read data from your DBMS by using DATA step programs. Some products support both reading and writing in the DATA step interface.

The DATA step interface consists of the following four statements:

- The INFILE statement identifies the database or message queue to be accessed.
- The INPUT statement is used with the INFILE statement to issue a GET call to retrieve DBMS data.
- The FILE statement identifies the database or message queue to be updated, if writing to the DBMS is supported.
- The PUT statement is used with the FILE statement to issue an UPDATE call, if writing to the DBMS is supported.

The following example updates data in an IMS database by using the FILE and INFILE statements in a DATA step. The statements generate calls to the database in the IMS native language, DL/I. The DATA step reads Bank.Customer, an existing SAS data set that contains information about new customers, and then it updates the ACCOUNT database with the data in the SAS data set.

```sas
data _null_;  
  set bank.customer;  
  length ssa1 $9;  
  infile accupdt dli call=func dbname=db ssa=ssa1;  
  file accupdt dli;  
  func = 'isrt';  
  db = 'account';  
  ssa1 = 'customer';  
  put @1   ssnumber $char11.  
  @12  custname $char40.  
  @52  addr1 $char30.  
  @82  addr2 $char30.  
  @112 custcity $char28.
```
In SAS/ACCESS products that provide a DATA step interface, the INFILE statement has special DBMS-specific options that enable you to specify DBMS variable values and to format calls to the DBMS appropriately. See the SAS/ACCESS documentation for your DBMS for a full listing of the DBMS-specific INFILE statement options and the Base SAS INFILE statement options that can be used with your DBMS.
Definitions for SAS Views

A SAS view is a virtual data set that extracts data values from other files in order to provide a customized and dynamic representation of the data.

- A view contains no data, but rather references data that is stored elsewhere.
- In most cases, you can use a SAS view as if it were a SAS data set.

Here are the types of SAS views:

**DATA step view**
- is a stored DATA step program. See “SAS Views” in SAS V9 LIBNAME Engine: Reference.

**PROC SQL view**
- is a stored query expression that is created in PROC SQL. See “SAS Views” in SAS V9 LIBNAME Engine: Reference. See also “Creating and Using PROC SQL Views” in SAS SQL Procedure User’s Guide.

SAS views are supported by the V9 engine. SAS views can also reference DBMS data if the appropriate SAS/ACCESS engine is licensed. SAS views are not supported by the CAS engine or the SPD Engine.
Definition of a DICTIONARY Table

A DICTIONARY table is a read-only SAS view that contains information about SAS libraries, SAS data sets, SAS macros, and external files that are in use or available in the current SAS session. A DICTIONARY table also contains the settings for SAS system options that are currently in effect.

When you access a DICTIONARY table, SAS determines the current state of the SAS session and returns the desired information accordingly. This process is performed each time a DICTIONARY table is accessed, so that you always have current information.

DICTIONARY tables can be accessed by a SAS program by using either of these methods:

- run a PROC SQL query against the table, using the DICTIONARY libref
- use any SAS procedure or the DATA step, referring to the PROC SQL view of the table in the Sashelp library

For more information about DICTIONARY tables, including a list of available DICTIONARY tables and their associated Sashelp views, see SAS SQL Procedure User’s Guide.
How to View DICTIONARY Tables

About Dictionary Tables

You might want to view the contents of DICTIONARY tables in order to see information about your current SAS session, before actually using the table in a DATA step or a SAS procedure.

Some DICTIONARY tables can become quite large. In this case, you might want to view a part of a DICTIONARY table that contains only the data that you are interested in. The best way to view part of a DICTIONARY table is to subset the table using a PROC SQL WHERE clause.

How to View a DICTIONARY Table

Each DICTIONARY table has an associated PROC SQL view in the Sashelp library. You can see the entire contents of a DICTIONARY table by opening its Sashelp view with the VIEWTABLE or FSVIEW utilities. This method provides more detail than you receive in the output of the DESCRIBE TABLE statement, as shown in “How to View a Summary of a DICTIONARY Table” on page 356.

The following steps describe how to use the VIEWTABLE or FSVIEW utilities to view a DICTIONARY table in a windowing environment.

1  Invoke the Explorer window in your SAS session.
2  Select the Sashelp library. A list of members in the Sashelp library appears.
3  Select a SAS view with a name that starts with V (for example, VMEMBER).

A VIEWTABLE window appears that contains its contents. (For z/OS, type the letter 'O' in the command field for the desired member and press Enter. The FSVIEW window appears with the contents of the view.)

In the VIEWTABLE window the column headings are labels. To see the column names, select View ➔ Column Names.

How to View a Summary of a DICTIONARY Table

The DESCRIBE TABLE statement in PROC SQL produces a summary of the contents of a DICTIONARY table. The following example uses the DESCRIBE TABLE statement in order to generate a summary for the table DICTIONARY_INDEXES. (The Sashelp view for this table is Sashelp.VINDEX).

```
proc sql;
  describe table dictionary.indexes;
```
The result of the DESCRIBE TABLE statement appears in the SAS log:

NOTE: SQL table DICTIONARY.INDEXES was created like:

create table DICTIONARY.INDEXES
{
   libname char(8) label='Library Name',
   memname char(32) label='Member Name',
   memtype char(8) label='Member Type',
   name char(32) label='Column Name',
   idxusage char(9) label='Column Index Type',
   indxname char(32) label='Index Name',
   indxpos num label='Position of Column in Concatenated Key',
   nomiss char(3) label='Nomiss Option',
   unique char(3) label='Unique Option
};

The first word on each line is the column (or variable) name. You need to use this name when you write a SAS statement that refers to the column (or variable).

Following the column name is the specification for the type of variable and the width of the column.

The name that follows label= is the column (or variable) label.

After you know how a table is defined, you can use the processing ability of the PROC SQL WHERE clause in a PROC SQL step to extract a portion of a SAS view.

---

How to View a Subset of a DICTIONARY Table

When you know that you are accessing a large DICTIONARY and you need to use only a portion of it, use a PROC SQL WHERE clause in order to extract a subset of the original. The following PROC SQL statement demonstrates the use of a PROC SQL WHERE clause in order to extract lines from DICTIONARY.INDEXES.

```sql
proc sql;
   title 'Subset of the DICTIONARY.INDEX View';
   title2 'Rows with Column Name equal to STATE';
   select libname, memname, name
       from dictionary.indexes
       where name = 'STATE';
quit;
```

The results are shown in the following output:
Note that many character values in the DICTIONARY tables are stored as all-uppercase characters; you should design your queries accordingly.

**CAUTION**

Do not confuse the GENUM variable value in CONTENTS OUT= data set with the GEN variable value from DICTIONARY tables. GENUM from a CONTENTS procedure or statement refers to a specific generation of a data set. GEN from DICTIONARY tables refers to the total number of generations for a data set.

**DICTIONARY Tables and Performance**

When you query a DICTIONARY table, SAS gathers information that is pertinent to that table. Depending on the DICTIONARY table that is being queried, this process can include searching libraries, opening tables, and executing SAS views. Unlike other SAS procedures and the DATA step, PROC SQL can improve this process by optimizing the query before the select process is launched. Therefore, although it is possible to access DICTIONARY table information with SAS procedures or using the DATA step to access Sashelp views, it is often more efficient to use PROC SQL.

For example, the following programs both produce the same result, but the PROC SQL step runs much faster because the WHERE clause is processed before opening the tables used by Sashelp.VCOLUMN view:

```sas
data mytable;
  set sashelp.vcolumn;
  where libname='WORK' and memname='SALES';
run;

proc sql;
  create table mytable as
```
select * from sashelp.vcolumn
    where libname='WORK' and memname='SALES';
quit;

Note: SAS does not maintain DICTIONARY table information between queries. Each query of a DICTIONARY table launches a new discovery process.

If you are querying the same DICTIONARY table several times in a row, you can get even faster performance by creating a temporary SAS data set and running your query against that data set. You can create the temporary data set by using the DATA step SET statement or PROC SQL CREATE TABLE AS statement.
# PART 4

## Manipulating Data

<table>
<thead>
<tr>
<th>Chapter 19</th>
<th>Chapter 20</th>
<th>Chapter 21</th>
<th>Chapter 22</th>
<th>Chapter 23</th>
<th>Chapter 24</th>
<th>Chapter 25</th>
<th>Chapter 26</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grouping Data</td>
<td>WHERE Statement Processing</td>
<td>Combining Data</td>
<td>Using Indexes</td>
<td>Using Arrays</td>
<td>Debugging Errors</td>
<td>Optimizing System Performance</td>
<td>Using Parallel Processing</td>
</tr>
<tr>
<td>363</td>
<td>381</td>
<td>401</td>
<td>497</td>
<td>499</td>
<td>519</td>
<td>555</td>
<td>569</td>
</tr>
</tbody>
</table>
Definitions for BY-Group Processing

BY-group processing is a method of processing observations from one or more SAS data sets that are grouped or ordered by values of one or more common variables. The most common use of BY-group processing in the DATA step is to combine two or more
SAS data sets. To do this, you use the BY statement with a SET, MERGE, MODIFY, or UPDATE statement.

**BY variable**
names a variable or variables by which the data set is sorted or indexed. All data sets must be ordered or indexed on the values of the BY variable if you use the SET, MERGE, or UPDATE statements. If you use MODIFY, data does not need to be ordered. However, your program might run more efficiently with ordered data. All data sets that are being combined must include one or more BY variables. The position of the BY variable in the observations does not matter.

**BY value**
is the value or formatted value of the BY variable.

**BY group**
includes all observations with the same BY value. If you use more than one variable in a BY statement, a BY group is a group of observations with the same combination of values for these variables. Each BY group has a unique combination of values for the variables.

**FIRST.variable and LAST.variable**
are variables that SAS creates for each BY variable. SAS sets FIRST.variable when it is processing the first observation in a BY group, and sets LAST.variable when it is processing the last observation in a BY group. These assignments enable you to take different actions, based on whether processing is starting for a new BY group or ending for a BY group. For more information, see “FIRST. and LAST. DATA Step Variables” in *SAS Language Reference: Concepts*.

For more information about BY-Group processing, see “Reading, Combining, and Modifying SAS Data Sets” in *SAS Language Reference: Concepts*. See also *Combining and Modifying SAS Data Sets: Examples*.

---

**Syntax**

**Syntax**

DATA step BY-groups are created and managed using the BY statement in SAS. See “BY Statement” in *SAS DATA Step Statements: Reference* for complete syntax information.

**FIRST. and LAST. Automatic DATA Step Variables**

In the DATA step, SAS identifies the beginning and end of each BY group by creating two temporary variables for each BY variable. See How the DATA Step Identifies BY Groups “FIRST. and LAST. DATA Step Variables” on page 370 for more information about how you can use the FIRST. and LAST. variable with BY groups.
Understanding BY Groups

BY Groups with a Single BY Variable

The following figure represents the results of using a single BY variable, zipCode, in a DATA step. The input data set, zip contains street names, cities, states, and ZIP codes. The groups are created by specifying the variable zipCode in the BY statement. The DATA step arranges the zipcodes that have the same values into groups.

The figure shows five BY groups that are created from the examples “Create the Zip Data Set” in SAS Language Reference: Concepts and “Sort and Group the zipCode Data Set by a Single Variable” in SAS Language Reference: Concepts.

The first BY group contains all observations with the smallest value for the BY variable zipCode. The second BY group contains all observations with the next smallest value for the BY variable, and so on.

**Figure 19.1  BY Group Using a Single BY Variable (ZipCode)**

<table>
<thead>
<tr>
<th>ZipCode</th>
<th>State</th>
<th>City</th>
<th>Street</th>
</tr>
</thead>
<tbody>
<tr>
<td>33133</td>
<td>FL</td>
<td>Miami</td>
<td>Rice St</td>
</tr>
<tr>
<td>33133</td>
<td>FL</td>
<td>Miami</td>
<td>Thomas Ave</td>
</tr>
<tr>
<td>33133</td>
<td>FL</td>
<td>Miami</td>
<td>Surrey Dr</td>
</tr>
<tr>
<td>33133</td>
<td>FL</td>
<td>Miami</td>
<td>Trade Ave</td>
</tr>
<tr>
<td>33146</td>
<td>FL</td>
<td>Miami</td>
<td>Nervia St</td>
</tr>
<tr>
<td>33146</td>
<td>FL</td>
<td>Miami</td>
<td>Corsica St</td>
</tr>
<tr>
<td>33801</td>
<td>FL</td>
<td>Lakeland</td>
<td>French Ave</td>
</tr>
<tr>
<td>33809</td>
<td>FL</td>
<td>Lakeland</td>
<td>Egret Dr</td>
</tr>
<tr>
<td>85730</td>
<td>AZ</td>
<td>Tucson</td>
<td>Domenic Ln</td>
</tr>
<tr>
<td>85730</td>
<td>AZ</td>
<td>Tucson</td>
<td>Gleeson Pl</td>
</tr>
</tbody>
</table>

**Example Code 19.1  Create the Zip Data Set**

```sas
data zip;
  input zipCode State $ City $ Street $20-29;
datalines;
  85730 AZ Tucson Domenic Ln
```


BY Groups with Multiple BY Variables

The following figure represents the results of processing the zip data set with two BY variables, State and City. This example uses the same data set as in "Create the Zip Data Set" in SAS Language Reference: Concepts, and is arranged in an order that you can use with the following BY statement:

```
by State City;
```

The figure shows three BY groups. The data set is shown with the BY variables State and City printed on the left for easy reading. The position of the BY variables in the observations does not affect how the values are grouped and ordered.

The observations are arranged so that the observations for Arizona occur first. The observations within each value of State are arranged in order of the value of City. Each BY group has a unique combination of values for the variables State and City. For example, the BY value of the first BY group is AZ Tucson, and the BY value of the second BY group is FL Lakeland.
BY Groups with Multiple BY Variables (State and City)

**Figure 19.2**

Here is the code for creating the output shown in the figure “BY Groups with Multiple BY Variables” on page 366:

**Example Code 19.3**  Create the Zip Data Set

```plaintext
/* BY Groups with Multiple BY Variables */
data zip;
  input State $ City $ Street $13-22 ZipCode ;
datalines;
    FL Miami    Nervia St  33146
    FL Miami    Rice St    33133
    FL Miami    Corsica St 33146
    FL Miami    Thomas Ave 33133
    FL Miami    Surrey Dr  33133
    FL Miami    Trade Ave  33133
    FL Lakeland French Ave 33801
    FL Lakeland Egret Dr   33809
    AZ Tucson   Domenic Ln 85730
    AZ Tucson   Gleeson Pl 85730;
```

**Example Code 19.4**  Sort and Group the zipCode Data Set by Multiple BY Variables

```plaintext
proc sort data=zip;
  by State City;
run;

data zip;
  set zip;
  by State City;
run;
proc print data=zip noobs;
  title 'BY Groups with Multiple BY Variables: State City';
run;
```
Invoking BY-Group Processing

To create BY groups, you use the BY statement in one of two ways:

- DATA step
- PROC step (For information about BY-group processing with procedures, see “Creating Titles That Contain BY-Group Information ” in Base SAS Procedures Guide.)

The following DATA step program uses the SET statement to combine observations from three SAS data sets by interleaving the files. The data is ordered by State City and Zip.

```sas
data all_sales;
  set region1 region2 region3;
  by State City Zip;
  ... more SAS statements ...
run;
```

Determining Whether the Data Requires Preprocessing for BY-Group Processing

Before you perform BY-group processing on multiple data sets using the SET, MERGE, and UPDATE statements, you must check the data to determine whether it requires preprocessing. They require no preprocessing if the observations in all of the data sets occur in one of the following patterns:

- ascending or descending numeric order
- ascending or descending character order
- not alphabetically or numerically ordered, but grouped in some way, such as by calendar month or by a formatted value

If the observations are not in the order that you want, you must either sort the data set or create an index for it before using BY-group processing.

If you use the MODIFY statement in BY-group processing, you do not need to presort the input data. Presorting, however, can make processing more efficient and less costly.

You can use PROC SQL views in BY-group processing. For complete information, see SAS SQL Procedure User’s Guide.

Note: SAS/ACCESS Users: If you use SAS views or librefs, see SAS/ACCESS for Relational Databases: Reference for information about using BY groups in your SAS programs.
Preprocessing Input Data for BY-Group Processing

Sorting Observations for BY-Group Processing

You can use the SORT procedure to change the physical order of the observations in the data set. You can either replace the original data set, or create a new, sorted data set by using the OUT= option of the SORT procedure. In this example, PROC SORT rearranges the observations in the data set INFORMATION based on ascending values of the variables State and ZipCode, and replaces the original data set.

```sas
proc sort data=information;
   by State ZipCode;
run;
```

As a general rule, specify the variables in the PROC SORT BY statement in the same order that you specify them in the DATA step BY statement. For a detailed description of the default sorting orders for numeric and character variables, see the SORT procedure in Base SAS Procedures Guide.

Note: The BY statement honors the linguistic collation of sorted data when you use the SORT procedure with the SORTSEQ=LINGUISTIC option.

Indexing for BY-Group Processing

You can also ensure that observations are processed in ascending order by creating an index based on one or more variables in the data set. If you specify a BY statement in a DATA step, SAS looks for an appropriate index. If it finds the index, SAS automatically retrieves the observations from the data set in indexed order.

Note: Because creating and maintaining indexes require additional resources, you should determine whether using them significantly improves performance. Depending on the nature of the data in your SAS data set, using PROC SORT to order data values can be more advantageous than indexing. For an overview of indexes, see “Understanding SAS Indexes” in SAS Language Reference: Concepts.
FIRST. and LAST. DATA Step Variables

How the DATA Step Identifies BY Groups

In the DATA step, SAS identifies the beginning and end of each BY group by creating the following two temporary variables for each BY variable:

- FIRST.variable
- LAST.variable

For example, if the DATA step specifies the variable `state` in the BY statement, then SAS creates the temporary variables `FIRST.state` and `LAST.state`.

These temporary variables are available for DATA step programming but are not added to the output data set. Their values indicate whether an observation is one of the following positions:

- the first one in a BY group
- the last one in a BY group
- neither the first nor the last one in a BY group
- both first and last, as is the case when there is only one observation in a BY group

You can take actions conditionally, based on whether you are processing the first or the last observation in a BY group. See “Processing BY-Groups Conditionally” on page 375 for more information.

How SAS Determines FIRST.variable and LAST.variable

- When an observation is the first in a BY group, SAS sets the value of the `FIRST.variable` to 1. This happens when the value of the variable changed from the previous observation.
- For all other observations in the BY group, the value of `FIRST.variable` is 0.
- When an observation is the last in a BY group, SAS sets the value of `LAST.variable` to 1. This happens when the value of the variable changes in the next observation.
- For all other observations in the BY group, the value of `LAST.variable` is 0.
- For the last observation in a data set, the value of all `LAST.variable` variables are set to 1.
Using a Name Literal as the FIRST. and LAST. Variable

When you designate a name literal as the BY variable in BY-group processing and you want to refer to the corresponding FIRST. or LAST. temporary variables, you must include the FIRST. or LAST. portion of the two-level variable name within quotation marks. Here is an example:

```sas
data sedanTypes;
  set cars;
  by 'Sedan Types'n;
  if 'first.Sedan Types'n then type=1;
run;
```

For more information about BY-Group Processing and how SAS creates the temporary variables, FIRST and LAST, see “How SAS Determines FIRST.variable and LAST.variable” in SAS Language Reference: Concepts and “How SAS Identifies the Beginning and End of a BY Group” in SAS DATA Step Statements: Reference.

Example 1: Grouping Observations by State, City, and ZIP Code

This example shows how SAS uses the FIRST.variable and LAST.variable to flag the beginning and end of BY groups. Note the following:

- FIRST and LAST variables are created automatically by SAS.
- FIRST and LAST variables are referenced in the DATA step but they are not part of the output data set. Also, fixed output image.
- Six temporary variables are created for each BY variable: FIRST.State, LAST.State, FIRST.City, LAST.City, FIRST.ZipCode, and LAST.ZipCode.

```sas
data zip;
  input State $ City $ ZipCode Street $20-29;
datalines;
  FL Miami     33133 Rice St
  FL Miami     33133 Thomas Ave
  FL Miami     33133 Surrey Dr
  FL Miami     33133 Trade Ave
  FL Miami     33146 Nervia St
  FL Miami     33146 Corsica St
  FL Lakeland  33801 French Ave
  FL Lakeland  33809 Egret Dr
  AZ Tucson    85730 Domenic Ln
  AZ Tucson    85730 Gleeson Pl
;
proc sort data=zip; by State City ZipCode; run;
data zip2;
  set zip;
  by State City ZipCode;
  put _n_= City State ZipCode
  first.city= last.city=
```
Example Code 19.1  Grouping Observations by State, City, and ZIP Code

```
_1 =1  AZ Tucson  85730   FIRST.State=1 LAST.State=0  FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=0
_1 =2  AZ Tucson  85730   FIRST.State=0 LAST.State=1  FIRST.City=0 LAST.City=1
FIRST.ZipCode=0 LAST.ZipCode=1
_1 =3  FL Lakeland 33801   FIRST.State=1 LAST.State=0  FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=1
_1 =4  FL Lakeland 33809   FIRST.State=0 LAST.State=0  FIRST.City=0 LAST.City=1
FIRST.ZipCode=0 LAST.ZipCode=1
_1 =5  FL Miami    33133   FIRST.State=0 LAST.State=0  FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=0
_1 =6  FL Miami    33133   FIRST.State=0 LAST.State=0  FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=0
_1 =7  FL Miami    33133   FIRST.State=0 LAST.State=0  FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=0
_1 =8  FL Miami    33133   FIRST.State=0 LAST.State=0  FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=0
_1 =9  FL Miami    33146   FIRST.State=0 LAST.State=0  FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=1
_1 =10 FL Miami    33146   FIRST.State=0 LAST.State=1  FIRST.City=0 LAST.City=1
FIRST.ZipCode=0 LAST.ZipCode=1
```

Figure 19.3  BY Groups for State, City, and Zipcode

<table>
<thead>
<tr>
<th>State</th>
<th>City</th>
<th>ZipCode</th>
<th>FIRST. State</th>
<th>LAST. State</th>
<th>FIRST. City</th>
<th>LAST. City</th>
<th>FIRST. ZipCode</th>
<th>LAST. ZipCode</th>
</tr>
</thead>
<tbody>
<tr>
<td>AZ</td>
<td>Tucson</td>
<td>85730</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>AZ</td>
<td>Tucson</td>
<td>85730</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>FL</td>
<td>Lakeland</td>
<td>33801</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FL</td>
<td>Lakeland</td>
<td>33809</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33133</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33133</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33133</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33133</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33146</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>FL</td>
<td>Miami</td>
<td>33146</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: This is a chart used to display the contents of the log more clearly. It is not the output data set.

Example 2: Grouping Observations by City, State, and ZIP Code

This example is similar to “Example 1: Grouping Observations by State, City, and ZIP Code” in SAS Language Reference: Concepts except that the observations are grouped by City first and then by State and ZipCode.
data zip;
input State $ City $ ZipCode Street $20-29;
datalines;
FL Miami 33133 Rice St  
FL Miami 33133 Thomas Ave  
FL Miami 33133 Surrey Dr  
FL Miami 33133 Trade Ave  
FL Miami 33146 Nervia St  
FL Miami 33146 Corsica St  
FL Lakeland 33801 French Ave  
FL Lakeland 33809 Egret Dr  
AZ Tucson 85730 Domenic Ln  
AZ Tucson 85730 Gleeson Pl  
;
proc sort data=zip; by City State ZipCode; run;
data zip2;
set zip;
by City State ZipCode;
put _n_= City State ZipCode  
  first.city= last.city=  
  first.state= last.state=  
run;
proc print data=zip2; title 'By City, State, Zip'; run;

Example Code 19.2  Grouping Observations by City, State, and ZIP Code

Example 3: A Change Affecting the FIRST. variable

The value of FIRST. variable can be affected by a change in a previous value, even if the current value of the variable remains the same.

In this example, the values of FIRST. variable and LAST. variable are dependent on sort order, and not just by the value of the BY variable. For observation 3, the value
of FIRST.\(Y\) is set to 1 because BLUEBERRY is a new value for \(Y\). This change in \(Y\) causes FIRST.\(Z\) to be set to 1 as well, even though the value of \(Z\) did not change.

```sas
data fruit;
  input x $ y $ 10-18 z $ 21-29;
datalines;
apple   banana  coconut
apple   banana  coconut
apple   blueberry  citron
apricot   blueberry  citron
;

data _null_;  
  set fruit; by x y z;
  if _N_=1 then put 'Grouped by X Y Z';
  put _N_= x= first.x= last.x= first.y= last.y= first.z= last.z= ;
r
 unarmed;

data _null_;  
  set fruit; by y x z;
  if _N_=1 then put 'Grouped by Y X Z';
  put _N_= first.y= last.y= first.x= last.x= first.z= last.z= ;
r
```

**Grouped by X Y Z**

<table>
<thead>
<tr>
<th><em>N</em></th>
<th>1</th>
<th>FIRST.x=1</th>
<th>LAST.x=0</th>
<th>FIRST.y=1</th>
<th>LAST.y=0</th>
<th>FIRST.z=1</th>
<th>LAST.z=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>N</em></td>
<td>2</td>
<td>FIRST.x=0</td>
<td>LAST.x=0</td>
<td>FIRST.y=0</td>
<td>LAST.y=0</td>
<td>FIRST.z=0</td>
<td>LAST.z=1</td>
</tr>
<tr>
<td><em>N</em></td>
<td>3</td>
<td>FIRST.x=0</td>
<td>LAST.x=1</td>
<td>FIRST.y=1</td>
<td>LAST.y=1</td>
<td>FIRST.z=1</td>
<td>LAST.z=1</td>
</tr>
<tr>
<td><em>N</em></td>
<td>4</td>
<td>FIRST.x=1</td>
<td>LAST.x=1</td>
<td>FIRST.y=1</td>
<td>LAST.y=1</td>
<td>FIRST.z=1</td>
<td>LAST.z=1</td>
</tr>
</tbody>
</table>

**Grouped by Y X Z**

<table>
<thead>
<tr>
<th><em>N</em></th>
<th>1</th>
<th>FIRST.y=1</th>
<th>LAST.y=0</th>
<th>FIRST.x=1</th>
<th>LAST.x=0</th>
<th>FIRST.z=1</th>
<th>LAST.z=0</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>N</em></td>
<td>2</td>
<td>FIRST.y=0</td>
<td>LAST.y=1</td>
<td>FIRST.x=0</td>
<td>LAST.x=1</td>
<td>FIRST.z=0</td>
<td>LAST.z=1</td>
</tr>
<tr>
<td><em>N</em></td>
<td>3</td>
<td>FIRST.y=1</td>
<td>LAST.y=0</td>
<td>FIRST.x=1</td>
<td>LAST.x=1</td>
<td>FIRST.z=1</td>
<td>LAST.z=1</td>
</tr>
<tr>
<td><em>N</em></td>
<td>4</td>
<td>FIRST.y=0</td>
<td>LAST.y=1</td>
<td>FIRST.x=1</td>
<td>LAST.x=1</td>
<td>FIRST.z=1</td>
<td>LAST.z=1</td>
</tr>
</tbody>
</table>

### Processing BY-Groups in the DATA Step

#### Overview

The most common use of BY-group processing is to combine data sets by using the BY statement with the SET, MERGE, MODIFY, or UPDATE statements. (If you use a SET, MERGE, or UPDATE statement with the BY statement, your observations must be grouped or ordered.) When processing these statements, SAS reads one observation at a time into the program data vector. With BY-group processing, SAS selects the observations from the data sets according to the values of the BY variable or variables. After processing all the observations from one BY group, SAS expects the next observation to be from the next BY group.
The BY statement modifies the action of the SET, MERGE, MODIFY, or UPDATE statement by controlling when the values in the program data vector are set to missing. During BY-group processing, SAS retains the values of variables until it has copied the last observation that it finds for that BY group in any of the data sets. Without the BY statement, the SET statement sets variables to missing when it reads the last observation. The MERGE statement does not set variables to missing after the DATA step starts reading observations into the program data vector.

### Processing BY-Groups Conditionally

You can process observations conditionally by using the subsetting IF or IF-THEN statements, or the SELECT statement, with the temporary variables FIRST.\textit{variable} and LAST.\textit{variable} (set up during BY-group processing). For example, you can use the IF or IF THEN statements to perform calculations for each BY group and to write an observation when the first or the last observation of a BY group has been read into the program data vector.

The following example computes annual payroll by department. It uses IF-THEN statements and the values of FIRST.\textit{variable} and LAST.\textit{variable} automatic variables to reset the value of PAYROLL to 0 at the beginning of each BY group and to write an observation after the last observation in a BY group is processed.

```sas
data salaries;
    input Department $ Name $ WageCategory $ WageRate;
datalines;
  BAD Carol Salaried 20000
  BAD Elizabeth Salaried 5000
  BAD Linda Salaried 7000
  BAD Thomas Salaried 9000
  BAD Lynne Hourly 230
  DDG Jason Hourly 200
  DDG Paul Salaried 4000
  PPD Kevin Salaried 5500
  PPD Amber Hourly 150
  PPD Tina Salaried 13000
  STD Helen Hourly 200
  STD Jim Salaried 8000
;
proc print data=salaries;
run;
proc sort data=salaries out=temp; by Department; run;
data budget (keep=Department Payroll);
    set temp;
    by Department;
    if WageCategory='Salaried' then YearlyWage=WageRate*12;
    else if WageCategory='Hourly' then YearlyWage=WageRate*2000;
    /* SAS sets FIRST.\textit{variable} to 1 if this is a new department in the BY group. */
    if first.Department then Payroll=0;
    Payroll+YearlyWage;
    /* SAS sets LAST.\textit{variable} to 1 if this is the last department in the current BY group. */
run;
```

**Processing BY-Groups in the DATA Step**
if last.Department;
run;

proc print data=budget;
   format Payroll dollar10.;
   title 'Annual Payroll by Department';
run;

Output 19.1  Output from Conditional BY-Group Processing

**Annual Payroll by Department**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Department</th>
<th>Payroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BAD</td>
<td>$952,000</td>
</tr>
<tr>
<td>2</td>
<td>DDG</td>
<td>$448,000</td>
</tr>
<tr>
<td>3</td>
<td>PPD</td>
<td>$522,000</td>
</tr>
<tr>
<td>4</td>
<td>STD</td>
<td>$496,000</td>
</tr>
</tbody>
</table>

Data Not in Alphabetic or Numeric Order

In BY-group processing, you can use data that is arranged in an order other than alphabetic or numeric, such as by calendar month or by category. To do this, use the NOTSORTED option in a BY statement when you use a SET statement. The NOTSORTED option in the BY statement tells SAS that the data is not in alphabetic or numeric order, but that it is arranged in groups by the values of the BY variable. You cannot use the NOTSORTED option with the MERGE statement, the UPDATE statement, or when the SET statement lists more than one data set.

This example assumes that the data is grouped by the character variable MONTH. The subsetting IF statement conditionally writes an observation, based on the value of LAST.month. This DATA step writes an observation only after processing the last observation in each BY group.

data sales;
   input month
data total_sale(drop=sales);
   set region.sales
      by month notsorted;
   total+sales;
   if last.month;
run;

Data Grouped by Formatted Values

Use the GROUPFORMAT option in the BY statement to ensure that
- formatted values are used to group observations when a FORMAT statement and a BY statement are used together in a DATA step

---

**Output 19.1**  Output from Conditional BY-Group Processing

**Annual Payroll by Department**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Department</th>
<th>Payroll</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BAD</td>
<td>$952,000</td>
</tr>
<tr>
<td>2</td>
<td>DDG</td>
<td>$448,000</td>
</tr>
<tr>
<td>3</td>
<td>PPD</td>
<td>$522,000</td>
</tr>
<tr>
<td>4</td>
<td>STD</td>
<td>$496,000</td>
</tr>
</tbody>
</table>
The FIRST.variable and LAST.variable are assigned by the formatted values of the variable.

The GROUPFORMAT option is valid only in the DATA step that creates the SAS data set. It is particularly useful with user-defined formats. The following examples illustrate the use of the GROUPFORMAT option.

Example 1: Using GROUPFORMAT with Formats

```sas
proc format;
  value range
    low -55 = 'Under 55'
    55-60 = '55 to 60'
    60-65 = '60 to 65'
    65-70 = '65 to 70'
    other = 'Over 70';
run;

proc sort data=class out=sorted_class;
  by height;
run;

data _null_
  format height range.;
  set sorted_class;
  by height groupformat;
  if first.height then
    put 'Shortest in ' height ' measures ' height:best12.;
run;
```

SAS writes the following output to the log:

Example Code 19.3  SAS Log Output Using the BY Statement GROUPFORMAT Option

```
Shortest
in Under 55 measures 51.3
Shortest in 55 to 60 measures 56.3
Shortest in 60 to 65 measures 62.5
Shortest in 65 to 70 measures 65.3
Shortest in Over 70 measures 72
```

Example 2: Using GROUPFORMAT with Formats

```sas
options
  linesize=80 pagesize=60;

  /* Create SAS data set test */
  data test;
    infile datalines;
    input name $ Score;
  datalines;
  Jon 1
  ./
```
Anthony 3
Miguel 3
Joseph 4
Ian 5
Jan 6
;
  /* Create a user-defined format */
proc format;
    value Range 1-2='Low'
      3-4='Medium'
      5-6='High';
run;

  /* Create the SAS data set newtest */
data newtest;
    set test;
    by groupformat Score;
    format Score Range.;
run;

  /* Print using formatted values */
proc print data=newtest;
  title 'Score Categories';
  var Name Score;
  by Score;
run;
### Score Categories

#### Score=Low

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jon</td>
<td>Low</td>
</tr>
</tbody>
</table>

#### Score=Medium

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Anthony</td>
<td>Medium</td>
</tr>
<tr>
<td>3</td>
<td>Miguel</td>
<td>Medium</td>
</tr>
<tr>
<td>4</td>
<td>Joseph</td>
<td>Medium</td>
</tr>
</tbody>
</table>

#### Score=High

<table>
<thead>
<tr>
<th>Obs</th>
<th>name</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Ian</td>
<td>High</td>
</tr>
<tr>
<td>6</td>
<td>Jan</td>
<td>High</td>
</tr>
</tbody>
</table>
Definition of WHERE-Expression Processing

WHERE-expression processing enables you to conditionally select a subset of observations, so that SAS processes only the observations that meet a set of specified conditions. For example, if you have a SAS data set that contains sales records, you might want to print just the subset of observations for which the sales are greater than $300,000 but less than $600,000. In addition, WHERE-expression processing can improve efficiency of a request. For example, if a WHERE expression can be optimized with an index, it is not necessary for SAS to read all observations in the data set in order to perform the request.

WHERE expression

defines a condition that selected observations must satisfy in order to be processed. You can have a single WHERE expression, referred to as a simple expression, such as the following:

where sales gt 600000;

Or you can have multiple WHERE expressions, referred to as a compound expression, such as the following:
Where to Use a WHERE Expression

In SAS, you can use a WHERE expression in the following situations:

- **WHERE statement in both DATA and PROC steps.** For example, the following PRINT procedure includes a WHERE statement so that only the observations where the year is greater than 2001 are printed:

  ```sas
  proc print data=employees;
  where startdate > '01jan2001'd;
  run;
  ```

- **WHERE= data set option.** The following PRINT procedure includes the WHERE= data set option:

  ```sas
  proc print data=employees (where=(startdate > '01jan2001'd));
  run;
  ```

- **WHERE clause in the SQL procedure, SCL, and SAS/IML software.** For example, the following SQL procedure includes a WHERE clause to select only the states where the murder count is greater than seven:

  ```sql
  proc sql;
  select state from crime
  where murder > 7;
  ```

- **WHERE command in windowing environments like SAS/FSP software:**

  ```sas
  where age > 15
  ```

- **SAS view (DATA step view, SAS/ACCESS view, PROC SQL view), stored with the definition.** For example, the following SQL procedure creates an SQL view named STAT from the data file Crime and defines a WHERE expression for the SQL view definition:

  ```sql
  proc sql;
  create view stat as
  select * from crime
  where murder > 7;
  ```

In some cases, you can combine the methods that you use to specify a WHERE expression. That is, you can use a WHERE statement as follows:

- in conjunction with a WHERE= data set option

- along with the WHERE= data set option in windowing procedures, and in conjunction with the WHERE command

- on a SAS view that has a stored WHERE expression

For example, it might be useful to combine methods when you merge data sets. That is, you might want different criteria to apply to each data set when you create a subset of data. However, when you combine methods to create a subset of data, there are some restrictions. For example, in the DATA step, if a WHERE statement and a WHERE= data set option apply to the same data set, the data set option takes precedence. For details, see the documentation for the method that you are using to specify a WHERE expression.
Note: By default, a WHERE expression does not evaluate added and modified observations. To specify whether a WHERE expression should evaluate updates, you can specify the WHEREUP= data set option. See the "WHEREUP= Data Set Option" in SAS Data Set Options: Reference.

Syntax of WHERE Expression

WHERE Expression Contents

A WHERE expression is a type of SAS expression that defines a condition for selecting observations. A WHERE expression can be as simple as a single variable name or a constant (which is a fixed value). A WHERE expression can be a SAS function, or it can be a sequence of operands and operators that define a condition for selecting observations. In general, the syntax of a WHERE expression is as follows:

WHERE operand <operator> <operand>

operand

something to be operated on. An operand can be a variable, a SAS function, or a constant. See “Specifying an Operand” on page 383.

operator

a symbol that requests a comparison, logical operation, or arithmetic calculation. All SAS expression operators are valid for a WHERE expression, which include arithmetic, comparison, logical, minimum and maximum, concatenation, parentheses to control order of evaluation, and prefix operators. In addition, you can use special WHERE expression operators. These expression operators include BETWEEN-AND, CONTAINS, IS NULL or IS MISSING, LIKE, sounds-like, and SAME-AND. See “Specifying an Operator” on page 386.

Specifying an Operand

Variable

A variable is a column in a SAS data set. Each SAS variable has attributes like name and type (character or numeric). The variable type determines how you specify the value for which you are searching. For example:

where score > 50;
where date >= '01jan2001'd and time >= '9:00't;
where state = 'Texas';

In a WHERE expression, you cannot use automatic variables created by the DATA step (for example, FIRST.variable, LAST.variable, _N_, or variables created in assignment statements).
As in other SAS expressions, the names of numeric variables can stand alone. SAS treats numeric values of 0 or missing as false; other values as true. In the following example, the WHERE expression returns all rows where EMPNUM is not missing and not zero and ID is not missing and not zero:

```
where empnum and id;
```

The names of character variables can also stand alone. SAS selects observations where the value of the character variable is not blank. For example, the following WHERE expression returns all values not equal to blank:

```
where lastname;
```

### SAS Function

A SAS function returns a value from a computation or system manipulation. Most functions use arguments that you supply, but a few obtain their arguments from the operating environment. To use a SAS function in a WHERE expression, enter its name and arguments enclosed in parentheses. Some functions that you might want to specify include:

- SUBSTR extracts a substring.
- TODAY returns the current date.
- PUT returns a given value using a given format.

The following DATA step produces a SAS data set that contains only observations from data set Customer in which the value of Name begins with `Mac` and the value of variable City is Charleston or Atlanta:

```sas
data testmacs;
  set customer;
  where substr (name,1,3) = 'Mac' and
      (city='Charleston' or city='Atlanta');
run;
```

The OF syntax is permitted in some SAS functions, but it cannot be used when using those functions that are specified in a WHERE clause. In the following DATA step example, OF can be used with RANGE.

```sas
data abc;
x1=2;
x2=3;
x3=4;
r=range(of x1-x3);
run;
```

When you use the WHERE clause with RANGE and OF, an error is written to the SAS log.
**Example Code 20.1**  Output When WHERE Clause Is Used with OF

```sas
proc print data=abc;
where range(of x1-x3)=6;
--
 22
 76
ERROR: Syntax error while parsing WHERE clause.
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &, {, *, **, +, ',', -, /, <, <=, <>, =, >, >=, ?, AND, BETWEEN, CONTAINS, EQ, GE, GT, LE, LIKE, LT, NE, OR, ^=, |, ||, -=.
ERROR 76-322: Syntax error, statement will be ignored.
run;
```

Below is a table of SAS functions that can use the OF syntax:

*Table 20.1*  SAS Functions That Use the OF Syntax

<table>
<thead>
<tr>
<th>CAT</th>
<th>HARMEANZ</th>
<th>RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATS</td>
<td>KURTOSIS</td>
<td>SKEWNESS</td>
</tr>
<tr>
<td>CATT</td>
<td></td>
<td>STD</td>
</tr>
<tr>
<td>CATX</td>
<td></td>
<td>STDERR</td>
</tr>
<tr>
<td>CSS</td>
<td></td>
<td>SUM</td>
</tr>
<tr>
<td>CV</td>
<td></td>
<td>USS</td>
</tr>
<tr>
<td>GEOMEAN</td>
<td></td>
<td>NMис</td>
</tr>
<tr>
<td>GEOMEANZ</td>
<td></td>
<td>ORDINAL</td>
</tr>
<tr>
<td>HARMEAN</td>
<td></td>
<td>RANGE</td>
</tr>
</tbody>
</table>

**Note:** The SAS functions that are used in a WHERE expression and can be optimized by an index are the SUBSTR function and the TRIM function.

For more information about SAS functions, see *SAS Functions and CALL Routines: Reference*.

**Constant**

A constant is a fixed value such as a number or quoted character string, that is, the value for which you are searching. A constant is a value of a variable obtained from the SAS data set, or values created within the WHERE expression itself. Constants are also called literals. For example, a constant could be a flight number or the name of a city. A constant can also be a time, date, or datetime value.

The value is either numeric or character. Note the following rules regarding whether to use quotation marks:

- If the value is numeric, do not use quotation marks.
  ```sas
  where price > 200;
  ```

- If the value is character, use quotation marks.
  ```sas
  where lastname eq 'Martin';
  ```
You can use either single or double quotation marks, but do not mix them. Quoted values must be exact matches, including case.

It might be necessary to use single quotation marks when double quotation marks appear in the value, or use double quotation marks when single quotation marks appear in the value.

where item = '6" decorative pot';
where name ? "D'Amico";

A SAS date constant must be enclosed in quotation marks. When you specify date values, case is not important. You can use single or double quotation marks. The following expressions are equivalent:

where birthday = '24sep1975'd;
where birthday = '24sep1975"d;

### Specifying an Operator

#### Arithmetic Operators

Arithmetic operators enable you to perform a mathematical operation. The arithmetic operators include the following:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiplication</td>
<td>where bonus = salary * .10;</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>where f = g/h;</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>where c = a+b;</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>where f = g-h;</td>
</tr>
<tr>
<td>**</td>
<td>exponentiation</td>
<td>where y = a**2;</td>
</tr>
</tbody>
</table>

#### Comparison Operators

Comparison operators (also called binary operators) compare a variable with a value or with another variable. Comparison operators propose a relationship and ask SAS to determine whether that relationship holds. For example, the following WHERE expression accesses only those observations that have the value 78753 for the numeric variable ZipCode:

where zipcode eq 78753;

The following table lists the comparison operators:
### Table 20.3 Comparison Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Mnemonic Equivalent</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>EQ</td>
<td>equal to</td>
<td>where empnum eq 3374;</td>
</tr>
<tr>
<td>^= or ~= or ~= or &lt;&gt;</td>
<td>NE</td>
<td>not equal to</td>
<td>where status ne full-time;</td>
</tr>
<tr>
<td>&gt;</td>
<td>GT</td>
<td>greater than</td>
<td>where hiredate gt '01jun1982'd;</td>
</tr>
<tr>
<td>&lt;</td>
<td>LT</td>
<td>less than</td>
<td>where empnum &lt; 2000;</td>
</tr>
<tr>
<td>&gt;=</td>
<td>GE</td>
<td>greater than or equal to</td>
<td>where empnum &gt;= 3374;</td>
</tr>
<tr>
<td>&lt;=</td>
<td>LE</td>
<td>less than or equal to</td>
<td>where empnum &lt;= 3374;</td>
</tr>
<tr>
<td>IN</td>
<td></td>
<td>equal to one from a list of values</td>
<td>where state in ('NC','TX');</td>
</tr>
</tbody>
</table>

When you do character comparisons, you can use the colon (:) modifier to compare only a specified prefix of a character string. For example, in the following WHERE expression, the colon modifier, used after the equal sign, tells SAS to look at only the first character in the values for variable LastName and to select the observations with names beginning with the letter S:

```sql
where lastname=: 'S';
```

Note that in the SQL procedure, the colon modifier that is used in conjunction with an operator is not supported; you can use the LIKE operator instead.

### IN Operator

The IN operator, which is a comparison operator, searches for character and numeric values that are equal to one from a list of values. The list of values must be in parentheses, with each character value in quotation marks and separated by either a comma or blank.

For example, suppose you want all sites that are in North Carolina or Texas. You could specify:

```sql
where state = 'NC' or state = 'TX';
```

However, it is easier to use the IN operator, which selects any state in the list:

```sql
where state in ('NC','TX');
```

In addition, you can use the NOT logical operator to exclude a list.

```sql
where state not in ('CA', 'TN', 'MA');
```
You can use a shorthand notation to specify a range of sequential integers to search. The range is specified by using the syntax M:N as a value in the list to search, where M is the lower bound and N is the upper bound. M and N must be integers, and M, N, and all the integers between M and N are included in the range. For example, the following statements are equivalent.

\[
\begin{align*}
y &= x \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}; \\
y &= x \in \{1:10\};
\end{align*}
\]

**Fully Bounded Range Condition**

A fully bounded range condition consists of a variable between two comparison operators, specifying both an upper and lower limit. For example, the following expression returns the employee numbers that fall within the range of 500 to 1000 (inclusive):

\[
\text{where } 500 \leq \text{empnum} \leq 1000;
\]

Note that the previous range condition expression is equivalent to the following:

\[
\text{where empnum} \geq 500 \text{ and empnum} \leq 1000;
\]

You can combine the NOT logical operator with a fully bounded range condition to select observations that fall outside the range. Note that parentheses are required:

\[
\text{where not } (500 \leq \text{empnum} \leq 1000);
\]

**BETWEEN-AND Operator**

The BETWEEN-AND operator is also considered a fully bounded range condition that selects observations in which the value of a variable falls within an inclusive range of values.

You can specify the limits of the range as constants or expressions. Any range that you specify is an inclusive range, so that a value equal to one of the limits of the range is within the range. The general syntax for using BETWEEN-AND is as follows:

\[
\text{WHERE variable BETWEEN value AND value;}
\]

For example:

\[
\begin{align*}
\text{where empnum between 500 and 1000;}
\text{where taxes between salary*0.30 and salary*0.50;}
\end{align*}
\]

You can combine the NOT logical operator with the BETWEEN-AND operator to select observations that fall outside the range:

\[
\text{where empnum not between 500 and 1000;}
\]

Note: The BETWEEN-AND operator and a fully bounded range condition produce the same results. That is, the following WHERE expressions are equivalent:

\[
\begin{align*}
\text{where 500} \leq \text{empnum} \leq 1000; \\
\text{where empnum between 500 and 1000;}
\end{align*}
\]
CONTAINS Operator

The most common usage of the CONTAINS (?) operator is to select observations by searching for a specified set of characters within the values of a character variable. The position of the string within the variable’s values does not matter. However, the operator is case sensitive when making comparisons.

The following examples select observations having the values Mobay and Brisbayne for the variable Company, but they do not select observations containing Bayview:

```
where company contains 'bay';
where company ? 'bay';
```

You can combine the NOT logical operator with the CONTAINS operator to select observations that are not included in a specified string:

```
where company not contains 'bay';
```

You can also use the CONTAINS operator with two variables, that is, to determine whether one variable is contained in another. When you specify two variables, keep in mind the possibility of trailing spaces, which can be resolved using the TRIM function.

```
proc sql;
  select *
    from table1 as a, table2 as b
    where a.fullname contains trim(b.lastname) and
      a.fullname contains trim(b.firstname);
```

In addition, the TRIM function is helpful when you search on a macro variable.

```
proc print;
  where fullname contains trim("&lname");
run;
```

IS NULL or IS MISSING Operator

The IS NULL or IS MISSING operator selects observations in which the value of a variable is missing. The operator selects observations with both regular or special missing value characters and can be used for both character and numeric variables.

```
where idnum is missing;
where name is null;
```

The following are equivalent for character data:

```
where name is null;
where name = ' '
```

And the following is equivalent for numeric data. This statement differentiates missing values with special missing value characters:

```
where idnum <= .Z;
```

You can combine the NOT logical operator with IS NULL or IS MISSING to select nonmissing values, as follows:

```
where salary is not missing;
```
LIKE Operator

The LIKE operator selects observations by comparing the values of a character variable to a specified pattern, which is referred to as pattern matching. The LIKE operator is case sensitive. There are two special characters available for specifying a pattern:

- **percent sign (%)**
  - specifies that any number of characters can occupy that position. The following WHERE expression selects all employees with a name that starts with the letter N. The names can be of any length.
  
  ```sql
  where lastname like 'N%';
  ```

- **underscore (_)**
  - matches just one character in the value for each underscore character. You can specify more than one consecutive underscore character in a pattern, and you can specify a percent sign and an underscore in the same pattern. For example, you can use different forms of the LIKE operator to select character values from this list of first names:

    - Diana
    - Diane
    - Dianna
    - Dianthus
    - Dyan

  The following table shows which of these names is selected by using various forms of the LIKE operator:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Name Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>like 'D_an'</td>
<td>Dyan</td>
</tr>
<tr>
<td>like 'D_an_'</td>
<td>Diana, Diane</td>
</tr>
<tr>
<td>like 'D_an__'</td>
<td>Dianna</td>
</tr>
<tr>
<td>like 'D_an%'</td>
<td>all names from list</td>
</tr>
</tbody>
</table>

You can use a SAS character expression to specify a pattern, but you cannot use a SAS character expression that uses a SAS function.

You can combine the NOT logical operator with LIKE to select values that do not have the specified pattern, such as the following:

```sql
where firstname not like 'D_an%';
```

Because the % and _ characters have special meaning for the LIKE operator, you must use an escape character when searching for the % and _ characters in values. An escape character is a single character that, in a sequence of characters, signifies that what follows takes an alternative meaning. For the LIKE operator, an escape character signifies to search for literal instances of the % and _ characters in the variable's values instead of performing the special-character function.
For example, if the variable X contains the values abc, a_b, and axb, the following LIKE operator with an escape character selects only the value a_b. The escape character (/) specifies that the pattern searches for a literal '_ .' that is surrounded by the characters a and b. The escape character (/) is not part of the search.

where x like 'a/_b' escape '/';

Without an escape character, the following LIKE operator would select the values a_b and axb. The special character underscore in the search pattern matches any single b character, including the value with the underscore:

where x like 'a_b';

To specify an escape character, include the character in the pattern-matching expression, and then the keyword ESCAPE followed by the escape-character expression. When you include an escape character, the pattern-matching expression must be enclosed in quotation marks, and it cannot contain a column name. The escape-character expression evaluates to a single character. The operands must be character or string literals. If it is a single character, enclose it in quotation marks.

LIKE 'pattern-matching-expression' ESCAPE 'escape-character-expression'

Sounds-like Operator

The sounds-like (=*) operator selects observations that contain a spelling variation of a specified word or words. The operator uses the Soundex algorithm to compare the variable value and the operand. For more information, see the SOUNDEX function in SAS Functions and CALL Routines: Reference.

Note: Note that the SOUNDEX algorithm is English-biased, and is less useful for languages other than English.

Although the sounds-like operator is useful, it does not always select all possible values. For example, consider that you want to select observations from the following list of names that sound like Smith:

- Schmitt
- Smith
- Smithson
- Smitt
- Smythe

The following WHERE expression selects all the names from this list except Smithson:

where lastname=* 'Smith';

You can combine the NOT logical operator with the sounds-like operator to select values that do not contain a spelling variation of a specified word or words, such as:

where lastname not =* 'Smith';

Note: The sounds-like operator cannot be optimized with an index.
SAME-AND Operator

Use the SAME-AND operator to add more conditions to an existing WHERE expression later in the program without retyping the original conditions. This capability is useful with the following:

- interactive SAS procedures
- full-screen SAS procedures that enable you to enter a WHERE expression on the command line
- any type of RUN-group processing

Use the SAME-AND operator when you already have a WHERE expression defined and you want to insert additional conditions. The SAME-AND operator has the following form:

```
where-expression-1;
...
SAS statements...
WHERE SAME AND where-expression-2;
...
SAS statements...
WHERE SAME AND where-expression-n;
```

SAS selects observations that satisfy the conditions after the SAME-AND operator in addition to any previously defined conditions. SAS treats all of the existing conditions as if they were conditions separated by AND operators in a single WHERE expression.

The following example shows how to use the SAME-AND operator within RUN groups in the GPLOT procedure. The SAS data set YEARS has three variables and contains quarterly data for the 2009–2011 period:

```
proc gplot data=years;
   plot unit*quar=year;
run;

   where year > '01jan2009'd;
run;

   where same and year < '01jan2012'd;
run;
```

The following WHERE expression is equivalent to the preceding code:

```
where year > ''01jan2009'd and year < '01jan2012'd;
```

MIN and MAX Operators

Use the MIN or MAX operators to find the minimum or maximum value of two quantities. Surround the operators with the two quantities whose minimum or maximum value you want to know.

- The MIN operator returns the lower of the two values.
- The MAX operator returns the higher of two values.

For example, if A is less than B, then the following would return the value of A:

```
where x = (a min b);
```
Note: The symbol representation `<>` is not supported, and `<>` is interpreted as “not equal to.”

Concatenation Operator

The concatenation operator concatenates character values. You indicate the concatenation operator as follows:

- `||` (two OR symbols)
- `!!` (two exclamation marks)
- `¦¦` (two broken vertical bars).

For example:

```sql
where name = 'John' || 'Smith';
```

Prefix Operators

The plus sign (+) and minus sign (−) can be either prefix operators or arithmetic operators. They are prefix operators when they appear at the beginning of an expression or immediately preceding an open parenthesis. A prefix operator is applied to the variable, constant, SAS function, or parenthetic expression.

```sql
where z = −(x + y);
```

Note: The NOT operator is also considered a prefix operator.

Combining Expressions By Using Logical Operators

Syntax

You can combine or modify WHERE expressions by using the logical operators (also called Boolean operators) AND, OR, and NOT. The basic syntax of a compound WHERE expression is as follows:

```
WHERE where-expression-1 AND | OR | NOT where-expression-n
```

AND combines two conditions by finding observations that satisfy both conditions. For example:

```sql
where skill eq 'java' and years eq 4;
```

OR combines two conditions by finding observations that satisfy either condition or both. For example:

```sql
where skill eq 'java' or years eq 4;
```
NOT modifies a condition by finding the complement of the specified criteria. You can use the NOT logical operator in combination with any SAS and WHERE expression operator. And you can combine the NOT operator with AND and OR. For example:

where skill not eq 'java' or years not eq 4;

The logical operators and their equivalent symbols are shown in the following table:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Mnemonic Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
</tr>
<tr>
<td>! or</td>
<td>or</td>
</tr>
<tr>
<td>^ or ~ or ¬</td>
<td>NOT</td>
</tr>
</tbody>
</table>

Processing Compound Expressions

When SAS encounters a compound WHERE expression (multiple conditions), the software follows rules to determine the order in which to evaluate each expression. When WHERE expressions are combined, SAS processes the conditions in a specific order:

1. The NOT expression is processed first.
2. Then the expressions joined by AND are processed.
3. Finally, the expressions joined by OR are processed.

Using Parentheses to Control Order of Evaluation

Even though SAS evaluates logical operators in a specific order, you can control the order of evaluation by nesting expressions in parentheses. That is, an expression enclosed in parentheses is processed before one not enclosed. The expression within the innermost set of parentheses is processed first, followed by the next deepest, moving outward until all parentheses have been processed.

For example, suppose you want a list of all the Canadian sites that have both SAS/GRAPH and SAS/STAT software, so you issue the following expression:

where product='GRAPH' or product='STAT' and country='Canada';

The result, however, includes all sites that license SAS/GRAPH software along with the Canadian sites that license SAS/STAT software. To obtain the correct results, you can use parentheses, which causes SAS to evaluate the comparisons within the parentheses first, providing a list of sites with either product licenses, then the result is used for the remaining condition:

where (product='GRAPH' or product='STAT') and country='Canada';
Improving Performance of WHERE Processing

Indexing a SAS data set can significantly improve the performance of WHERE processing. An index is an optional file that you can create for SAS data files in order to provide direct access to specific observations.

Processing a WHERE expression without an index requires SAS to sequentially read observations in order to find the ones that match the selection criteria. Without an index, SAS first checks for the sort indicator, which is stored with the data file from a previous SORT procedure or SORTEDBY= data set option. If the sort indicator is validated, SAS takes advantage of it and stops reading the file once it is clear there are no more values that satisfy the WHERE expression. For example, consider a data set that is sorted by Age, without an index. To process the expression `where age le 25`, SAS stops reading observations after it finds an observation that is greater than 25. Note that while SAS can determine when to stop reading observations, without an index, there is no indication where to begin, so SAS always begins with the first observation, which can require reading a lot of observations.

Having an index enables SAS to determine which observations satisfy the criteria, which is referred to as optimizing the WHERE expression. However, by default, SAS decides whether to use the index or read the entire data set sequentially. For details about how SAS uses an index to process a WHERE expression, see "Using an Index for WHERE Processing" in SAS Language Reference: Concepts.

In addition to creating indexes for the data set, here are some guidelines for writing efficient WHERE expressions:

<table>
<thead>
<tr>
<th>Guideline</th>
<th>Efficient</th>
<th>Inefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoid using the LIKE operator that begins with % or _.</td>
<td><code>where country like 'A%INA'</code>;</td>
<td><code>where country like '%INA'</code>;</td>
</tr>
<tr>
<td>Avoid using arithmetic expressions.</td>
<td><code>where salary &gt; 48000;</code></td>
<td><code>where salary &gt; 12*4000;</code></td>
</tr>
</tbody>
</table>
Processing a Segment of Data That Is Conditionally Selected

Applying FIRSTOBS= and OBS= Options

When you conditionally select a subset of observations with a WHERE expression, you can also segment that subset by applying FIRSTOBS=, OBS=, or both processing (as data set options and system options). When used with a WHERE expression,

- FIRSTOBS= specifies the observation number within the subset of data selected by the WHERE expression to begin processing.
- OBS= specifies when to stop processing observations from the subset of data selected by the WHERE expression.

When used with a WHERE expression, the values specified for OBS= and FIRSTOBS= are not the physical observation number in the data set, but a logical number in the subset. For example, obs=3 does not mean the third observation number in the data set. Instead, it means the third observation in the subset of data selected by the WHERE expression.

Applying OBS= and FIRSTOBS= processing to a subset of data is supported for the WHERE statement, WHERE= data set option, and WHERE clause in the SQL procedure.

If you are processing a SAS view that is a view of another view (nested views), applying OBS= and FIRSTOBS= to a subset of data could produce unexpected results. For nested views, OBS= and FIRSTOBS= processing is applied to each SAS view, starting with the root (lowest-level) view, and then filtering observations for each SAS view. The result could be that no observations meet the subset and segment criteria. See “Processing a SAS View” on page 397.

Applying FIRSTOBS= and OBS= to a Subset of Data

The following SAS program illustrates how to specify a condition to subset data, and how to specify a segment of the subset of data to process.

data A;
  do I=1 to 100;
    X=I + 1;
    output;
  end;
run;

proc print data=work.a (firstobs=2 obs=4);
  where I > 90;
run;
The DATA step creates a data set named Work.A containing 100 observations and two variables: I and X.

The WHERE expression I > 90 tells SAS to process only the observations that meet the specified condition, which results in the subset of observations 91 through 100.

The FIRSTOBS= data set option tells SAS to begin processing with the 2nd observation in the subset of data, which is observation 92.

The OBS= data set option tells SAS to stop processing when it reaches the 4th observation in the subset of data, which is observation 94.

The result of PROC PRINT is observations 92, 93, and 94.

---

Processing a SAS View

The following SAS program creates a data set, a SAS view for the data set, then a second SAS view that subsets data from the first SAS view. Both a WHERE statement and the OBS= system option are used.

data a;
  do I=1 to 100;
    X=I + 1;
    output;
  end;
run;

data viewa/view=viewa;
set a;
  Z = X+1;
run;

data viewb/view=viewb;
set viewa;
  where I > 90;
run;

options obs=3;

proc print data=work.viewb;
run;

---

1 The first DATA step creates a data set named Work.A, which contains 100 observations and two variables: I and X.

2 The second DATA step creates a SAS view named Work.ViewA containing 100 observations and three variables: I, X (from data set Work.A), and Z (assigned in this DATA step).

3 The third DATA step creates a SAS view named Work.ViewB and subsets the data with a WHERE statement, which results in the view accessing ten observations.
The OBS= system option applies to the previous SET ViewA statement, which tells SAS to stop processing when it reaches the 3rd observation in the subset of data being processed.

When SAS processes the PRINT procedure, the following occurs:

1. First, SAS applies obs=3 to Work.ViewA, which stops processing at the 3rd observation.
2. Next, SAS applies the condition \( I > 90 \) to the three observations being processed. None of the observations meet the criteria.
3. PROC PRINT results in no observations.

To prevent the potential of unexpected results, you can specify obs=max when creating Work.ViewA to force SAS to read all the observations in the root (lowest-level) view:

```sas
data viewa/view=viewa;
set a (obs=max);
Z = X+1;
run;
```

The PRINT procedure processes observations 91, 92, and 93.

---

Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement

To conditionally select observations from a SAS data set, you can use either a WHERE expression or a subsetting IF statement. They both test a condition to determine whether SAS should process an observation. However, they differ as follows:

- The subsetting IF statement can be used only in a DATA step. A subsetting IF statement tests the condition after an observation is read into the Program Data Vector (PDV). If the condition is true, SAS continues processing the current observation. Otherwise, the observation is discarded, and processing continues with the next observation.

- You can use a WHERE expression in both a DATA step and SAS procedures, as well as in a windowing environment, SCL programs, and as a data set option. A WHERE expression tests the condition before an observation is read into the PDV. If the condition is true, the observation is read into the PDV and processed. If the condition is false, the observation is not read into the PDV, and processing continues with the next observation. This can yield substantial savings when observations contain many variables or very long character variables (up to 32K bytes). In addition, a WHERE expression can be optimized with an index, and the WHERE expression enables more operators, such as LIKE and CONTAINS.

Note: Although it is generally more efficient to use a WHERE expression and avoid the move to the PDV before processing, if the data set contains observations with very few variables, the move to the PDV could be cheap. However, one variable containing 32K bytes of character data is not cheap, even though it is only one variable.
In most cases, you can use either method. However, the following table provides a list of tasks that require you to use a specific method:

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Make the selection in a procedure without using a preceding DATA step</td>
<td>WHERE expression</td>
</tr>
<tr>
<td>Take advantage of the efficiency available with an indexed data set</td>
<td>WHERE expression</td>
</tr>
<tr>
<td>Use one of a group of special operators, such as BETWEEN-AND, CONTAINS, IS MISSING or IS NULL, LIKE, SAME-AND, and Sounds-Like</td>
<td>WHERE expression</td>
</tr>
<tr>
<td>Base the selection on anything other than a variable value that already exists in a SAS data set. For example, you can select a value that is read from raw data, or a value that is calculated or assigned during the course of the DATA step</td>
<td>subsetting IF</td>
</tr>
<tr>
<td>Make the selection at some point during a DATA step rather than at the beginning</td>
<td>subsetting IF</td>
</tr>
<tr>
<td>Execute the selection conditionally</td>
<td>subsetting IF</td>
</tr>
</tbody>
</table>
Combining Data

Overview of Combining Data ........................................................................................................... 402
  Definition of Combining Data ......................................................................................................... 402
  Terms ........................................................................................................................................... 402
  Data Relationships ......................................................................................................................... 403
  Summary of Ways to Combine SAS Data Sets ................................................................................ 406
  Concatenating ............................................................................................................................... 407
  Interleaving ................................................................................................................................... 408
  Match-Merging ............................................................................................................................... 409
  One-to-One Merging ....................................................................................................................... 410
  One-to-One Reading ....................................................................................................................... 411
  Hash Table Merging ....................................................................................................................... 412
  Updating ......................................................................................................................................... 413
  Modifying ....................................................................................................................................... 414

Comparing Methods ........................................................................................................................ 415
  PROC SQL versus Match-Merging ................................................................................................. 415
  Updating and Modifying Comparison ............................................................................................ 416
  One-to-One Reading versus One-to-One Merging ....................................................................... 418
  Comparison of SAS Language Elements for Combining Data Sets ............................................. 420

Examples: Prepare Data .................................................................................................................... 423
  Example: Examine the Data to Be Combined ................................................................................ 423
  Example: Find the Common Variables in Multiple Input Data Sets ............................................. 425
  Example: Creating Unique BY Values When Data Contains Duplicate BY Values .................... 428
  Example: Prepare Data in Which Common Variables Have Different Data Types ..................... 430
  Example: Prepare Data in Which Common Variables Have Different Lengths ............................. 433
  Example: Prepare Data in Which Common Variables Have Different Formats and Labels .......... 435
  Example: Prepare Data in Which Common Variables Represent Different Data .......................... 437

Examples: Concatenate Data ............................................................................................................ 438
  Example: Concatenate Using the SET Statement ......................................................................... 438
  Example: Concatenate Using PROC SQL .................................................................................... 440
  Example: Append Files Using PROC APPEND ......................................................................... 442
  Example: Add a Message to the Log When Variables Are Missing from Input Data Sets .......... 443

Examples: Interleave Data ................................................................................................................ 446
  Example: Interleave Data Sets ....................................................................................................... 446
  Example: Interleave with Duplicate Values of the BY Variable ................................................... 447
  Example: Interleave with Different Values of the Common BY Variable .................................... 451

Examples: Match-Merge Data .......................................................................................................... 453
  Example: Match-Merge Observations Based on a BY Variable ..................................................... 453
Overview of Combining Data

Definition of Combining Data

To combine SAS data sets means to read data from multiple input data sets and to combine them using one of the methods outlined in this section. For information about how to read SAS data sets, see "Ways to Read and Create SAS Data Sets" on page 288.

Terms

The following terms are used throughout this chapter and are defined here in the context of combining data:

common variable
a variable that is present in all of the input data sets that are being combined. A common variable can be specified as the BY variable, but it does not have to be. Variables are recognized as the same, or common to multiple data sets if they have the same name regardless of the use of upper or lowercase letters in the
name. For example, SAS considers the variable name in one data set to be identical to the variable NAME in another data set. In addition to having the same name, common variables must also have the same data type. See “Example: Prepare Data in Which Common Variables Have Different Data Types” on page 430 for more information.

**BY variable**
a variable (or variables) whose values serve as the basis for how rows are merged when you combine multiple data sets into one. BY variables are specified in the BY statement in SAS. BY variables are always common variables in the context of combining data. The BY variables are the variables named in the BY statement of the MERGE, UPDATE, SET, or MODIFY statements when combining data. See “Example: Examine the Data to Be Combined” on page 423 and “Example: Creating Unique BY Values When Data Contains Duplicate BY Values” on page 428 for more information.

**BY value**
the value that a BY variable has in a particular row or observation.

**BY group**
a group of rows that have the same value for a variable that is specified in a BY statement. If more than one variable is specified in a BY statement, then the BY group is a group of rows that have a unique combination of values for those variables.

**data relationships**
associations between data sets or tables that exist in one of three ways: one-to-one, one-to-many (or, many-to-one), and many-to-many. The data sets are associated by common data, either at the physical or logical level. For example, in a business database, employee data and department data could be related through an Employee_ID variable that shares common values. Another data set could contain a numeric sequence of numbers whose partial values logically relate it to a separate data set by row number. See “Data Relationships” for more information about data relationships and how they relate to combining data.

**matched rows**
rows in input data sets that are selected for output when the values match the selected criteria. The matching criteria can be based on row number (or position within the data set) or it can be based on the values of variables that are specified by the user in a BY statement.

**unmatched rows**
rows in input data sets that are not selected for output because the values do not match the specified criteria. The matching criteria can be based on row number (or position within the data set) or it can be based on the values of variables that you specify in a BY statement.

---

**Data Relationships**

The following four categories are characterized by how rows relate among the data sets. All related data fall into one of these categories. You must be able to identify the existing relationships in your data, because this knowledge is crucial to understanding how input data can be processed to produce desired results.
One-to-One Relationship

In a one-to-one relationship, a single row in one data set is related to only one row in a second data set, and a single row in the second data set is related to only one row in the first data set. The relationship is based on the values of one or more selected variables. A one-to-one relationship implies that there are no duplicate values of the selected variable in each data set. When you work with multiple selected variables, this relationship implies that each combination of values occurs no more than once in each data set.

In the figure, rows in data sets Animal and Plant are related by matching values for the variable Common. The values for the variable Common are unique in both data sets.

---

One-to-Many Relationship

A one-to-many relationship between input data sets implies that one data set has at most one row with a specific value of the selected variable, but the other input data set can have more than one, or duplicates, of each value. When you work with multiple selected variables, this relationship implies that each combination of values occurs no more than once in one data set. The combination can occur more than once in the other data set. The order in which the input data sets are processed determines whether the relationship is one-to-many or many-to-one.

In the figure, rows in data sets Animal and Plant are related by common values for the variable Common. Values for the variable Common are unique in the data set Animal but not in data set Plant.
One-to-Many and Many-to-One Relationships

A one-to-many or many-to-one relationship between input data sets implies that one data set has at most one row with a specific value of the selected variable, but the other input data set can have more than one occurrence of each value. When you work with multiple selected variables, this relationship implies that each combination of values occurs no more than once in one data set. However, the combination can occur more than once in the other data set. The order in which the input data sets are processed determines whether the relationship is one-to-many or many-to-one.

In the figure, rows in data sets Animal, Plant, and Mineral are related by common values for the variable Common. Values for Common are unique in data sets Animal and Mineral but not in Plant. A one-to-many relationship exists between rows in data sets Animal and Plant and a many-to-one relationship exists between rows in data sets Plant and Mineral.

Many-to-Many Relationship

The many-to-many category implies that multiple rows from each input data set can be related based on the values of one or more common variables.

In the figure, rows in data sets Animal and Plant are related by common values for the variable Common. Values for the variable Common are not unique in either data set. A many-to-many relationship exists between rows in these data sets for values a and c.
### Summary of Ways to Combine SAS Data Sets

**Table 21.1  Summary of Ways to Combine Data Sets**

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Statement or Procedure</th>
<th>Simplified Example Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concatenate</td>
<td>Concatenating Using the SET Statement</td>
<td>SET statement</td>
<td><code>data concat; set data1 data2; run;</code></td>
</tr>
<tr>
<td></td>
<td>Concatenating using PROC SQL</td>
<td>SQL procedure</td>
<td><code>proc sql; select * from data1 union all corresponding select * from data2; quit;</code></td>
</tr>
<tr>
<td></td>
<td>Appending using PROC APPEND</td>
<td>APPEND procedure</td>
<td><code>proc append base=data1 data=data2; run;</code></td>
</tr>
<tr>
<td>Interleave</td>
<td>Interleaving</td>
<td>SET statement with BY statement</td>
<td><code>data interleav; set data1 data2; by year; run;</code></td>
</tr>
<tr>
<td>Merge</td>
<td>Match-Merging</td>
<td>MERGE statement with BY statement</td>
<td><code>data matchmerged; merge data1 data2; by year; run;</code></td>
</tr>
<tr>
<td></td>
<td>One-to-One Merging</td>
<td>MERGE statement</td>
<td><code>data merged; merge data1 data2; run;</code></td>
</tr>
<tr>
<td></td>
<td>One-to-One Reading</td>
<td>SET statement</td>
<td><code>data merged; set data1; set data2; run;</code></td>
</tr>
<tr>
<td></td>
<td>Hash Table Merging</td>
<td>HASH object with FIND method</td>
<td><code>declare hash h(dataset:'data1'); rc = h.definekey('key'); rc = h.definedata('data1'); h.deissuedone(); set data2; h.find(); run;</code></td>
</tr>
<tr>
<td></td>
<td>SQL procedure</td>
<td>PROC SQL Statement</td>
<td><code>proc sql; select * from data1, data2; by year; where data1.key=data2.key; run;</code></td>
</tr>
<tr>
<td>Type</td>
<td>Method</td>
<td>Statement or Procedure</td>
<td>Simplified Example Code</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------</td>
<td>------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Update</td>
<td>Updating</td>
<td>UPDATE statement with BY statement</td>
<td>data master;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>update master trans; by year;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>run;</code></td>
</tr>
<tr>
<td>Modifying</td>
<td>MODIFY</td>
<td>MODIFY statement with BY statement</td>
<td>data master;</td>
</tr>
<tr>
<td></td>
<td>statement</td>
<td></td>
<td><code>modify master trans; by year;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>run;</code></td>
</tr>
<tr>
<td>Hash Table Updating</td>
<td>HASH object</td>
<td>HASH object with REPLACE method</td>
<td>declare hash h(dataset:'data1');</td>
</tr>
<tr>
<td></td>
<td>with REPLAC</td>
<td></td>
<td><code>rc = h.definekey('key');</code></td>
</tr>
<tr>
<td></td>
<td>e method</td>
<td></td>
<td><code>rc = h.definedata('data1');</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>h.definedone();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>set data2;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>rc = h.find();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>if rc = 0 then h.replace();</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><code>run;</code></td>
</tr>
</tbody>
</table>

## Concatenating

### Definition
Concatenating combines two or more SAS data sets, one after the other, into a single, new output data set.

### Syntax
```sas
SET <data-set-1> <data-set-2><… data-set-n>;
```

### Concatenating Two Data Sets

<table>
<thead>
<tr>
<th>Data1</th>
<th>Data2</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year</td>
<td>Year</td>
<td>Year</td>
</tr>
<tr>
<td>2011</td>
<td>2011</td>
<td>2011</td>
</tr>
<tr>
<td>2012</td>
<td>2012</td>
<td>2012</td>
</tr>
<tr>
<td>2013</td>
<td>2013</td>
<td>2013</td>
</tr>
<tr>
<td>2014</td>
<td>2014</td>
<td>2014</td>
</tr>
<tr>
<td>2015</td>
<td>2015</td>
<td>2015</td>
</tr>
</tbody>
</table>

In the figure, the rows from **Data2** are appended to the rows from **Data1**.

### Details
- Concatenating using the SET statement processes the input data sets sequentially, in the order in which they are listed in the SET statement.
- Concatenating does not require that input data sets contain the same variables.
- Concatenating creates output that contains all the rows from all the input data sets and all the columns from all the input data sets.
- Concatenating sets the values of variables that do not exist in all input data sets to missing.

**See Also**
- “Examples: Concatenate Data” on page 438
- “Concatenating SAS Data Sets” in SAS DATA Step Statements: Reference

### Interleaving

**Definition**
Interleaving combines two or more SAS data sets into a single data set by interspersing their rows with one another based on the values of common BY variables. In some programming languages, the term merge means to interleave rows. However, rows that are interleaved in SAS data sets are not merged; they are copied from the original data sets in the order of the values of the BY variable.

**Syntax**

```sas
SET <data-set-1> <data-set-2> <...data-set-n>;
BY variable-1 <...variable-n>;
```

**Details**
- Interleaving requires the SET statement together with the **BY** statement, which specifies one or more common variables by which rows are matched.
- The input data sets must be indexed on or sorted by the **BY** variables.
- Interleaving returns rows that are ordered within each **BY** group by their positions in the original input data set.

**See Also**
- “Examples: Interleave Data” on page 446
- “Interleaving SAS Data Sets” in SAS DATA Step Statements: Reference

---

**Interleaving Two Data Sets**

In the figure, rows from Data2 are interspersed between rows from Data1.
Match-Merging

**Definition**
Match-Merging combines rows from two or more input data sets into a single row in an output data set based on the values of the BY variable.

**Syntax**
```
MERGE data-set-1 <data-set-2><...data-set-n>
BY variable-1 <...variable-n>
;```

**Details**
- Match-merging requires the MERGE statement together with the **BY** statement, which specifies one or more common variables by which rows are matched.
- Match-merging requires that input data sets contain indexes or that they are sorted by the values of the BY variables.
- The variables in the **BY** statement must be common to all input data sets.
- Only one **BY** statement can accompany each MERGE statement in a DATA step.
- The MERGE, **UPDATE**, and **MODIFY** statements can be used to combine and update rows in SAS data sets. For a comparison of these methods, see “Updating and Modifying Comparison” on page 416.

**See Also**
- “Examples: Match-Merge Data” on page 453
- “Match-Merging” in SAS DATA Step Statements: Reference
One-to-One Merging

**Definition**

Merging data one-to-one combines rows from multiple input data sets into a single row in a new output data set. Rows are combined based on their positions in the input data sets. The first row in the first input data set is combined with the first row in the second input data set, and so on.

**Syntax**

```mergesyntax
MERGE data-set-1 <data-set-2><… data-set-n>;
```

**One-to-One Merging of Two Data Sets**

<table>
<thead>
<tr>
<th>Data1</th>
<th>Data2</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>x1</td>
<td>y1</td>
<td>x1 y1</td>
</tr>
<tr>
<td>x2</td>
<td>y2</td>
<td>x2 y2</td>
</tr>
<tr>
<td>x3</td>
<td>y3</td>
<td>x3 y3</td>
</tr>
<tr>
<td>y4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y5</td>
<td></td>
<td>y5</td>
</tr>
</tbody>
</table>

In the figure, rows from `Data2` are merged with rows from `Data1` based on row number.

**Details**

- One-to-one merging requires the MERGE statement without the BY statement. There is no key variable on which to base the merge. Instead, rows are merged implicitly by row number.
- If the input data sets contain common variables, SAS replaces the values from the first data set that is read with values from the last data set that is read.
- SAS does not stop reading rows until it has read all rows from all input data sets. Compare this to one-to-one reading, in which SAS stops reading rows when it has read the last row from the smallest input data set.
- The output data set contains all variables from all input data sets. The number of rows in the output data set equals the number of rows in the largest input data set.

**See Also**

- “Examples: Merge Data One-to-One” on page 470
- “One-to-One Merging” in SAS DATA Step Statements: Reference
- Example 1: One-to-One Merging
- “One-to-One Reading versus One-to-One Merging” on page 418
One-to-One Reading

Definition

One-to-one reading combines rows from multiple input data sets into a single row in a new data set. Rows are matched based on their positions in the data sets. The first row in the first data set is combined with the first row in the second data set, and so on.

Syntax

SET <data-set-1>;
SET <data-set-2>;

One-to-One Reading of Two Data Sets

<table>
<thead>
<tr>
<th>Data1</th>
<th>Data2</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>y1</td>
<td>x1</td>
</tr>
<tr>
<td>x2</td>
<td>y2</td>
<td>x2</td>
</tr>
<tr>
<td>x3</td>
<td>y3</td>
<td>x3</td>
</tr>
<tr>
<td></td>
<td>y4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>y5</td>
<td></td>
</tr>
</tbody>
</table>

In the figure, rows from Data2 are merged with rows from Data1 based on row number.

Details

- One-to-one reading requires multiple SET statements without a BY statement. There is no key BY variable on which to merge the data sets. Like One-to-One Merging, rows are merged implicitly by row number rather than by the value of a BY variable.

- If the input data sets contain common variables, SAS replaces the values of the common variables from the first data set that is read with values from the last data set that is read. There is no key BY variable on which to merge rows, so they are combined implicitly by row number.

- SAS stops reading rows from input data sets after it has read the last row from the smallest input data set. Compare this to One-to-One Merging, in which SAS does not stop reading until it has read all rows from all input data sets.

- The output data set contains all variables from all input data sets. The number of rows in the output data set equals the number of rows in the smallest input data set.

See Also

- “Examples: Combine Data One-to-One” on page 478
- “Performing a Table Lookup When the Master File Contains Duplicate Observations” in SAS DATA Step Statements: Reference
- “Performing a Table Lookup” in SAS DATA Step Statements: Reference
- “One-to-One Reading versus One-to-One Merging” on page 418
Hash Table Merging

Definition
The DATA step hash object is a temporary, in-memory data structure that enables you to efficiently store, search, and retrieve data based on lookup keys. Hash objects are only available for the duration of the DATA step. Hash objects are initialized through the DECLARE statement. Methods such as FIND, ADD, and OUTPUT operate on hash objects. The contents of the hash object can be written to a data table by using the hash OUTPUT method. In-memory processing with hash tables is usually faster than other DATA step methods.

Syntax
DECLARE object hash-table-name

Hash Table Merging of Two Data Sets

<table>
<thead>
<tr>
<th>Data1</th>
<th>Data2</th>
<th>HashCombined</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>x1</td>
<td>y1</td>
<td>x1</td>
</tr>
<tr>
<td>x2</td>
<td>y2</td>
<td>x2</td>
</tr>
<tr>
<td>x3</td>
<td>y3</td>
<td>x3</td>
</tr>
<tr>
<td>y4</td>
<td></td>
<td>y4</td>
</tr>
<tr>
<td>y5</td>
<td></td>
<td>y5</td>
</tr>
</tbody>
</table>

In this example, the data set written by the DATA statement contains the observations from the data set that is being read, along with the data variables enumerated in the DEFINEDATA method for those observations that have matched on the key variable or variables.

Details
- Hash objects are available only for the duration of the DATA step.
- You can output the contents of a hash object to a data table by using the OUTPUT method.
- The data set that is written by the DATA statement contains the observations from the data set that is being read, along with the data variables named in the DEFINEDATA method for those observations that match the key variable or variables.
- Sorting is not required when using hash tables to merge data sets.
- Hash-table merging enables fast processing on large tables, as long as the hash table can be held in memory.
- The same variable name does not have to be used for the key or keys in both data sets. The match can be done on different variables.

See Also
- “Example: Merge Data Using a Hash Table” on page 480
- SAS Component Objects: Reference
Updating

Definition

The UPDATE statement creates a new, updated data set by using rows from a transaction data set to change the values of corresponding rows in a master data set. The master data set contains the original information, and the transaction data set contains the new information that needs to be applied to the master data set.

Syntax

```
UPDATE master-data-set transaction-data-set <(data-set-options)>
BY by-variable;
```

In the figure, a new output data set, Combined, is created by updating rows in the Master data set based on values in the Trans data set.

Details

- The UPDATE statement requires a BY statement that specifies one or more common variables by which rows are matched.
- The input data sets must be indexed on or sorted by the BY variables.
- Values of the BY variable must be unique for each observation in the master data set. If the master data set contains multiple rows with the same value of the BY variable, the first observation is updated and the remaining observations in the BY group are ignored. SAS writes a warning message to the log when the DATA step executes.
- The transaction data set can contain more than one observation with the same BY value. Multiple transaction rows are all applied to the master observation before it is written to the output file.
- Updating creates a new output data set. It does not update the existing master input data set the way that modifying does.
- Usually, the master data set and the transaction data set contain the same variables. However, to reduce processing time, you can create a transaction data set that contains only those variables that are being updated. The transaction data set can also contain new variables to be added to the output data set.
- The MERGE, UPDATE, and MODIFY statements can be used to combine and update rows in SAS data sets. For a comparison of these methods, see “Updating and Modifying Comparison” on page 416.
### Modifying

#### Definition

The MODIFY statement matches the values of one or more BY variables in a master data set against the same variables in a transaction data set. The existing master data set is modified. A new data set is not created.

#### Syntax

```plaintext
MODIFY master-data-set transaction-data-set <(data-set-options)>
BY by-variable;
```

#### Example: Modifying a Master Data Set from a Transaction Data Set

```
Master:
<table>
<thead>
<tr>
<th>Year</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2005</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2006</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2007</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2008</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2009</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2010</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2011</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2012</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2013</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2014</td>
<td>X1</td>
<td>Y1</td>
</tr>
</tbody>
</table>

Trans:
<table>
<thead>
<tr>
<th>Year</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>X2</td>
<td>Y2</td>
</tr>
<tr>
<td>2012</td>
<td>X2</td>
<td>Y2</td>
</tr>
<tr>
<td>2013</td>
<td>X2</td>
<td>Y2</td>
</tr>
<tr>
<td>2014</td>
<td>X1</td>
<td>Y1</td>
</tr>
<tr>
<td>2015</td>
<td>X2</td>
<td>Y2</td>
</tr>
</tbody>
</table>
```

In the figure, rows in the `Master` data set are modified based on the values of the specified BY variable `Year` in the `Trans` data set.

#### Details

- The MODIFY statement requires that the input data sets share a common variable that is specified in a **BY statement**.
- Modifying does not require that the input data sets are sorted or indexed.
- Both the master data set and the transaction data set can have rows with duplicate values of the BY variables.
- If duplicate values of the BY variable exist in the master data set, only the first occurrence is updated.
- If duplicate values of the BY variable exist in the transaction data set, the duplicates are applied one on top of another so that only the value in the last transaction appears in the master observation.
- Modifying does not permit you to create new variables or change variable attributes.
- The MERGE, UPDATE, and MODIFY statements can all be used to combine and update rows in SAS data sets. For a comparison of these methods, see “Updating and Modifying Comparison” on page 416.
Comparing Methods

PROC SQL versus Match-Merging

You can create and modify tables by using either the MERGE statement with the BY statement in the DATA step or by using the SQL procedure. For more information about DATA step match-merging versus PROC SQL, see "Creating and Updating Tables and Views" in SAS SQL Procedure User's Guide.

When values of the common variables are different between data sets that are being merged, they are often referred to as "unmatched rows" or "unmatched records." Missing values also result in unmatched rows.

The DATA step and PROC SQL treat unmatched rows differently.

**PROC SQL inner join:**

```sql
proc sql number;
  select *
  from inventory, Sales
  where inventory.PartNumber= Sales.PartNumber;
quit;
```

Includes only the matching rows in the output by default.

**DATA step match-merge:**

```sas
data merged;
  merge Inventory Sales; by PartNumber;
run;
```

Includes both the matching and the non-matching rows in the output by default.
The yellow highlights in the match-merge indicate the un-matched rows, which are not included in the SQL output. For another example that compares the DATA step with PROC SQL, see “Comparing PROC SQL with the SAS DATA Step” in SAS SQL Procedure User’s Guide.

### Updating and Modifying Comparison

**Table 21.2** Comparing Updating a Table with Modifying a Table

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Updating</th>
<th>Modifying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td><strong>UPDATE statement</strong>&lt;br&gt;data &lt;output-data-set&gt;;&lt;br&gt;UPDATE master-data-set;&lt;br&gt;trans-data-set&lt;br&gt;BY variable-name;</td>
<td><strong>Form 1: MODIFY statement</strong> (matching access)&lt;br&gt;data &lt;output-data-set&gt;;&lt;br&gt;MODIFY master-data-set;&lt;br&gt;trans-data-set&lt;br&gt;BY variable-name;</td>
</tr>
<tr>
<td>When to use</td>
<td>■ You have a master data set that needs to be changed or updated based on the values of a transaction data set.&lt;br&gt;■ You want to process most of the data in the master data set.</td>
<td>■ You want to change the master data set based on the values of another data set without creating an additional, new output data set. MODIFY creates an updated version of the original master data set.&lt;br&gt;■ You want to process or change only a small portion of the master data set.</td>
</tr>
<tr>
<td>Number of data sets that can be processed</td>
<td><strong>UPDATE and MODIFY</strong> when specified with the BY statement can combine rows from exactly two data sets.</td>
<td><strong>UPDATE and MODIFY</strong> when specified with the BY statement can combine rows from exactly two data sets.</td>
</tr>
</tbody>
</table>
### Comparing Methods

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Updating</th>
<th>Modifying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disk space</td>
<td>&quot;Updating&quot; requires more disk space because it produces an updated copy of the data set.</td>
<td>&quot;Modifying&quot; saves disk space because it updates the existing table without creating a new table.</td>
</tr>
<tr>
<td>BY statement requirements</td>
<td>The BY statement is required.</td>
<td>The BY statement is required when using the MODIFY statement to update a master data based on values in a separate, transaction data set when the input data sets are related in some way by one or more common variables. *</td>
</tr>
<tr>
<td>Unique BY or key value</td>
<td>Duplicates values for BY variables are allowed in the transaction data set. Multiple transaction rows are all applied to the master observation before it is written to the output file.</td>
<td>Duplicates values for BY variables are allowed in either the master data set or the transaction data set.</td>
</tr>
<tr>
<td>key value requirements</td>
<td>Unique values for BY variables are required in the master data set. If duplicate values for the BY variable exist in the master data set, then only the first observation with that value in the BY group is updated and SAS issues a warning.</td>
<td>If duplicates exist in the master data set, only the first occurrence is updated because the generated WHERE statement always finds the first occurrence in the master data set.</td>
</tr>
<tr>
<td>Sorting or index requirements</td>
<td>Sorting or indexing is required on input data sets.</td>
<td>Sorting or indexing is not required for any data set.</td>
</tr>
<tr>
<td></td>
<td>If an index exists on the input data set, then the UPDATE statement does not maintain it on the updated data set. You must rebuild the index.</td>
<td>Neither the master data set nor the transaction data set require sorting or indexing because the BY statement, when used with the MODIFY statement, triggers dynamic WHERE processing.</td>
</tr>
<tr>
<td>Treatment of missing values in</td>
<td>MODIFY and UPDATE do not overwrite values in the master data set with missing ones in the transaction data set by default.</td>
<td>If an index exists on the input data set, then the MODIFY statement maintains it on the updated data set.</td>
</tr>
<tr>
<td>the input data sets or master data set</td>
<td>To cause missing values in the transaction data set to replace existing values in the master data set, specify NOMISSINGCHECK in the UPDATEMODE= option. For an example, see Example: Update a Data Set with Missing Values on page 488</td>
<td></td>
</tr>
</tbody>
</table>

* For information on common variables, see the SAS documentation.
### Characteristic

<table>
<thead>
<tr>
<th>Updating</th>
<th>Modifying</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes. You can add new variables, delete variables, change variables, and perform any other change that affects the descriptor information of the data set.</td>
<td>No. You cannot add new variables, delete variables, change variables, or perform any change that affects the descriptor information of the data set.</td>
</tr>
</tbody>
</table>

#### Error-checking capabilities

- **Updating:** No. Updating does not need error checking because transactions without a corresponding master record are added to the data set by default.
- **Modifying:** Yes. Error-checking capabilities include using the automatic variable _IORC_ and the SYSRC autocall macro. For more information, see “Automatic Variable _IORC_ and the SYSRC Autocall Macro” in SAS DATA Step Statements: Reference.

#### Data set integrity

- **Updating:** No data loss occurs because the UPDATE statement works on a copy of the data.
- **Modifying:** Data might be only partially updated due to an abnormal task termination.

---

* The MODIFY statement, unlike the UPDATE statement, can be used in its other forms to make changes to a single input data set. In these cases, syntax Forms 2 through 4 and a BY statement are not required. However, for the purposes of this topic (combining data), the MODIFY statement requires the Form 1 version of its syntax, which does require the BY statement.

### One-to-One Reading versus One-to-One Merging

**Table 21.3** One-to-One Reading versus One-to-One Merging

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>One-to-One Reading</th>
<th>One-to-One Merging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>SET statement (DATA step)</td>
<td>MERGE statement (DATA step)</td>
</tr>
<tr>
<td>How rows are matched</td>
<td>Rows are matched by row number for both one-to-one reading and one-to-one merging. For example, row 1 of data set 1 is matched with row 1 of data set 2, row 2 of data set 1 is matched with row 2 of data set 2, and so on.</td>
<td></td>
</tr>
<tr>
<td>Characteristic</td>
<td>One-to-One Reading</td>
<td>One-to-One Merging</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td>Treatment of “unmatched rows”</td>
<td>Only matched rows are selected for output. That is, only rows that have the identical row number values in both data sets are included in the merged output. This means that merging data sets with different numbers of rows result in a merged data set that has the same number of rows as the smallest input data set.</td>
<td>Both matched and unmatched rows are selected for output. That is, all rows from all data sets are included in the output even when the input data sets have a different number of rows.</td>
</tr>
<tr>
<td>Treatment of common variables</td>
<td>Values of common variables are overwritten. That is, values of common variables in the last data set that is named in the SET or MERGE statement overwrite the values in the previous data sets. This is true for both one-to-one reading and one-to-one merging of data sets.</td>
<td></td>
</tr>
<tr>
<td>Sort requirements</td>
<td>None. There are no sort requirements for either of these methods.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 21.1 One-to-One Reading versus One-to-One Merging
Comparison of SAS Language Elements for Combining Data Sets

Table 21.4  Comparison of SAS Language Elements for Combining Data Sets

<table>
<thead>
<tr>
<th>Language Element</th>
<th>Purpose</th>
<th>Access Method</th>
<th>Use with BY statement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
<td>Direct</td>
</tr>
<tr>
<td><strong>BY</strong></td>
<td>The BY statement controls the operation of MERGE, MODIFY, SET, and UPDATE statements and creates and orders groups. The BY statement enables you to process rows that contain variables with equal values.</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td><strong>MERGE statement</strong></td>
<td>The MERGE statement reads rows from two or more data sets and joins them into a single row. When used with the BY statement, this type of merging is known as match-merging. When used without the BY statement, this type of merging is known as one-to-one merging. When the MERGE statement is used with the BY statement, data must be sorted or indexed based on the values of the BY variable.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Language Element</td>
<td>Purpose</td>
<td>Access Method</td>
<td>Use with BY statement</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>---------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
<td>Direct</td>
</tr>
<tr>
<td>MODIFY</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td><strong>Modifying</strong> processes rows in a SAS data set by updating the existing master data set rather than creating a new data set (as opposed to <strong>updating</strong>, which creates a new data set).</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sorting is not required but is recommended for performance.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td><strong>The SET statement</strong> reads any row from one or more SAS data sets.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td><strong>Interleaving</strong> of data set rows is achieved when a single SET statement is used with the BY statement and multiple input data sets are specified in the SET statement.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Concatenating</strong> (appending) of data sets is achieved in the same way that interleaving is achieved, except that no BY statement is used. A single SET statement is specified and multiple data sets are specified.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>One-to-one reading</strong> is achieved when multiple SET statements are used, one for each input data set, and no BY statement is used.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>The <strong>KEY= option and the POINT= option</strong> can be specified in the SET statement. They enable you to directly select specific rows for output.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Language Element</td>
<td>Purpose</td>
<td>Access Method</td>
<td>Use with BY statement</td>
</tr>
<tr>
<td>------------------</td>
<td>---------</td>
<td>---------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td><strong>UPDATE</strong></td>
<td></td>
<td>Sequential</td>
<td>Direct</td>
</tr>
<tr>
<td>“Updating” applies transactions to rows in a master data set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updating does not update rows by changing the existing master data set. It produces an updated copy of the existing master data set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Updating requires that the data in both the master and transaction data sets are indexed or sorted by the values of the BY variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>PROC APPEND</strong></td>
<td>PROC APPEND adds rows from one SAS data set to the end of another SAS data set.</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td><strong>PROC SQL</strong></td>
<td>PROC SQL reads any row from one or more SAS data sets.</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PROC SQL reads rows from up to 32 SAS data sets and joins them into single rows.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROC SQL manipulates rows in an existing SAS data set without creating a new data set.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROC SQL can produce a Cartesian product of the variables in the input data sets. The access method is chosen by the internal optimizer.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Hash Table</strong></td>
<td>Hash table merging combines data sets when values of the variable in the data sets match the value of the key in the hash table.</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Examples: Prepare Data

Example: Examine the Data to Be Combined

Example Code

In this example, the CONTENTS, SORT, and PRINT procedures are used to examine the data in three data sets before combining them.

The following table shows the three input data sets to be combined: Inventory, Sales, and Sales2019.

<table>
<thead>
<tr>
<th>Inventory</th>
<th>Sales</th>
<th>Sales2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part</td>
<td>Part</td>
<td>Part</td>
</tr>
<tr>
<td>Number</td>
<td>Name</td>
<td>Number</td>
</tr>
<tr>
<td>K89R</td>
<td>seal</td>
<td>NCF3</td>
</tr>
<tr>
<td>M4J7</td>
<td>sander</td>
<td>BV1E</td>
</tr>
<tr>
<td>LK43</td>
<td>filter</td>
<td>LK43</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>K89R</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>LK43</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>M4J7</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>BV1E</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td></td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td></td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td></td>
</tr>
</tbody>
</table>

1 View the descriptive information about the input data sets to determine whether they share any common variables. In the PROC CONTENTS output, ensure that the common variable, PartNumber, has the same attributes and represents the same data in each of the input data sets.
2 Sort the input data sets by the values of the common variable. Because the variable partNumber is common to all three input data sets, it can be used as a BY variable for merging the data.

3 Print the data sets to examine the values of the common variable and to determine the relationship between the data sets. The PROC PRINT output shows that there is a one-to-many relationship between the data sets Inventory and Sales, a many-to-one relationship between the data sets Sales and Sales2019, and a one-to-one relationship between the data sets Inventory and Sales2019.

**Figure 21.2** Partial PROC CONTENTS Output Comparing the Inventory, Sales, and Sales2019 Data Sets

**Figure 21.3** PROC PRINT Output for the Inventory, Sales, and Sales2019 Data Sets Showing One-to-Many and Many-to-One Relationships

---

**Note:** There are BY variable values that exist in the Inventory data set that do not exist in the Sales2019 data set. For example, in the Inventory data set, PartNumber JD03 does not exist in the Sales2019 data set. When the values of the BY variables are different between data sets that are being merged, they are often referred to as “unmatched observations” or “unmatched records.” Missing values are also considered unmatched observations. Different merge methods treat unmatched observations differently. See “Comparing Methods” on page 415 and “PROC SQL versus Match-Merging” on page 415 for more information.
Key Ideas

- The **PRINT procedure** lets you examine the structure and the contents of the data sets to be combined.
- The **CONTENTS procedure** displays descriptive information about data sets, including variable information, formats, number of observations, file size, and other summary information about the data.
- When merging data sets that have a one-to-many or a many-to-one relationship, you can use **PROC SQL**, a DATA step **MERGE BY (match-merge)** on page 458, the **MODIFY** or **UPDATE** statements, or a hash-table merge on page 480.
- When combining data sets using the DATA step **SET**, **MERGE**, **MODIFY**, and **UPDATE** statements, attributes of common variables must match. See, "Example: Prepare Data in Which Common Variables Have Different Data Types" on page 430, "Example: Prepare Data in Which Common Variables Have Different Lengths" on page 433, and "Example: Prepare Data in Which Common Variables Have Different Formats and Labels" on page 435 for more information.

See Also

- “COMPARE Procedure” in *Base SAS Procedures Guide*
- “Example: Find the Common Variables in Multiple Input Data Sets” on page 425

---

Example: Find the Common Variables in Multiple Input Data Sets

Example Code

In this example, PROC SQL uses the metadata in SAS session Dictionary tables to determine variables that are in common to three input data sets: **Inventory**, **Sales**, and **Sales2019**.

```sql
proc sql;
  title "Variables Common to Inventory, Sales, and Sales2019";
  create table commonvars as /*
  select memname, upcase(name) as name
    from dictionary.columns
    where libname='WORK' and
      memname in ('INVENTORY', 'SALES', 'SALES2019'); /*
  select name */
  from commonvars
  group by name
  having count(*)={select count(distinct(memname)) from commonvars};
quit;
```
1. Create the table, `commonvars`, that contains the variables from `work.inventory`, `work.sales`, and `work.sales2019`.

2. Identify the unique variable names from the `commonvars` table.

### Key Ideas

- You can also use the **COMPARE procedure** to compare up to two data sets.
- Identifying common variables across data sets is useful when you want to merge data based on matching values rather than concatenating or appending data. 
  PROC COMPARE and PROC SQL can be used to identify common variables between two data sets. PROC SQL can be used to identify common variables across multiple data sets.

### See Also

- "COMPARE Procedure"

### Example: Add a Message to the Log When Variables Are Missing from Input Data Sets

#### Example Code

In this example, the `OPEN=DEFER` option in the DATA statement causes a note to be written to the SAS log when variables in one or more of the input data sets are not present in the first input data set that is read.

The following table shows the input data sets that are used in this example. Notice how the variables `predict` and `lowermean`, which are defined in `Table2`, are not present in `Table1`.

When you concatenate the data sets without specifying the `OPEN=DEFER` option and you print the results, the output shows a typical concatenation in which the values for the variables that are not in all input data sets appear as missing:

```sas
data concat;
  set table1 table2;
run;
proc print data=concat;
  title "Concatenate Table1 and Table2";
run;
```
Compare this output to when you use the OPEN=DEFER option. Notice how only the variables from the data set that are listed in the first SET statement appear in the final output data set.

data concat2;
  set table1 table2 open=defer;
run;
proc print data=concat2;
  title "Concatenate with OPEN=DEFER";
run;

The following message appears in the log:

\textit{Output 21.1} Partial Log Output Showing Missing Variables When Using OPEN=DEFER

\begin{verbatim}
NOTE: There were 4 observations read from the data set WORK.TABLE1.
NOTE: For OPEN=DEFER processing, all variables processed should be specified by the first data set listed in the SET statement.
NOTE: Variable predict, found on WORK.TABLE2, is being ignored.
NOTE: Variable lowermean, found on WORK.TABLE2, is being ignored.
NOTE: There were 4 observations read from the data set WORK.TABLE2.
NOTE: The data set WORK.CONCAT2 has 8 observations and 5 variables.
\end{verbatim}
Key Ideas

- In most cases, if the set of variables defined by any subsequent data set differs from the variables defined by the first data set, SAS prints a warning message to the log but does not stop execution.

- When you concatenate data sets using the SET statement, SAS reads every row of all input data sets and creates a new output data set. When you use the APPEND procedure, SAS reads only the rows in the data set specified in the DATA= option and it does not create a new data set. See “Choosing between the SET Statement and the APPEND Statement” in Base SAS Procedures Guide for more information about choosing between the two methods.

- If the DATA= data set contains variables that are not in the BASE= data set, you must specify the FORCE option in the APPEND statement. See “Appending to Data Sets with Different Variables” in Base SAS Procedures Guide for more information.

- If no additional processing is necessary, using PROC APPEND or the APPEND statement in PROC DATASETS is more efficient than using a DATA step to concatenate data sets.

- Generally, you concatenate SAS data sets that have the same variables.

See Also

- “SET Statement” in SAS DATA Step Statements: Reference
- “Concatenating Two SAS Data Sets” in Base SAS Procedures Guide
- “APPEND Procedure” in Base SAS Procedures Guide
- FORCE option
- “Appending to Data Sets That Contain Variables with Different Attributes” in Base SAS Procedures Guide
- “Concatenating a CAS Table to a SAS Data Set” in Base SAS Procedures Guide

Example: Creating Unique BY Values When Data Contains Duplicate BY Values

Example Code

This example uses the FREQ procedure on the sales data set to check for duplicate values of the variable partNumber in the Sales data set. It then uses the _N_ DATA step automatic variable with the CATX function to create a unique row ID for the duplicate values.

```
proc sort data=sales; /* 1 */
   by partNumber;
run;
proc print data=sales; run;
```
proc freq data = Sales noprint;                  /* 2 */
  tables partNumber / out = SalesDupes
     (keep = partNumber Count
      where = (Count > 1));
run;
proc print data=SalesDupes;run;
data SalesUnique;                                /* 3 */
  set Sales;
  uniqueID = catx('.',partNumber,_n_);
run;
proc print data=SalesUnique;                     /* 4 */
  var uniqueID partNumber partName salesPerson;
run;

1 Sort the data by the variable, partNumber, which is used as the BY variable.
2 Use PROC FREQ to determine whether there are any duplicate values for partNumber. Create a data set, SalesDupes, which includes only the BY variable, partNumber, and Count. Count is a variable that is automatically generated by PROC FREQ and represents the frequency of each value of partNumber in the Sales data set. The value of partNumber is written to SalesDupes only if the value appears more than once in Sales (Count > 1).
3 Create a data set to contain the unique values of the BY variable partNumber. Modify the value of partNumber to create unique values by appending a unique number to each value. The CATX function appends the value of _N_ to each value of partNumber and stores the new value in uniqueID.
4 PROC PRINT shows the values of SalesUnique.

Output 21.2  Output of PROC FREQ That Shows Duplicate Values for the Common Variable, PartNumber, Found in the Sales Data Set

<table>
<thead>
<tr>
<th>Obs</th>
<th>partNumber</th>
<th>COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BV1E</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>LK43</td>
<td>2</td>
</tr>
</tbody>
</table>

Output 21.3  PROC PRINT Output Showing a New Variable Containing Unique Values for the Common Variable PartNumber

<table>
<thead>
<tr>
<th>Obs</th>
<th>uniqueID</th>
<th>partNumber</th>
<th>partName</th>
<th>salesPerson</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BV1E.1</td>
<td>BV1E</td>
<td>timer</td>
<td>JN</td>
</tr>
<tr>
<td>2</td>
<td>BV1E.2</td>
<td>BV1E</td>
<td>timer</td>
<td>KM</td>
</tr>
<tr>
<td>3</td>
<td>K8R.3</td>
<td>K8R</td>
<td>s seal</td>
<td>SJ</td>
</tr>
<tr>
<td>4</td>
<td>LK43.4</td>
<td>LK43</td>
<td>filter</td>
<td>KM</td>
</tr>
<tr>
<td>5</td>
<td>LK43.5</td>
<td>LK43</td>
<td>filter</td>
<td>JN</td>
</tr>
<tr>
<td>6</td>
<td>M4J.6</td>
<td>M4J</td>
<td>sander</td>
<td>KM</td>
</tr>
<tr>
<td>7</td>
<td>NCF3.7</td>
<td>NCF3</td>
<td>valve</td>
<td>JN</td>
</tr>
</tbody>
</table>
Example: Prepare Data in Which Common Variables Have Different Data Types

Example Code

In this example, the two data sets contain a common variable that has different data types.

This example uses the following input data sets: CarsSmall and a subset of the Sashelp.Cars data set.

data cars;                          /* 1 */
   set sashelp.cars;
run;

proc contents data=cars; run;     /* 2 */
proc contents data=CarsSmall; run;

data combineCars;
   merge cars CarsSmallNum;        /* 3 */
      by make model;
   keep Make DriveTrain Model MakeModelDrive Weight;
run;

1 Create some input data for the example from the Sashelp.Cars data set.
2 Display descriptive information about each of the input data sets to identify common variables. The CONTENTS procedure output shows that the common
variable, Weight, is a CHAR type variable in the CarsSmall data set and it is a NUMERIC type variable in the Cars data set.

Merge the data set by the values of the common variable, Weight. Because the type attribute is different in each of the input data sets, SAS prints an error message in the log stating that the variable type is incompatible.

Figure 21.4  Comparing PROC CONTENTS Output for the Data Sets Vehicles and VehiclesSmall Showing Different Data Types for the Common Variable

Example Code 21.1  Log Error Showing that the Type Attribute of the BY Variable is Different in the Input Data Sets

ERROR: Variable WeightLBS has been defined as both character and numeric. 1274  run;

To fix the problem, re-create the common variable, Weight, in the carsSmall data set by using the INPUT function in a new DATA step.

data CarsSmallNum;                   /* 1 */
  set CarsSmall;
  weightNum=input(weight, 8.);
  drop weight;
  rename WeightNum=weight;
run;

proc sort data=cars; by make model; run;    /* 2 */
proc sort data=CarsSmallNum; by Make Model; run;

data combineCars;
  merge cars CarsSmallNum;                 /* 3 */
    by make model;
  keep Make DriveTrain Model MakeModelDrive Weight;
run;
proc contents data=combineCars; run;        /* 4 */
proc print data=combineCars; run;

The INPUT function converts the variable, Weight, into a numeric data type and stores the value in a new variable WeightNum. The original Weight character
variable in CarsSmall is dropped, and the new numeric variable WeightNum is renamed to Weight. Renaming the variable ensures that the Cars and CarsSmall data sets have a common variable to merge the data sets.

2. The Cars and the CarsSmallNum data sets are sorted by the same variables so that they can be merged.

3. The Cars and the CarsSmallNum data sets are merged by make and model. Because the data type for the Weight variable in each data set matches, the data sets merge successfully to create the combineCars data set.

4. Only a portion of the CarsSmall data set is shown in Figure 21.5.

**Figure 21.5  Partial PROC PRINT Output for Match-Merged Data Sets after Normalizing the Common Variable's Type Attribute**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Make</th>
<th>Model</th>
<th>DriveTrain</th>
<th>Weight</th>
<th>MakeModelDrive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Acura</td>
<td>3.5 RL 4dr</td>
<td>Front</td>
<td>3869</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Acura</td>
<td>3.5 RL w/Navigation 4dr</td>
<td>Front</td>
<td>3853</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Acura</td>
<td>MDX</td>
<td>All</td>
<td>4451</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Acura</td>
<td>NSX coupe 2dr manual 5</td>
<td>Rear</td>
<td>3153</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Acura</td>
<td>RSX Type S 2dr</td>
<td>Front</td>
<td>2778</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Acura</td>
<td>TL 4dr</td>
<td>Front</td>
<td>3575</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>Chevrolet</td>
<td>Venture LS</td>
<td>Front</td>
<td>3699</td>
<td></td>
</tr>
<tr>
<td>91</td>
<td>Chevrolet</td>
<td>Aveo 4dr</td>
<td>Front</td>
<td>2370</td>
<td>Chevrolet Aveo 4dr - Front wheel drive</td>
</tr>
<tr>
<td>92</td>
<td>Chevrolet</td>
<td>Aveo LS 4dr hatch</td>
<td>Front</td>
<td>2348</td>
<td>Chevrolet Aveo LS 4dr hatch - Front wheel drive</td>
</tr>
<tr>
<td>93</td>
<td>Chrysler</td>
<td>360M 4dr</td>
<td>Front</td>
<td>3581</td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>Chrysler</td>
<td>360M Special Edition 4dr</td>
<td>Front</td>
<td>3550</td>
<td></td>
</tr>
</tbody>
</table>

**Key Ideas**

- When combining data sets, variables in the input data sets that have the same name are referred to as common variables. The common variable in one input data set does not always share the same attributes as the same variable in another data set.
- Common variables with the same data but different attributes can cause problems when the data sets are combined.
- If the type attribute is different, SAS stops processing the DATA step and issues an error message stating that the variables are incompatible. To correct this error, you must use a DATA step to re-create the variables.
- If the length attribute is different, SAS takes the length from the first data set that is specified in the SET, MERGE, or UPDATE statement.

**See Also**

- "Converting Character Values to Numeric Values" in *SAS Functions and CALL Routines: Reference*
Example: Prepare Data in Which Common Variables Have Different Lengths

Example Code

In this example, the data sets `quarter1`, `quarter2`, `quarter3`, and `quarter4` are match-merged into one output data set using the MERGE statement with the BY statement. The data sets are pre-sorted and merged on the common variable `account`.

This example uses the `Quarter1`, `Quarter 2`, `Quarter3`, `Quarter4` data sets.

```sas
proc print data=quarter1; run;          /* 1 */
proc print data=quarter2; run;
proc print data=quarter3; run;
proc print data=quarter4; run;

data yearly;
   merge quarter1 quarter2 quarter3 quarter4;  /* 2 */
   by Account;
run;

data yearly;                                   /* 3 */
   length Mileage 6;
   merge quarter1 quarter2 quarter3 quarter4;
   by Account;
run;
proc contents data=yearly; run;                /* 4 */
```

1. View the descriptive information about the input data sets to compare whether the attributes of the common variables are the same in each of the input data sets. The `PROC CONTENTS output` shows that the length of the common variable, `mileage`, is four bytes in the `quarter1` data set, eight bytes in the `quarter2` data set, and six bytes in the `quarter3` and `quarter4` data sets.

2. Merge the data sets by the common variable, `mileage`. Notice that SAS issues a nonzero return code and prints a warning in the log output.

3. Change the length of the `mileage` variable by specifying the appropriate length in the `LENGTH` statement before specifying the `MERGE` statement. The `LENGTH` statement must also come before the `SET`, `MERGE`, and `UPDATE` statements if they are used.

4. View the descriptive information for the merged output data set.
Example Code 21.2  Log Output for Match-Merge of Data Sets Containing Variables with Different Lengths

WARNING: Multiple lengths were specified for the variable mileage by input data set(s). This can cause truncation of data.

You can also use the ATTRIB statement to change the length attribute on the variable:

```sas
data yearly;
  merge quarter1 quarter2 quarter3 quarter4;
  by Account;
  attrib Mileage
    length = 6;
run;
```

If you expect truncation of data (for example, when removing insignificant blanks from the end of character values), the warning is expected and you do not want SAS to issue a nonzero return code. In this case, you can turn this warning off by setting the VARLENCHK= system option to NOWARN.
Key Ideas

- When combining data sets, variables in the input data sets that have the same name are referred to as common variables. The common variable in one input data set does not always share the same attributes as the same variable in another data set.
- Common variables with the same data but different attributes can cause problems when the data sets are combined.
- When combining data sets, if the length of the common variable is different, SAS takes the length from the first data set that is specified in the SET, MERGE, or UPDATE statement, and then prints a warning message on page 434 in the SAS log.

See Also

- “LENGTH Statement” in SAS DATA Step Statements: Reference

Example: Prepare Data in Which Common Variables Have Different Formats and Labels

Example Code

In this example, the data sets class and classfit are match-merged using the MERGE statement with the BY variable, Name.

The ATTRIB statement is used to control the length, label, and format of the common variable in the final output data set.

This example uses the Class and Classfit data sets.

Before merging the data sets, PROC CONTENTS is run on each data set to identify the common variable and to look for any differences between the attributes of the common variables.

```
proc contents data=class; run;               /* 1 */
proc contents data=classfit; run;            /* 1 */

proc sort data=class; by name; run;                /* 2 */
proc sort data=classfit; by name; run;            /* 2 */

data merged;
merge class classfit; by Name;
attrib Weight
   label = "Weight";
attrib Height Weight Predict format=comma8.2;   /* 3 */
run;
proc print data=merged;                         /* 4 */
run;
```
proc contents data=merged; run;

1 View the descriptive information about the input data sets to determine whether they share any common variables and to compare their attributes. The PROC CONTENTS output shows that the common variables Height and Weight have different labels in each of the input data sets.

2 Sort the input data sets by the values of the BY variable.

3 Change the attributes by specifying the ATTRIB statement:

4 Print the descriptor information for the merged output data set.

**Output 21.4** PROC CONTENTS Output for the Data Sets Class and Classfit Showing the Differences in Formats and Labels

<table>
<thead>
<tr>
<th>#</th>
<th>Variable</th>
<th>Type</th>
<th>Len</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Age</td>
<td>Num</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Height</td>
<td>Num</td>
<td>8</td>
<td>COMMA8.</td>
</tr>
<tr>
<td>1</td>
<td>Name</td>
<td>Char</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Weight</td>
<td>Num</td>
<td>8</td>
<td>COMMA8.</td>
</tr>
</tbody>
</table>

**Output 21.5** PROC CONTENTS Output for the Default Behavior When Formats and Labels Are Different in the Merged Data Sets

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Age</th>
<th>Height</th>
<th>Weight in Pounds</th>
<th>Predict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alice</td>
<td>13</td>
<td>57</td>
<td>84</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Barbara</td>
<td>13</td>
<td>65</td>
<td>98</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Carol</td>
<td>14</td>
<td>63</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Jane</td>
<td>12</td>
<td>60</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Janet</td>
<td>15</td>
<td>63</td>
<td>113</td>
<td>100.7</td>
</tr>
<tr>
<td>6</td>
<td>Joyce</td>
<td>11</td>
<td>51</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Judy</td>
<td>14</td>
<td>64</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Louise</td>
<td>12</td>
<td>56</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Mary</td>
<td>15</td>
<td>67</td>
<td>112</td>
<td>116.3</td>
</tr>
<tr>
<td>10</td>
<td>Philip</td>
<td>16</td>
<td>72</td>
<td>112</td>
<td>137.7</td>
</tr>
<tr>
<td>11</td>
<td>Ronald</td>
<td>15</td>
<td>67</td>
<td>133</td>
<td>118.2</td>
</tr>
<tr>
<td>12</td>
<td>William</td>
<td>15</td>
<td>67</td>
<td>150</td>
<td>116.3</td>
</tr>
</tbody>
</table>
**Key Ideas**

- If the label, format, or informat attributes of the common variable in the input data sets are different, SAS takes the attribute from the first data set that is listed in the `SET`, `MERGE`, `MODIFY`, or `UPDATE` statement.
- Any label, format, or informat that you explicitly specify overrides the default. If all data sets contain explicitly specified attributes, the one specified in the first data set listed in the `SET` or `MERGE` statement overrides the others.
- You can ensure that the new output data set has the attributes that you want by using the `ATTRIB` statement in the `DATA` step.
- You can also use `VLABEL` and `VLABELX` functions to modify attributes on variables.

**See Also**

- “`ATTRIB` Statement” in *SAS DATA Step Statements: Reference*
- “`VLABEL` Function” in *SAS Functions and CALL Routines: Reference*
- “`VLABELX` Function” in *SAS Functions and CALL Routines: Reference*

---

**Example: Prepare Data in Which Common Variables Represent Different Data**

**Example Code**

In this example, the data sets to be merged contain variables that have the same name but they represent completely different data. In this sense, they are not meant to be common variables.
To resolve this problem, rename one of the variables in one of the input data sets. Here, the `RENAME=` data set option is used to rename the variable `Weight` in the `CarsSmall` input data set:

```sas
data vehicles(rename=(weight=weightLBS));
  set vehicles;
run;
```

**Key Ideas**

- If the label, format, or informat attributes of the common variable in the input data sets are different, SAS takes the attribute from the first data set that is listed in the `SET`, `MERGE`, `MODIFY`, or `UPDATE` statement.
- Any label, format, or informat that you explicitly specify overrides the default. If all data sets contain explicitly specified attributes, the one specified in the first data set listed in the `SET` or `MERGE` statement overrides the others.
- You can ensure that the new output data set has the attributes that you want by using the `ATTRIB` statement in the `DATA` step.
- You can also use the “VLABEL Function” in SAS Functions and CALL Routines: Reference and the “VLABELX Function” in SAS Functions and CALL Routines: Reference to modify attributes on variables.

**See Also**

- “RENAME Statement” in SAS DATA Step Statements: Reference

---

**Examples: Concatenate Data**

**Example: Concatenate Using the SET Statement**

**Example Code**

In this example, the `SET` statement is used to concatenate two data sets into a single output data set. The input data sets share one common variable (`common`). Concatenation does not require that the data sets share any variables or that the data is related in anyway since it simply involves the placing of one intact data set after the other.

The following table shows the input data sets that are used in this example: `animal` and `plant`. 
The SET statement reads all of the observations sequentially from both input data sets, and it then appends the observations from the first data set that is read to the second data set that is read. The results are stored in a new output data set named `concatenate`. As a result, observations from the `animal` data set are displayed first in the output, followed by the observations from the `plant` data set.

Print the results (Output 21.7 on page 439).

**Output 21.7**  PROC PRINT Output Showing Concatenated Data Sets Using the SET Statement

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>a</td>
<td></td>
<td>Apple</td>
</tr>
<tr>
<td>8</td>
<td>b</td>
<td></td>
<td>Banana</td>
</tr>
<tr>
<td>9</td>
<td>c</td>
<td></td>
<td>Coconut</td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td></td>
<td>Dewberry</td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td></td>
<td>Eggplant</td>
</tr>
<tr>
<td>12</td>
<td>f</td>
<td></td>
<td>Fig</td>
</tr>
</tbody>
</table>

Notice that the number of observations in the output data set is 12, which is the sum of the observations from both input data sets.
Key Ideas

- Concatenation is the combining of two or more data sets, one after the other, into a single data set.
- In the output data set, observations from the data set that is listed first in the SET statement are followed by observations from the data set that is listed second in the SET statement, and so on.
- The output data set contains all of the variables from both input data sets. Values of variables that are found in one data set but not in another are set to missing.
- Generally, you concatenate SAS data sets that have the same variables.

See Also

- Concatenating on page 407 data sets
- “SET Statement” in SAS DATA Step Statements: Reference

Example: Concatenate Using PROC SQL

Example Code

In this example, the SQL procedure is used to concatenate two data sets together into a new output SAS data set and an SQL table. During the concatenation, SQL reads all the rows from both input data sets and creates a new output data set named combined.

The following table shows the input data sets that are used in this example: animal and plant:

<table>
<thead>
<tr>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS common animal</td>
<td>OBS common plant</td>
</tr>
<tr>
<td>1 a Ant</td>
<td>1 a Apple</td>
</tr>
<tr>
<td>2 b Bird</td>
<td>2 b Banana</td>
</tr>
<tr>
<td>3 c Cat</td>
<td>3 c Coconut</td>
</tr>
<tr>
<td>4 d Dog</td>
<td>4 d Dewberry</td>
</tr>
<tr>
<td>5 e Eagle</td>
<td>5 e Eggplant</td>
</tr>
<tr>
<td>6 f Frog</td>
<td>6 f Fig</td>
</tr>
</tbody>
</table>

```sql
proc sql;
  create table combined as /* 1 */
  select * from animal    /* 2 */
  outer union corresponding /* 3 */
  select * from plant;    /* 4 */
quit;
```

1. Use the Create statement to create the table, combined.
Include all variables from animal.

Use an OUTER UNION CORRESPONDING statement to include all rows from both tables and to overlay the common variables.

Include all variables from plant.

**Output 21.8**  Concatenated animal and plant Data Sets Using PROC SQL

<table>
<thead>
<tr>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ant</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Bird</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Cat</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>Eagle</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>Frog</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>Banana</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>Coconut</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Dewberry</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>Eggplant</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>Fig</td>
<td></td>
</tr>
</tbody>
</table>

The resulting output consists of a SAS data set and an SQL table. Both tables have 12 rows (observations) each. The total number of rows in the output is equal to the sum of the rows from the combined data sets. Values of variables that are found in one data set but not in another are set to missing.

In the output data set, the observations from the plant data set are appended to the observations from the animal data set. Because the animal data set is first in the SET statement, it appears first in the output. The output data set contains all the variables from both input data sets.

**Key Ideas**

- Concatenation is the combining of two or more data sets, one after the other, into a single data set.
- Generally, you concatenate SAS data sets that have the same variables.

**See Also**

- “Concatenating Query Results (OUTER UNION)” in SAS SQL Procedure User’s Guide
- “Union Joins” in SAS SQL Procedure User’s Guide
- “OUTER UNION” in SAS SQL Procedure User’s Guide
- “Comparing PROC SQL with the SAS DATA Step” in SAS SQL Procedure User’s Guide
Example: Append Files Using PROC APPEND

Example Code

In this example, the APPEND procedure is used to add observations from the Year2 data set to the end of the Year1 data set. During the processing, the procedure reads only the rows from the Year2 data set. The procedure then updates the Year1 data set by appending the observations from the Year2 data set to it. The procedure does not generate a new output data set. It simply updates the existing Year1 data set.

The following table shows the input data sets that are used in this example: Year1 and Year2:

**Table 21.5  Input Data to be Combined**

<table>
<thead>
<tr>
<th>Year1</th>
<th>Year2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>Date</td>
</tr>
<tr>
<td>1</td>
<td>2009</td>
</tr>
<tr>
<td>2</td>
<td>2010</td>
</tr>
<tr>
<td>3</td>
<td>2011</td>
</tr>
<tr>
<td>4</td>
<td>2012</td>
</tr>
<tr>
<td>5</td>
<td>2014</td>
</tr>
</tbody>
</table>

```
proc append base=Year1 data=Year2;
run;
proc print data=Year1;
  title "Year1";
run;
```

**Output 21.9  Concatenated Tables (PROC APPEND)**

The Year1 data set contains all the observations from both data sets.

**Note:** You cannot use PROC APPEND to add observations to a SAS data set in a sequential library.
Key Ideas

- When you **concatenate data sets using the SET statement**, SAS reads every row of all input data sets and creates a new output data set. When you use the APPEND procedure, SAS reads only the rows in the data set specified in the DATA= option and it does not create a new data set. See “Choosing between the SET Statement and the APPEND Statement” in *Base SAS Procedures Guide* for more information about choosing between the two methods.

- If the DATA= data set contains variables that are not in the BASE= data set, you must specify the FORCE option in the APPEND statement. See “Appending to Data Sets with Different Variables” in *Base SAS Procedures Guide* for more information.

- If no additional processing is necessary, using PROC APPEND or the APPEND statement in PROC DATASETS is more efficient than using a DATA step to concatenate data sets.

- Generally, you concatenate SAS data sets that share one or more common variables.

See Also

- “Concatenating Two SAS Data Sets” in *Base SAS Procedures Guide*
- “APPEND Procedure” in *Base SAS Procedures Guide*
- “APPEND Statement” in *Base SAS Procedures Guide*
- “Appending to Data Sets That Contain Variables with Different Attributes” in *Base SAS Procedures Guide*
- “Concatenating a CAS Table to a SAS Data Set” in *Base SAS Procedures Guide*

Example: Add a Message to the Log When Variables Are Missing from Input Data Sets

Example Code

In this example, the **OPEN=DEFER** option in the DATA statement causes a note to be written to the SAS log when variables in one or more of the input data sets are not present in the first input data set that is read.

The following table shows the input data sets that are used in this example. Notice how the variables *predict* and *lowermean*, which are defined in Table2, are not present in Table1.

When you concatenate the data sets without specifying the OPEN=DEFER option and you print the results, the output shows a typical concatenation in which the values for the variables that are not in all input data sets appear as missing:

```plaintext
   data concat;
```
set table1 table2;
run;
proc print data=concat;
   title "Concatenate Table1 and Table2";
run;

### Concatenate Table1 and Table2

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
<th>predict</th>
<th>lowermean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>Judy</td>
<td>F</td>
<td>14</td>
<td>64.3</td>
<td>90.0</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
<td>126.006</td>
<td>116.942</td>
</tr>
<tr>
<td>6</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
<td>101.832</td>
<td>96.375</td>
</tr>
<tr>
<td>7</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
<td>104.562</td>
<td>98.982</td>
</tr>
<tr>
<td>8</td>
<td>Judy</td>
<td>F</td>
<td>14</td>
<td>64.3</td>
<td>90.0</td>
<td>107.681</td>
<td>101.842</td>
</tr>
</tbody>
</table>

Compare this output to when you use the OPEN=DEFER option. Notice how only the variables from the data set that are listed in the first SET statement appear in the final output data set.

data concat2;
   set table1 table2 open=defer;
run;
proc print data=concat2;
   title "Concatenate with OPEN=DEFER";
run;

### Concatenate with OPEN=DEFER

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
</tr>
<tr>
<td>2</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
</tr>
<tr>
<td>3</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
</tr>
<tr>
<td>4</td>
<td>Judy</td>
<td>F</td>
<td>14</td>
<td>64.3</td>
<td>90.0</td>
</tr>
<tr>
<td>5</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
</tr>
<tr>
<td>6</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
</tr>
<tr>
<td>7</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
</tr>
<tr>
<td>8</td>
<td>Judy</td>
<td>F</td>
<td>14</td>
<td>64.3</td>
<td>90.0</td>
</tr>
</tbody>
</table>

The following message appears in the log:
Output 21.10  Partial Log Output Showing Missing Variables When Using OPEN=DEFER

NOTE: There were 4 observations read from the data set WORK.TABLE1.
NOTE: For OPEN=DEFER processing, all variables processed should be specified by the first data
    set listed in the SET statement.
NOTE: Variable predict, found on WORK.TABLE2, is being ignored.
NOTE: Variable lowermean, found on WORK.TABLE2, is being ignored.
NOTE: There were 4 observations read from the data set WORK.TABLE2.
NOTE: The data set WORK.CONCAT2 has 8 observations and 5 variables.

Key Ideas

- In most cases, if the set of variables defined by any subsequent data set differs from the variables defined by the first data set, SAS prints a warning message to the log but does not stop execution.

- When you concatenate data sets using the SET statement, SAS reads every row of all input data sets and creates a new output data set. When you use the APPEND procedure, SAS reads only the rows in the data set specified in the DATA= option and it does not create a new data set. See “Choosing between the SET Statement and the APPEND Statement” in Base SAS Procedures Guide for more information about choosing between the two methods.

- If the DATA= data set contains variables that are not in the BASE= data set, you must specify the FORCE option in the APPEND statement. See “Appending to Data Sets with Different Variables” in Base SAS Procedures Guide for more information.

- If no additional processing is necessary, using PROC APPEND or the APPEND statement in PROC DATASETS is more efficient than using a DATA step to concatenate data sets.

- Generally, you concatenate SAS data sets that have the same variables.

See Also

- “SET Statement” in SAS DATA Step Statements: Reference
- “Concatenating Two SAS Data Sets” in Base SAS Procedures Guide
- “APPEND Procedure” in Base SAS Procedures Guide
- FORCE option
- “Appending to Data Sets That Contain Variables with Different Attributes” in Base SAS Procedures Guide
- “Concatenating a CAS Table to a SAS Data Set” in Base SAS Procedures Guide
Examples: Interleave Data

Example: Interleave Data Sets

Example Code

The following program creates the input data sets, sorts each of the data sets by their common variable, interleaves the data sets, and then prints the results. The common variable, common, is specified as the BY variable.

The following input data sets, animal and plant are used in this example:

<table>
<thead>
<tr>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS 1</td>
<td>a</td>
</tr>
<tr>
<td>OBS 2</td>
<td>b</td>
</tr>
<tr>
<td>OBS 3</td>
<td>c</td>
</tr>
<tr>
<td>OBS 4</td>
<td>d</td>
</tr>
<tr>
<td>OBS 5</td>
<td>e</td>
</tr>
<tr>
<td>OBS 6</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBS 1</th>
<th>common</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
</tr>
</tbody>
</table>

The following program first sorts both input data sets using the SORT procedure, interleaves the data sets, and then prints the results.

```
proc sort data=animal; by common; run;
proc sort data=plant; by common; run;

data interleave;
  set animal plant;
  by common;
run;
proc print data=interleave; run;
```

Notice that the input data sets are sorted by the same variable that is specified in the DATA step BY statement.

The output data set contains all the variables from all data sets, as well as variables created by the DATA step. Values of variables that are found in one data set but not in another are set to missing. The number of observations in the output data set is 12, which is the sum of the observations from both data sets.
Key Ideas

- Input data sets must first be indexed or sorted by the values of the BY variables.
- The observations in the interleaved data sets are not combined; they are copied from the original data sets in the order of the values of the BY variable.
- Input data sets are processed sequentially, in the order in which they are listed in the SET statement.
- Observations in the output data set are arranged by the values of the BY variable.

See Also

- “Interleaving” on page 408 data sets
- “SET Statement” in SAS DATA Step Statements: Reference
- “BY Statement” in SAS DATA Step Statements: Reference
- “Combining SAS Data Sets” in SAS DATA Step Statements: Reference

Example: Interleave with Duplicate Values of the BY Variable

Example Code

The following program creates the input data sets, sorts each of the data sets by its common variable, interleaves the data sets, and then prints the results.
The program first sorts the input data sets using the SORT procedure, which groups and sorts the rows in each data set by the values of the common variable, common. Notice that there are duplicate values for the shared BY variable in both input data sets.

The following table shows the input data sets animalDupes and plantDupes, with the duplicate values highlighted in each data set:

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ape</td>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Bird</td>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
<td>4</td>
<td>c</td>
<td>Celery</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>Dog</td>
<td>5</td>
<td>d</td>
<td>Dewberry</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Eagle</td>
<td>6</td>
<td>e</td>
<td>Eggplant</td>
</tr>
</tbody>
</table>

`proc sort data=animalDupes; by common; run;`  
`proc sort data=plantDupes; by common; run;`  
`data interleave;`  
`set animalDupes plantDupes;`  
`by common;`  
`run;`  
`proc print data=interleave; run;`

The output data set contains all the variables from both input data sets. Values of variables that are found in one data set but not in another are set to missing. The number of observations in the output data set is 12, which is the sum of the observations from the input data sets. The observations are written to the output data set in the order in which they occur in the original data sets.
Output 21.12  PROC PRINT Output for Interleaved Data Sets with Duplicate Values of the BY Variable

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ape</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>Bird</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>Banana</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>Cat</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>Coconut</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>Celery</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>d</td>
<td>Dewberry</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>Eagle</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>e</td>
<td>Eggplant</td>
<td></td>
</tr>
</tbody>
</table>

Notice that observations from the input data set `animalDupes` are listed first, followed by observations from the second input data set, `plantDupes`. This is because the DATA step processes data sets sequentially, in the order in which they are listed in the SET statement. For example, if you change the order of the input data sets so that the data set `plantDupes` is listed first in the SET statement, the observations from `plantDupes` would be listed first in the output data set.

data interleave;
  set plantDupes animalDupes; by common;
run;
proc print data=interleave; run;
Key Ideas

- Input data sets must first be indexed or sorted by the values of the BY variables.
- The observations in the interleaved data sets are not combined; they are copied from the original data sets in the order of the values of the BY variable.
- Input data sets are processed sequentially, in the order in which they are listed in the SET statement.
- Observations in the output data set are arranged by the values of the BY variable.

See Also

- “Interleaving” on page 408 data sets
- “SET Statement” in SAS DATA Step Statements: Reference
- “BY Statement” in SAS DATA Step Statements: Reference
- “Combining SAS Data Sets” in SAS DATA Step Statements: Reference
Example: Interleave with Different Values of the Common BY Variable

Example Code

The following program creates the input data sets, sorts each of the data sets by the common variable, interleaves the data sets, and then prints the results.

The following table shows the input data sets used in this example: animalDupes and plantMissing2 with the different BY values highlighted:

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>Obs</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ape</td>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Bird</td>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
<td>4</td>
<td>e</td>
<td>Eggplant</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>Dog</td>
<td>5</td>
<td>f</td>
<td>Fig</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Eagle</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both input data sets contain values for the variable common that are not present in the other data set. For example, the value "d" in the animalDupes data set is not present in the plantMissing2 data set. The value "f" in the plantMissing2 data set is not present in the animalDupes data set.

1 Create the input data sets, animalDupes and plantMissing2. Each input data set contains the variable common, and the SORT procedure sorts observations in order of the values of the BY variable, common.

2 The DATA step interleaves the data sets based on the values of common.

The output data set contains all the variables from both input data sets. Values of variables that are found in one data set but not in another are set to missing. The number of observations in the output data set is 11, which is the sum of the observations from the input data sets. The observations are written to the output data set in the order in which they occur in the original data sets.
Key Ideas

- Input data sets must first be indexed or sorted by the values of the BY variables.
- The observations in the interleaved data sets are not combined; they are copied from the original data sets in the order of the values of the BY variable.
- Input data sets are processed sequentially, in the order in which they are listed in the SET statement.
- Observations in the output data set are arranged by the values of the BY variable.

See Also

- “Interleaving” on page 408 data sets
- “SET Statement” in SAS DATA Step Statements: Reference
- “BY Statement” in SAS DATA Step Statements: Reference
- “Combining SAS Data Sets” in SAS DATA Step Statements: Reference
Examples: Match-Merge Data

Example: Match-Merge Observations Based on a BY Variable

Example Code

In this example, the MERGE statement is used with the BY statement to merge two data sets. The observations from the first data set are merged with the observations from the second data set based on the values of a common BY variable.

The input data sets animal and plant both contain the variable common, which is specified as the BY variable in this example.

This example shows a one-to-one merge because there are no duplicate values for the BY variable in either of the input data sets.

The following table shows the input data sets that are used in this example: animal and plant:

<table>
<thead>
<tr>
<th>Table 21.6 Input Data Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>animal</strong></td>
</tr>
<tr>
<td>OBS</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

The following program first sorts both input data sets using the SORT procedure, and then match-merges the data sets by the common BY variable, common. PROC PRINT is used to print the results:

```plaintext
proc sort data=animal; by common; run;
proc sort data=plant; by common; run;

data matchmerge;
    merge animal plant;
    by common;
run;
proc print data=matchmerge; run;
```
Output 21.15  PROC PRINT Output for Match-Merge Observations Based on a BY Variable

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Fig</td>
</tr>
</tbody>
</table>

Key Ideas

- Match-merging is used for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.
- Input data sets must be indexed or sorted on the BY variable prior to merging.
- A match-merge in SAS is comparable to an inner join in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in SAS SQL Procedure User’s Guide for more information.
- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.

See Also

- “Match-Merging” on page 409 data sets
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403
Example: Match-Merge Observations with Common Variables That Are Not the BY Variables

Example Code

In this example, the MERGE statement is used with the BY statement to merge two data sets in a one-to-many merge. The observations from the first data set are merged with the observations from the second data set based on the values of a common BY variable. Because there are duplicate values for the BY variable in one of the input data sets, this is a one-to-many merge.

This example shows what happens to the values of a common variable when it is not specified as the BY variable.

The input data sets that are used in this example, one and many, both contain the variables ID and state. The variable ID is the BY variable and its values are unique within the one data set. However, its values are not unique within the many data set. There are multiple observations for values of ID in the many data set. The variable state is common to both input data sets, but it is not specified as a BY variable.

The purpose for the merge is to replace the incorrect abbreviations for state in the many data set with the correct, two-letter abbreviations shown in the data set one.

The following table shows the one and many data sets that are used in this example:

<table>
<thead>
<tr>
<th>one</th>
<th>many</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
</tr>
<tr>
<td></td>
<td>state</td>
</tr>
<tr>
<td>ID</td>
<td>city</td>
</tr>
<tr>
<td></td>
<td>state</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
When these data sets are merged, the value of the common variable in data set `one` overwrites the values from data set `many` because the `one` data set is listed second in the MERGE statement. However, on subsequent iterations of the MERGE statement for the same BY group, the `one` data set is not read again. Therefore, the values from the `one` data set do not replace the remaining values in the BY group. This means that the first value in each BY group is replaced but the remaining values are not.

```plaintext
data three;
  merge many one;
  by ID;
run;
proc print data=three noobs; run;
title;
```

**Output 21.16** PROC PRINT Output Showing Input Data Sets That Contain a Common BY Variable and a Common Non-BY Variable

**Output 21.17** PROC PRINT Output for Match Merging Observations with a Common Variable That is Not the BY Variable

Only the first row of each BY group is updated
To replace the values for `state` for all observations within the BY group, the common (non-BY) variable from the `many` data set must be dropped or renamed:

```sas
data solution;
  merge many(drop=state) one;
  by ID;
  run;
proc print data=solution noobs; run;
```

**Output 21.18**  PROC PRINT Output for Solution to One-to-many Merge with Common Variables That Are Not the BY Variables

<table>
<thead>
<tr>
<th>ID</th>
<th>city</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Phoenix</td>
<td>AZ</td>
</tr>
<tr>
<td>2</td>
<td>Boston</td>
<td>MA</td>
</tr>
<tr>
<td>2</td>
<td>Foxboro</td>
<td>MA</td>
</tr>
<tr>
<td>3</td>
<td>Olympia</td>
<td>WA</td>
</tr>
<tr>
<td>3</td>
<td>Seattle</td>
<td>WA</td>
</tr>
<tr>
<td>3</td>
<td>Spokana</td>
<td>WA</td>
</tr>
<tr>
<td>4</td>
<td>Madison</td>
<td>WI</td>
</tr>
<tr>
<td>4</td>
<td>Milwaukee</td>
<td>WI</td>
</tr>
<tr>
<td>4</td>
<td>Madison</td>
<td>WI</td>
</tr>
<tr>
<td>4</td>
<td>Hurley</td>
<td>WI</td>
</tr>
</tbody>
</table>

SAS reads the value from data set `one` on the first iteration of the merge. Because variables that are read from data sets are automatically retained throughout a BY group, all observations for the BY group, `state`, contain the values from data set `one`.

**Key Ideas**

- For the first matching observation in a one-to-many match-merge, the value of the common variable in the last data set specified in the MERGE statement overwrites the values from the previous data sets. However, on subsequent iterations of the MERGE statement for the same BY group, the first data set is not read again. So, the remaining values of the common BY variable come from the originating data set rather than from the last data set read.

- Match-merging is used for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.

- Input data sets must be indexed or sorted on the BY variable prior to merging.

- A match-merge in SAS is comparable to an *inner join* in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in *SAS SQL Procedure User’s Guide* for more information.

- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
Example: Match-Merge Observations with Duplicate Values of the BY Variable

Example Code

In this example, the MERGE statement is used with the BY statement to merge two data sets. The observations from the first data set are merged with the observations from the second data set based on the values of a common BY variable. The input data sets contain duplicate values of the BY variable (common), so this is an example of a many-to-one merge.

The following table shows the input data sets that are used in this example: animalDupes and plantDupes:

<table>
<thead>
<tr>
<th>animalDupes</th>
<th>plantDupes</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>common</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
</tr>
</tbody>
</table>

The following program first sorts both input data sets using the SORT procedure, and then match-merges the data sets by the common variable, common. PROC PRINT is used to print the results:

```sas
proc sort data=animalDupes; by common; run;
proc sort data=plantDupes; by common; run;

data matchmerge;
merge animalDupes plantDupes;
by common;
run;
proc print data=matchmerge; run;
```
**Output 21.19  Match-Merge Observations with Duplicate Values of the BY Variable**

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ape</td>
<td>Apple</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>Cat</td>
<td>Celery</td>
</tr>
<tr>
<td>6</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
</tbody>
</table>

**Key Ideas**

- Match-merging is used for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.
- Input data sets must be indexed or sorted on the BY variable prior to merging.
- A match-merge in SAS is comparable to an *inner join* in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in *SAS SQL Procedure User’s Guide* for more information.
- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.
- When there are unequal members of observations in a data set, the common variables get their values from the data set contributing those values. Any unique variables to data sets retain their values until the end of the BY group.

**See Also**

- “Match-Merging” on page 409 data sets
- “MERGE Statement” in *SAS DATA Step Statements: Reference*
- “Data Relationships” on page 403
Example: Match-Merge Observations with Different Values of the BY Variable

Example Code

In this example, the MERGE statement is used with the BY statement to perform a match-merge on two input data sets. The observations from one data set are merged together with the observations from another data set by a common variable.

The two input data sets have different values for their common variables resulting in unmatched observations. Unmatched observations means that an observation in one input data set does not contain the same value for the BY variable in the other input data set.

The following table shows the input data sets that are used in this example: animalMissing and plantMissing2:

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>Cat</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>Dog</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>Eagle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>Eggplant</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>Fig</td>
</tr>
</tbody>
</table>

In the input data sets, data set animalMissing does not contain a value of "b" or "f" for the common variable, but data set plantMissing2 does. Data set plantMissing2 does not contain the value "d" for the common variable, but data set animalMissing does.

The following program first sorts both input data sets using the SORT procedure, and then match-merges the data sets by the common variable, common. PROC PRINT is used to print the results:

```sas
proc sort data=animalMissing; by common; run;
proc sort data=plantMissing2; by common; run;

data matchmerge;
    merge animalMissing plantMissing2;
    by common;
run;
proc print data=matchmerge; run;
```

Notice how SAS retains the values of all variables from both input data sets in the final output, even if the value is missing in one data set.
Output 21.20  PROC PRINT Output for Match-Merge with Unmatched Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Banana</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Fig</td>
<td></td>
</tr>
</tbody>
</table>

Key Ideas

- Match-merging is used for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.
- Input data sets must be indexed or sorted on the BY variable prior to merging.
- A match-merge in SAS is comparable to an inner join in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in SAS SQL Procedure User’s Guide for more information.
- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.

See Also

- “Match-Merging” on page 409 data sets
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403
Example: Match-Merge and Remove Unmatched Observations

Example Code

In this example, the MERGE statement is used with the BY statement to merge two data sets. The observations from the first data set are merged with the observations from the second data set based on the values of a common variable.

The following example uses the MERGE and BY statements to combine two data sets that have unmatched observations. This example is identical to “Example: Match-Merge Observations with Different Values of the BY Variable” on page 460 except that in this example, the IN= data set option is used to remove the unmatched observations from the output data set. Unmatched observations refers to observations in which the values for the shared BY variable are not equal in both input data sets.

The following table shows the input data sets that are used in this example: animalMissing and plantMissing2:

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>Cat</td>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>Dog</td>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>Eagle</td>
<td>4</td>
<td>e</td>
<td>Eggplant</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
<td>Fig</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the input data sets, data set animalMissing does not contain the value b or f for the common variable, but data set plant does. Data set plantMissing2 does not contain the value d for the common variable, but data set animal does.

The data sets animalMissing and plantMissing2 do not contain all values of the BY variable common.

The following program first sorts both input data sets using the SORT procedure, and then match-merges the data sets by the common variable, common. PROC PRINT is used to print the results:

```
proc sort data=animalMissing; by common; run;
proc sort data=plantMissing2; by common; run;

data matchmerge;
  merge animalMissing plantMissing2;
  by common;
run;
proc print data=matchmerge; run;
```
In the next example, the program match-merges the two data sets and uses the **IN=** data set option on the input data sets to remove the unmatched observations from the output data set.

The **IN=** data set option is a Boolean value variable, which has a value of 1 if the data set contributes to the current observation in the output and a value of 0 if the data set does not contribute to the current observation in the output.

```sas
data matchmerge2;
  merge animalMissing(in=i) plantMissing2(in=j);
  by common;
  if (i=1) and (j=1);
run;
proc print data=matchmerge2; run;
```

**Output 21.22  PROC PRINT Output for Match-Merge Using the IN= Data Set Option to Remove Unmatched Observations**

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Banana</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td></td>
<td>Fig</td>
</tr>
</tbody>
</table>
Key Ideas

- Match-merging is used for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.
- Input data sets must be indexed or sorted on the BY variable prior to merging.
- A match-merge in SAS is comparable to an inner join in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in SAS SQL Procedure User’s Guide for more information.
- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.

See Also

- “Match-Merging” on page 409 data sets
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403

Example: Match-Merge Data Sets with Duplicate Values in More Than One Data Set

Example Code

In this example, the MERGE statement is used with the BY statement to perform a match-merge on two input data sets. The observations from one data set are merged with observations from another data set based on a common BY variable.

The data set fruit is merged with the data set color based on the values of the common variable, ID.
Notice that the data set fruit has duplicate values, a and c for ID. The data set color has duplicate values, b and c for ID.

Note: In this example, it is assumed that the data sets are pre-sorted or indexed on the BY variable, ID.

```
proc print data=fruit; title 'Fruit'; run;
proc print data=fruit; title 'Color'; run;

data merged;
  merge fruit color;
  by id;
run;

proc print data=merged;
  title 'Merged by ID';
run;
```
Output 21.23  PROC PRINT Output for Merging Observations by a Common Variable When Duplicates Exist in More Than One Input Data Set

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td>id</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
</tr>
<tr>
<td>9</td>
<td>c</td>
</tr>
</tbody>
</table>

Notice the different values for the variable fruit for the BY group c. The value cocoa is overwritten in the PDV with cream when the second observation in the BY group is read from the data set color. The value cream carries down the rest of the BY group. The value is carried down because when a BY statement is used with the MERGE statement, variables do not reinitialize to missing until the BY group changes.

Key Ideas

- Match-merging is intended for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.
- Input data sets must be indexed or sorted on the BY variable prior to merging.
- A match-merge in SAS is comparable to an inner join in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in SAS SQL Procedure User’s Guide for more information.
- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).
- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.

See Also

- “Match-Merging” on page 409 data sets
Example: Match-Merge One-to-Many with Missing Values in Common Variables

Example Code

In this example, the MERGE statement is used with the BY statement to perform a match-merge on two input data sets when one data set contains missing data. The example shows how missing values are treated the same as nonmissing values.

The following table shows the input data sets that are used in this example. Note that data set one has missing values for age after the first value for each ID.

<table>
<thead>
<tr>
<th></th>
<th>one</th>
<th></th>
<th>two</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>ID</td>
<td>age</td>
<td>score</td>
<td>OBS</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>90</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>.</td>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>.</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>9</td>
<td>80</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>.</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Doing a simple merge of these two data sets by ID results in missing values for the variable age in the merged output data set:

```sas
proc print data=one; title 'One'; run;
proc print data=two; title 'Two'; run;

data merge1;
    merge one two;
    by id;
run;
proc print data=merge1; title 'Merged by ID'; run;
```
After the first observation from each data set is combined, SAS reads data set one for the next observation for the BY group. It overwrites the values in the PDV with those values, including the missing value for age. It does not read from data set two again since the BY group is complete. Therefore, the next observation contains a missing value for age.

To get the values for age in data set two to replace the missing values, you can use the IF statement to check for missing values. If the value for age is missing, a temporary variable, temp_age, is created to contain the last nonmissing value for the BY group. The value for temp_age is then used as the value for age if age is missing.

```
data merge2 (drop=temp_age);
  merge one two;
  by id;
  retain temp_age;
  if first.id then temp_age = .;
  if age = . then age = temp_age;
  else temp_age = age;
run;
proc print; title 'Merged by ID with Age Retained'; run;
```
Key Ideas

- Match-merging is intended for merging data sets that have one or more common variables and you want to merge the data sets based on the values of the common variables.

- Input data sets must be indexed or sorted on the BY variable prior to merging.

- A match-merge in SAS is comparable to an *inner join* in PROC SQL when all of the values of the BY variable match and there are no duplicate BY variables. See “Inner Joins” in *SAS SQL Procedure User’s Guide* for more information.

- SAS retains the values of all variables in the program data vector even if the value is missing (or unmatched).

- When SAS reads the last observation from a BY group in one data set, SAS retains its values in the program data vector for all variables that are unique to that data set until all observations for that BY group have been read from all data sets. The total number of observations in the final data set is the sum of the maximum number of observations in a BY group from either data set.

See Also

- “Match-Merging” on page 409 data sets
- “MERGE Statement” in *SAS DATA Step Statements: Reference*
- “Data Relationships” on page 403
Examples: Merge Data One-to-One

Example: Merge Data Sets with an Equal Number of Observations

Example Code

In this example, the MERGE statement is used without a BY statement to perform a one-to-one merge of two data sets that have an equal number of rows. The input data sets animal and plantG contain a common variable, named common. The values for the shared variable are equal except in row 6, where the value is f in the animal data set and g in the plantG data set.

The following table shows the input data sets that are used in this example: animal and plantG:

<table>
<thead>
<tr>
<th>animal</th>
<th>plantG</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>common</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
</tr>
</tbody>
</table>

The following program merges these data sets and prints the results:

```plaintext
data merged;
   merge animal plantG;
run;

proc print data=merged; run;
```
Output 21.24  PROC PRINT Output for One-to-One Merge of Data Sets with an Equal Number of Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
<td>Frog</td>
<td>Fig</td>
</tr>
</tbody>
</table>

The output data set contains all the variables from both input data sets. Notice that the value for the variable `common` in observation 6 in the output data set is g. The values for the common variable, `common`, in the data set `plantG` replace the values for `common` in the `animal` data set.

Key Ideas

- The MERGE statement recognizes common variables in the input data sets, but, without a BY statement, it does not merge observations based on variable values. Rather, it matches observations implicitly based on row number, regardless of the variable values.
- If the column names in the input data sets are the same and no BY statement is specified, then the merge overwrites the values of the common columns. The values of the common variables in the data set specified last in the MERGE statement overwrite the values in the previously specified data sets. Column names (variables) that are not shared by all the input data sets are added as new columns.
- The DATA step reads the first observation from the first data set and then reads the first observation from the second data set, and so on.
- The resulting output data set contains all the observations from all the input data sets, regardless of whether they have the same number of observations.
- You can use the MERGENOBY system option to control log messaging when performing a one-to-one merge.

See Also

- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403
Example: Merge Data Sets with an Unequal Number of Observations

Example Code

In this example, the MERGE statement is used without a BY statement to perform a one-to-one merge of two data sets that have an unequal number of rows.

The following table shows the input data sets that are used in this example: animal and plantMissing:

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The data sets animal and plantMissing both contain the variable common, and the observations are arranged by the values of common. The plantMissing data set has fewer observations than the animal data set.

The following program merges these unequal data sets and prints the results:

```sas
data animal;
input common $ animal $;
datalines;
a Ant
b Bird
c Cat
d Dog
e Eagle
f Frog
;

data plantMissing;
input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
;

data merged;
merge animal plantMissing;
run;
proc print data=merged; run;
```
PROC PRINT Output for Merging Data Sets with an Unequal Number of Observations

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td></td>
</tr>
</tbody>
</table>

Compare the program above to the one-to-one reading of the same data sets using the SET statement:

```plaintext
data combine;
  set animal;
  set plantMissing;
run;
proc print data=combine; run;
```

PROC PRINT Output for Combining Data Sets with an Unequal Number of Observations Using the SET Statement

<table>
<thead>
<tr>
<th>Obs</th>
<th>Common</th>
<th>Animal</th>
<th>Plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
</tbody>
</table>

The DATA step stops selecting observations for output after it reads the last observation in the data set with the least number of observations. Therefore, the number of observations in the resulting output data set is the number of observations in the smallest original data set. In this example, the DATA step stops selecting observations when it reads the last observation in plantMissing.
Key Ideas

- The MERGE statement recognizes common variables in the input data sets, but, without a BY statement, it does not merge observations based on variable values. Rather, it matches observations implicitly based on row number, regardless of the variable values.

- If the column names in the input data sets are the same and no BY statement is specified, then the merge overwrites the values of the common columns. The values of the common variables in the data set specified last in the MERGE statement overwrite the values in the previously specified data sets. Column names (variables) that are not shared by all the input data sets are added as new columns.

- The DATA step reads the first observation from the first data set and then reads the first observation from the second data set, and so on.

- The resulting output data set contains all the observations from all the input data sets, regardless of whether they have the same number of observations.

- You can use the MERGENOBY system option to control log messaging when performing a one-to-one merge.

See Also

- “MERGE Statement” in SAS DATA Step Statements: Reference
- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403

Example: Merge Data Sets with Duplicate Values of Common Variables

Example Code

The following example shows how you can get undesirable results when you merge data sets that contain duplicate values of common variables. In this case, the merging is done without specifying a BY variable. This type of merging should be reserved for data sets that have a one-to-one relationship. The input data sets in this example do not have a one-to-one relationship.

The following table shows the input data sets that are used in this example: animalDupes and plantDupes:
The data sets animalDupes and plantDupes contain the variable common, and each data set contains observations with duplicate values of common.

The following program merges the data sets and prints the results.

```sas
/* This program illustrates undesirable results. */
data animalDupes;
input common $ animal $;
datalines;
a  Ant
a  Ape
b  Bird
c  Cat
d  Dog
e  Eagle
;

data plantDupes;
input common $ plant $;
datalines;
a  Apple
b  Banana
c  Coconut
c  Celery
d  Dewberry
e  Eggplant
;

data merged;
    merge animalDupes plantDupes;
run;
proc print data=merged; run;
```

---

### animalDupes

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>Ape</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>Bird</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>Dog</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Eagle</td>
</tr>
</tbody>
</table>

### plantDupes

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Celery</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>Dewberry</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Eggplant</td>
</tr>
</tbody>
</table>
**Output 21.27**  PROC PRINT Output for Undesirable Results When Merging Data Sets That Have Duplicate Values of Common Variables

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Ape</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Bird</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>Cat</td>
<td>Celery</td>
</tr>
<tr>
<td>5</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>6</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
</tbody>
</table>

This method works as expected on data that has a one-to-one relationship. Since the relationship of the data in this example has both a one-to-many and many-to-one relationship, you might get unwanted results.

**Key Ideas**

- The MERGE statement recognizes common variables in the input data sets, but, without a BY statement, it does not merge observations based on variable values. Rather, it matches observations implicitly based on row number, regardless of the variable values.
- If the column names in the input data sets are the same and no BY statement is specified, then the merge overwrites the values of the common columns. The values of the common variables in the data set specified last in the MERGE statement overwrite the values in the previously specified data sets. Column names (variables) that are not shared by all the input data sets are added as new columns.
- The DATA step reads the first observation from the first data set and then reads the first observation from the second data set, and so on.
- The resulting output data set contains all the observations from all the input data sets, regardless of whether they have the same number of observations.
- You can use the MERGENOBY system option to control log messaging when performing a one-to-one merge.

**See Also**

- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403
Example: Merge Data Sets with Different Values for the Common Variables

Example Code

The following example shows the undesirable results obtained from using the one-to-one merge to combine data sets that have different values for their common variable.

In this example, the data sets `animalMissing` and `plantMissing2` have different values for the variable `common`.

The following table shows the input data sets that are used in this example: `animalMissing` and `plantMissing2`:

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>1</td>
<td>a</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>Cat</td>
<td>2</td>
<td>b</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
<td>Dog</td>
<td>3</td>
<td>c</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>Eagle</td>
<td>4</td>
<td>e</td>
<td>Eggplant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>f</td>
<td>Fig</td>
</tr>
</tbody>
</table>

The following program produces the data set `merged` and prints the results:

```sql
data merged;
merge animalMissing plantMissing2;
run;
proc print data=merged; run;
```

Output 21.28  PROC PRINT Output Showing Undesirable Results with a One-to-One Merge of Data Sets with Different Values for the Common Variable
Key Ideas

- The MERGE statement recognizes common variables in the input data sets, but, without a BY statement, it does not merge observations based on variable values. Rather, it matches observations implicitly based on row number, regardless of the variable values.

- If the column names in the input data sets are the same and no BY statement is specified, then the merge overwrites the values of the common columns. The values of the common variables in the data set specified last in the MERGE statement overwrite the values in the previously specified data sets. Column names (variables) that are not shared by all the input data sets are added as new columns.

- The DATA step reads the first observation from the first data set and then reads the first observation from the second data set, and so on.

- The resulting output data set contains all the observations from all the input data sets, regardless of whether they have the same number of observations.

- You can use the MERGENOBY system option to control log messaging when performing a one-to-one merge.

See Also

- “MERGE Statement” in SAS DATA Step Statements: Reference
- “Data Relationships” on page 403

Examples: Combine Data One-to-One

Example: Combine Data Sets That Contain an Equal Number of Observations

Example Code

In this example, two SET statements are used to combine observations from one data set together with the observations from another data set. The input data sets animal and plantG contain a common variable, named common. The values for the shared variable are equal except in row 6, where the value is f in the animal data set and g in the plantG data set.

The following table shows the input data sets that are used in this example: animal and plantG:
The following program combines the data sets and prints the results:

```plaintext
data combine;
set animal;
set plantG;
run;

proc print data=combine; run;
```

**Output 21.29**  PROC PRINT Output for Combine Data Sets That Contain an Equal Number of Observations Using the SET Statement

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Fig</td>
</tr>
</tbody>
</table>

Because the SET statement does not merge observations by matching the values of a common variable, using this method on data sets that have different values for the like-named variables can cause undesired results.

In this example, the animal data set has a value of f for the like-named variable, common, in row 6. The plantG data set has a value of g for common in row 6. The data set plantG is specified in the last SET statement, so the values for common in the plantG data set overwrite the values for common in the animal data set.
Key Ideas

- The values of common variables from the data set specified last in the SET statement replace the values of the common variables from previous data sets.
- The DATA step reads the first observation from the first data set and then reads the first observation from the second data set, and so on.
- The resulting output data set contains all the variables from all the input data sets.
- The DATA step stops selecting observations for output after it reads the last observation in the data set with the least number of observations. Therefore, the number of observations in the resulting output data set is the number of observations in the smallest original data set.
- To use the SET statement to combine data sets that have an unequal number of observations, you can use the POINT= option to directly access and match the observations by a common variable.

See Also

- KEY= data set option
- “Combining One Observation with Many” in SAS DATA Step Statements: Reference
- “Performing a Table Lookup When the Master File Contains Duplicate Observations” in SAS DATA Step Statements: Reference

Example: Merge Data Using a Hash Table

Example: Use a Hash Table to Merge Data Sets One-to-Many or Many-to-Many

Example Code

In this example, a hash table is used to merge two sets of data that have a common variable. The hash table is created from a SAS data set that is loaded into memory and is available for use by the DATA step that created it. The common variable in the hash table is unique and is used as a key that provides very fast lookup into the internal memory table.

The second data set is used as the base data set. The DATA step reads observations from the base data set and uses the common variable to find a match in the hash table. Matching observations are written out to the output data set named in the DATA statement.
The following tables show the input data sets **product_list** and **supplier**, with the common variable **Supplier_ID** highlighted in each of the data sets:

**product_list**

<table>
<thead>
<tr>
<th>Product_Id</th>
<th>Product_Name</th>
<th>Supplier_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>240200100101</td>
<td>Grandslam Staff Tour Mhl Golf Gloves</td>
<td>3808</td>
</tr>
<tr>
<td>210200100017</td>
<td>Sweatshirt Children's  O-Neck</td>
<td>3298</td>
</tr>
<tr>
<td>240400200022</td>
<td>Aftm 95 Vf Long Kg-6s White</td>
<td>1280</td>
</tr>
<tr>
<td>230100100017</td>
<td>Men's Jacket Rem</td>
<td>50</td>
</tr>
<tr>
<td>210200300006</td>
<td>Fleece Cuff Pant Kid'S</td>
<td>1303</td>
</tr>
<tr>
<td>210200500002</td>
<td>Children's Mittten</td>
<td>772</td>
</tr>
<tr>
<td>210200700016</td>
<td>Strap Pants BBQ</td>
<td>798</td>
</tr>
<tr>
<td>210201000050</td>
<td>Kid Children's T-Shirt</td>
<td>2963</td>
</tr>
<tr>
<td>21020200009</td>
<td>Kids Sweat Round Neck,Large Logo</td>
<td>3298</td>
</tr>
<tr>
<td>21020100067</td>
<td>Logo Coord.Children's Sweatshirt</td>
<td>2963</td>
</tr>
<tr>
<td>220100100019</td>
<td>Fit Racing Cap</td>
<td>1303</td>
</tr>
<tr>
<td>220100100025</td>
<td>Knit Hat</td>
<td>1303</td>
</tr>
<tr>
<td>22010300001</td>
<td>Fleece Jacket Compass</td>
<td>772</td>
</tr>
<tr>
<td>220200200036</td>
<td>Soft Astro Men's Running Shoes</td>
<td>1747</td>
</tr>
<tr>
<td>230100100015</td>
<td>Men's Jacket Caians</td>
<td>50</td>
</tr>
<tr>
<td>23010500004</td>
<td>Backpack Flag, 6,5x9 Cm.</td>
<td>316</td>
</tr>
<tr>
<td>210200500006</td>
<td>Rain Suit, Plain w/backpack Jacket</td>
<td>772</td>
</tr>
<tr>
<td>23010500006</td>
<td>Collapsible Water Can</td>
<td>316</td>
</tr>
<tr>
<td>224040020000</td>
<td>Bat 5.5 Fly</td>
<td>3808</td>
</tr>
<tr>
<td>220200200035</td>
<td>Soft Alta Plus Women's Indoor Shoes</td>
<td>1747</td>
</tr>
<tr>
<td>240400200066</td>
<td>Memhis  350,Yellow Medium, 6-pack</td>
<td>1280</td>
</tr>
<tr>
<td>240200100081</td>
<td>Extreme Distance 90 3-pack</td>
<td>3808</td>
</tr>
</tbody>
</table>

**supplier**

<table>
<thead>
<tr>
<th>Supplier_ID</th>
<th>Supplier_Name</th>
<th>Supplier_Address</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>Scandinavian Clothing A/S</td>
<td>Kr. Augusts Gate 13 NO</td>
<td></td>
</tr>
<tr>
<td>316</td>
<td>Prime Sports Ltd</td>
<td>9 Carlisle Place GB</td>
<td></td>
</tr>
<tr>
<td>755</td>
<td>Top Sports</td>
<td>Jernbanegade 45 DK</td>
<td></td>
</tr>
<tr>
<td>772</td>
<td>AllSeasons Outdoor Clothing</td>
<td>553 Cliffview Dr US</td>
<td></td>
</tr>
<tr>
<td>798</td>
<td>Sportico</td>
<td>C. Barquillo 1 ES</td>
<td></td>
</tr>
<tr>
<td>1280</td>
<td>British Sports Ltd</td>
<td>85 Station Street GB</td>
<td></td>
</tr>
<tr>
<td>1303</td>
<td>Eclipse Inc</td>
<td>1218 Carriole Ct US</td>
<td></td>
</tr>
<tr>
<td>1684</td>
<td>Magnifico Sports</td>
<td>Rua Costa Pinto 2 PT</td>
<td></td>
</tr>
<tr>
<td>1747</td>
<td>Pro Sportwear Inc</td>
<td>2434 Edggbrook Dr US</td>
<td></td>
</tr>
<tr>
<td>3298</td>
<td>A Team Sports</td>
<td>2687 Julie Ann Ct US</td>
<td></td>
</tr>
<tr>
<td>3808</td>
<td>Carolina Sports</td>
<td>3860 Grand Ave US</td>
<td></td>
</tr>
</tbody>
</table>

In this program, the **product_list** data set is used as the base data set. The **DECLARE** statement names the in-memory location of the supplier data set. The **DEFINEKEY** method identifies the unique key variable that is used to join the data sets. The **DEFINEDATA** method lists additional variables that we want loaded into memory.

**Note:** The base data set contains duplicates of the key. The observations are read and processed one at a time, and the duplicate values do not affect processing.
data supplier_info;
    drop rc;
    length Supplier_Name $40 Supplier_Address $ 45 Country $ 2; /* 1 */
    if _N_=1 then do;
        declare hash S(dataset:'work.supplier'); /* 2 */
        S.definekey('Supplier_ID');
        S.definedata('Supplier_Name', 'Supplier_Address', 'Country');
        S.definedone();
        call missing(Supplier_Name, Supplier_Address, Country); /* 3 */
    end;
    set work.product_list; /* 4 */
    rc=S.find(); /* 5 */
    run;
    proc print data=supplier_info;
    var Product_ID Supplier_ID Supplier_Name Supplier_Address Country;
    title "Product Information";
    run;
    title;

1 Include a LENGTH statement to ensure that the Supplier_Name, Supplier_Address, and Country are defined in the PDV.

2 In the first iteration of the DATA step, declare the hash object, S. Assign Supplier_ID as the hash object key. Include the values of Supplier_Name, Supplier_Address, and Country from the work.supplier data set.

3 Because Supplier_Name, Supplier_Address, and Country are not explicitly assigned initial values, SAS writes a NOTE to the log that the variables are not initialized. CALL MISSING suppresses the NOTE that is written to the log.

4 Read an observation from the product_list data set.

5 The FIND method is called to see if the Supplier_ID from product_list matches the Supplier_ID key for one of the records in the hash object. If there is a match (rc=0), the observation is written to the supplier_info data set.
Key Ideas

- A SAS hash table contains rows (hash entries) and columns (hash variables).
- Each hash entry must have at least one key column and one data column. Values can be hardcoded or loaded from a SAS data set.
- A hash table resides completely in memory, making its operations fast. The data does not need to be pre-sorted.
- A hash table is temporary: once the DATA step has stopped execution, it ceases to exist. Thus, it cannot be reused in any subsequent step. However, its content can be saved in a SAS data set or external database.
- The hash object is sized dynamically.

See Also

- “OUTPUT Method” in SAS Component Objects: Reference
- Fundamentals of The SAS® Hash Object
Examples: Update Data

Example: Update Data Using the UPDATE Statement

Example Code

In this example, the UPDATE statement is used to update a master data set based on new values in a transaction data set. The data set, `master`, contains the original values of the shared variables `common` and `plant`. The transaction data set, `plantNew`, contains the new values for the shared variable `plant`. The goal is to update the master data set with the new values for plant. Specifically, the value `Eggplant` for plant in row 5 should be replaced with the value `Escarole` from the `plantNew` data set.

The following table shows the input data set, `master`, and the transaction data set, `plantNew`.

<table>
<thead>
<tr>
<th>OBS</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
<th>OBS</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
<td>1</td>
<td>a</td>
<td>Apricot</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
<td>2</td>
<td>b</td>
<td>Barley</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
<td>3</td>
<td>c</td>
<td>Cactus</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
<td>4</td>
<td>d</td>
<td>Date</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
<td>5</td>
<td>e</td>
<td>Escarole</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Fig</td>
<td>6</td>
<td>f</td>
<td>Fennel</td>
</tr>
</tbody>
</table>

The program first creates the two input data sets, then updates the master data set based on the values of the BY variable, `common`. The PRINT procedure prints the results:

```sas
data master2;
  update master plantNew;
  by common;
run;
proc print data=master2; run;
```
Output 21.30  PROC PRINT Output for Using the UPDATE Statement to Update Data

<table>
<thead>
<tr>
<th>Obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apricot</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Barley</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Cactus</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Date</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Escarole</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Fennel</td>
</tr>
</tbody>
</table>

Key Ideas

- The UPDATE statement enables you to update values in one data set (the master data set) based on the values in another data set (transaction data set). The UPDATE statement creates a new output data set.
- The BY statement is required and the input data sets must contain an index or be sorted by the values of the BY variables.
- Because the UPDATE statement creates a new file when it generates output, you can add, delete, or rename variables when you perform an update.
- By default, missing values in the transaction data set do not replace existing values in the master data set. You can change this behavior so that missing values in the transaction data set replace values in the master data set by specifying NOMISSINGCHECK in the UPDATEMODE= option.
- If the transaction data set contains duplicate values of the BY variable, then the values from the transaction data set replace the values in the master data set.
- If an observation in the transaction data set does not have a corresponding observation in the master data set, then SAS adds an observation to the master output data set. Observations are matched based on the value of the BY variable.
- If no changes need to be made to an observation in the master data set, then that observation does not need to be included in the transaction data set.

See Also

- “Updating” on page 413
- Table 21.2 on page 416
- “UPDATE Statement” in SAS DATA Step Statements: Reference
- “SORT Procedure” in Base SAS Procedures Guide
- UPDATEMODE= in SAS DATA Step Statements: Reference
- “Error Checking When Using Indexes to Randomly Access or Update Data” on page 538
Example: Update Data Sets with Duplicate Values of the BY Variable

Example Code

In this example, the UPDATE statement is used to update a master data set based on new values in a transaction data set. The transaction data set contains duplicate values of the common variable in observations 4 and 5. The data sets also share the common variable plant. This example shows what happens to common variables that are not the BY variable when there are duplicate values for the BY variable.

The following table shows the master input data set, master, and the transaction data set, plantNewDupes:

<table>
<thead>
<tr>
<th>obs</th>
<th>common</th>
<th>animal</th>
<th>plant</th>
<th>obs</th>
<th>common</th>
<th>plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>Ant</td>
<td>Apple</td>
<td>1</td>
<td>a</td>
<td>Apricot</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>Bird</td>
<td>Banana</td>
<td>2</td>
<td>b</td>
<td>Barley</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>Cat</td>
<td>Coconut</td>
<td>3</td>
<td>c</td>
<td>Cactus</td>
</tr>
<tr>
<td>4</td>
<td>d</td>
<td>Dog</td>
<td>Dewberry</td>
<td>4</td>
<td>d</td>
<td>Date</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
<td>Eagle</td>
<td>Eggplant</td>
<td>5</td>
<td>d</td>
<td>Dill</td>
</tr>
<tr>
<td>6</td>
<td>f</td>
<td>Frog</td>
<td>Fig</td>
<td>6</td>
<td>e</td>
<td>Escarole</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>f</td>
<td>Fennel</td>
</tr>
</tbody>
</table>

The following program first creates the two input data sets, then updates the master data set based on the values of the BY variable, common. Because the transaction data set contains duplicate values for the BY variable, common. Because it was not specified as a BY variable, the values for the other common variable, plant, are replaced by the values in the transaction data set. The value Dewberry in the master data set is replaced by Dill, which is the last value for plant in the transaction data set. The PRINT procedure prints the results:

```plaintext
data master;
  update master plantNewDupes;
  by common;
run;
proc print data=master; run;
```

CAUTION

Values of the BY variable must be unique for each observation in the master data set. If the master data set contains duplicate values for the BY variable, then only the first observation containing the variable is updated and subsequent observations containing the variable are ignored. SAS writes a warning message to the log when the DATA step executes.
Key Ideas

- The UPDATE statement enables you to update values in one data set (the master data set) based on the values in another data set (transaction data set). The UPDATE statement creates a new output data set.
- The BY statement is required and the input data sets must contain an index or be sorted by the values of the BY variables.
- Because the UPDATE statement creates a new file when it generates output, you can add, delete, or rename variables when you perform an update.
- By default, missing values in the transaction data set do not replace existing values in the master data set. You can change this behavior so that missing values in the transaction data set replace values in the master data set by specifying NOMISSINGCHECK in the UPDatemode= option.
- If the transaction data set contains duplicate values of the BY variable, then the values from the transaction data set replace the values in the master data set.
- If an observation in the transaction data set does not have a corresponding observation in the master data set, then SAS adds an observation to the master output data set. Observations are matched based on the value of the BY variable.
- If no changes need to be made to an observation in the master data set, then that observation does not need to be included in the transaction data set.

See Also

- “Updating” on page 413
- Table 21.2 on page 416
- “UPDATE Statement” in SAS DATA Step Statements: Reference
- “SORT Procedure” in Base SAS Procedures Guide
- UPDatemode= in SAS DATA Step Statements: Reference
- “Error Checking When Using Indexes to Randomly Access or Update Data” on page 538
Example: Update a Data Set with Missing and Different Values for the BY Variables

Example Code

In this example, the UPDATE statement is used with the BY statement to update a master data set based on new values in a transaction data set.

The master data set, master, contains a missing value for the variable plant in the first observation. Not all of the values of the common BY variable (common) are included.

The transaction data set, minerals, contains a new variable (mineral), a new value for the BY variable, common, and missing values for several observations.

The following table shows the input data sets, master and minerals:

<table>
<thead>
<tr>
<th>master</th>
<th>minerals</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>common</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
</tr>
<tr>
<td>3</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
</tr>
<tr>
<td>5</td>
<td>f</td>
</tr>
<tr>
<td>6</td>
<td>g</td>
</tr>
</tbody>
</table>

The following program updates the master data set based on the values in the minerals data set and prints the results:

```plaintext
data master;
  update master minerals;
  by common;
run;

proc print data=master; run;
```
Note the following points about the updated master data set:

- The variable `mineral` was added to the master output data set and is set to missing for some observations.
- Values in observations 2 and 6 in the transaction data set do not have corresponding values in the master data set. They are added to the master data set as new observations.
- The value for `plant` in observation 4 is not changed to missing even though it is missing in the transaction data set.
- Three observations in the new data set have updated values for the variable `plant`.

If you want the values that are missing in the transaction data set to be updated as missing in the master output data set, then specify the `UPDATEMODE=` option in the UPDATE statement:

```plaintext
data master;
  update master minerals updatemode=nomissingcheck;
  by common;
run;
proc print data=master;
  title "Updated Data Set master";
  title2 "With Values Updated to Missing";
run;
```

In the following PROC PRINT output, the value of `plant` in observation 5 is set to missing because it is missing in the transaction data set and the `UPDATEMODE= NOMISSINGCHECK` option is in effect.
The UPDATE statement enables you to update values in one data set (the master data set) based on the values in another data set (transaction data set). The UPDATE statement creates a new output data set.

The BY statement is required and the input data sets must contain an index or be sorted by the values of the BY variables.

Because the UPDATE statement creates a new file when it generates output, you can add, delete, or rename variables when you perform an update.

By default, missing values in the transaction data set do not replace existing values in the master data set. You can change this behavior so that missing values in the transaction data set replace values in the master data set by specifying NOMISSINGCHECK in the UPDATEMODE= option.

If the transaction data set contains duplicate values of the BY variable, then the values from the transaction data set replace the values in the master data set.

If an observation in the transaction data set does not have a corresponding observation in the master data set, then SAS adds an observation to the master output data set. Observations are matched based on the value of the BY variable.

If no changes need to be made to an observation in the master data set, then that observation does not need to be included in the transaction data set.

See Also
- “Updating” on page 413
- Table 21.2 on page 416
- “UPDATE Statement” in SAS DATA Step Statements: Reference
- “SORT Procedure” in Base SAS Procedures Guide
- UPDATEMODE= in SAS DATA Step Statements: Reference
Example: Modify Data

Example: Modify a Data Set by Adding an Observation

Example Code

In this example, the MODIFY statement is used to update a master data set based on values contained in a transaction data set. The observations in the transaction data set are matched to the observations in the master data set by matching the values of the common variable, partNumber.

The data in this example represents inventory for a warehouse that stores tools and hardware. Each tool is uniquely identified by its part number. The master data set, Inventory, holds a record of the warehouse’s inventory. It is updated to reflect changes when a warehouse receives a new shipment of items. The InventoryAdd data set is the transaction data set. The transaction data set contains information about new items that are being added to the inventory (new kinds of tools). The transaction data set also adds inventory (newStock) to existing items and changes the price (newPrice) for existing items.

The following table shows the master input data set, Inventory, and the transaction data set, InventoryAdd:

<table>
<thead>
<tr>
<th>partNumber</th>
<th>partName</th>
<th>stock</th>
<th>price</th>
<th>receivedDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>K99R</td>
<td>seal</td>
<td>34</td>
<td>245.00</td>
<td>07jul2015</td>
</tr>
<tr>
<td>MAJ7</td>
<td>sander</td>
<td>98</td>
<td>45.88</td>
<td>20jun2015</td>
</tr>
<tr>
<td>LX43</td>
<td>filter</td>
<td>121</td>
<td>10.99</td>
<td>19may2016</td>
</tr>
<tr>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>27.87</td>
<td>10aug2016</td>
</tr>
<tr>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>9.55</td>
<td>16aug2016</td>
</tr>
<tr>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>24.50</td>
<td>20mar2016</td>
</tr>
<tr>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>19.77</td>
<td>18jun2016</td>
</tr>
<tr>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>11.55</td>
<td>09sep2016</td>
</tr>
<tr>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>13.99</td>
<td>09jan2017</td>
</tr>
<tr>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>34.50</td>
<td>03aug2017</td>
</tr>
</tbody>
</table>
To begin this example, first sort and print the `Inventory` and `InventoryAdd` data sets for comparison.

```plaintext
proc sort data=Inventory; by partNumber; run;
proc sort data=InventoryAdd; by partNumber; run;
proc print data=Inventory; title "Inventory"; run;
proc print data=InventoryAdd; title "InventoryAdd"; run;
```

Note: The `SORT` procedure is not required when modifying a data set using the `MODIFY` statement. The data sets in this example are sorted to better show the differences between the two data sets.

**Output 21.34**  PROC PRINT Output for the Master Inventory Data Set Sorted by `partNumber`

<table>
<thead>
<tr>
<th>Obs</th>
<th>partNumber</th>
<th>partName</th>
<th>stock</th>
<th>price</th>
<th>receivedDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BC35</td>
<td>clamp</td>
<td>80</td>
<td>$9.55</td>
<td>08/16/2017</td>
</tr>
<tr>
<td>2</td>
<td>BV1E</td>
<td>timer</td>
<td>26</td>
<td>$34.50</td>
<td>08/03/2018</td>
</tr>
<tr>
<td>3</td>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>$13.99</td>
<td>01/09/2018</td>
</tr>
<tr>
<td>4</td>
<td>K89R</td>
<td>seal</td>
<td>34</td>
<td>$245.00</td>
<td>07/07/2018</td>
</tr>
<tr>
<td>5</td>
<td>KJ66</td>
<td>cutter</td>
<td>6</td>
<td>$19.77</td>
<td>06/18/2018</td>
</tr>
<tr>
<td>6</td>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>$10.99</td>
<td>05/19/2018</td>
</tr>
<tr>
<td>7</td>
<td>M4J7</td>
<td>sander</td>
<td>98</td>
<td>$45.88</td>
<td>06/20/2018</td>
</tr>
<tr>
<td>8</td>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>$27.87</td>
<td>08/10/2018</td>
</tr>
<tr>
<td>9</td>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>$24.50</td>
<td>03/20/2017</td>
</tr>
<tr>
<td>10</td>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>$11.55</td>
<td>09/09/2017</td>
</tr>
</tbody>
</table>
Notice that observations 1, 2, and 4 in the transaction data set, InventoryAdd, do not exist in the master data set, Inventory. Also, notice that values for the variables newStock and newPrice in the transaction data set contain new values.

Now, modify the master data set based on the new information in the transaction data set, matching the observations by the unique values of the variable partNumber:

data Inventory;
   modify Inventory InventoryAdd; /*
   1*/
   by partNumber;
   select (_iorc_); /*
   2*/
   /*** The observation exists in the master data set  */
   when (%sysrc(_sok)) do; /*
   3*/
   stock = stock + newStock;
   price=newPrice;
   receivedDate = today();
   replace; /*
   4*/
   end;
   /*** The observation does not exist in the master data set*/
   when (%sysrc(_dsenmr)) do; /*
   5*/
   stock=newStock;
   price=newPrice;
   receivedDate=today();
   output; /*
   6*/
   _error_=0;
   end;
   otherwise do; /*
   7*/
   put "An unexpected I/O error has occurred."
   _error_ = 0;
   stop;
end;
end;
run;
proc sort data=Inventory;
  by partNumber;
run;
proc print data=Inventory;
  title "Modified Inventory Data Set Sorted by partNumber";
run;
quit;

1 The MODIFY statement loads the data from the master and transaction data sets. The BY statement matches observations from each data set based on the unique values of the variable partNumber.

2 If matches for partNumber from the transaction data set are found for partNumber in the master data set, then the _IORC_ automatic variable is automatically set to a code of _SOK_.

3 The %SYSRC autocall macro checks to see whether the value of _IORC_ is _SOK_. If the value is _SOK_, then the SELECT statement executes the first DO statement block. Because the observation in the transaction data set matches the observation in the master data set, the values in the observation can be updated by being replaced.

4 The REPLACE statement updates the master data set by replacing its observation with the observation from the transaction data set. The REPLACE statement updates observations 4, 7, and 8, (highlighted in blue in the output) with new values for stock and price. The stock values are updated based on the values for newStock in the transaction data set. The price values are updated based on the values for newPrice in the transaction data set. The receivedDate values for these observations are not updated because these are existing items that were received in the past.

5 If no matches for partNumber in the transaction data set are found for partNumber in the master data set, then the _IORC_ automatic variable is automatically set to a code of _DSENMR, which means that no match was found. The %SYSRC autocall macro checks to see whether the value of _IORC_ is _DSENMR_. If the value is _DSENMR_, then the SELECT statement executes the second DO block. Because the observation in the transaction data set does not exist in the master data set, the values cannot simply be replaced. An entire observation is created and added to the master data set.

6 The OUTPUT statement writes the new observation to the master data set. The OUTPUT statement adds observations 1, 2, and 5 to the master data set (see the observations highlighted in yellow in the output). The receivedDate values for these observations are updated based on the returned value for the TODAY function.

7 If neither condition is met, the OTHERWISE statement executes the last DO block and the PUT statement writes an error message to the log.

In the output below, the transaction data set contains three new items: hammer, wrench, and socket. Because some observations do not exist in the master data set and are being added from the transaction data set, an explicit OUTPUT statement is needed. For those observations that already exist in the master data set, the REPLACE statement is needed to update the values for these observations.

The program uses the OUTPUT statement to add observations 1, 2, and 5 to the master data set, and it uses the REPLACE statement to update observations 4, 7, and 8 with new values for stock and price.
Output 21.36  PROC PRINT Output for the Modified Inventory Master Data Set Sorted by partNumber

Modified Inventory Data Set Sorted by partNumber

<table>
<thead>
<tr>
<th>Obs</th>
<th>partNumber</th>
<th>partName</th>
<th>stock</th>
<th>price</th>
<th>receivedDate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AA11</td>
<td>hammer</td>
<td>55</td>
<td>$32.26</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>2</td>
<td>BB22</td>
<td>wrench</td>
<td>21</td>
<td>$17.35</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>3</td>
<td>BC85</td>
<td>clamp</td>
<td>80</td>
<td>$9.55</td>
<td>08/16/2017</td>
</tr>
<tr>
<td>4</td>
<td>BV1E</td>
<td>timer</td>
<td>56</td>
<td>$36.50</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>5</td>
<td>CC33</td>
<td>socket</td>
<td>7</td>
<td>$22.19</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>6</td>
<td>JD03</td>
<td>switch</td>
<td>383</td>
<td>$13.99</td>
<td>01/09/2018</td>
</tr>
<tr>
<td>7</td>
<td>K89R</td>
<td>seal</td>
<td>40</td>
<td>$247.50</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>8</td>
<td>KJ66</td>
<td>cutter</td>
<td>16</td>
<td>$24.50</td>
<td>01/25/2019</td>
</tr>
<tr>
<td>9</td>
<td>LK43</td>
<td>filter</td>
<td>121</td>
<td>$10.99</td>
<td>05/19/2018</td>
</tr>
<tr>
<td>10</td>
<td>M4J7</td>
<td>sander</td>
<td>98</td>
<td>$45.88</td>
<td>06/20/2018</td>
</tr>
<tr>
<td>11</td>
<td>MN21</td>
<td>brace</td>
<td>43</td>
<td>$27.87</td>
<td>08/10/2018</td>
</tr>
<tr>
<td>12</td>
<td>NCF3</td>
<td>valve</td>
<td>198</td>
<td>$24.50</td>
<td>03/20/2017</td>
</tr>
<tr>
<td>13</td>
<td>UYN7</td>
<td>rod</td>
<td>211</td>
<td>$11.55</td>
<td>09/09/2017</td>
</tr>
</tbody>
</table>

Note: Using OUTPUT or REPLACE in a DATA step overrides the default replacement of observations. If you use these statements in a DATA step, then you must explicitly program each action that you want to take.

Key Ideas

- The MODIFY statement updates the existing data set without creating a new output data set. Unlike the MODIFY statement, the SET, MERGE, and UPDATE statements create a new output data set. For more information, see Table 21.2 on page 416 and Table 21.4 on page 420.
- With MODIFY, you cannot add, delete, rename, or change variables.
- Both the master data set and the transaction data set can have observations with duplicate values of the BY variables. MODIFY treats the duplicates as follows:
  - If duplicate values exist in the master data set, only the first occurrence is updated.
  - If duplicates exist in the transaction data set, the last value of the duplicated variable overwrites the previous duplicated value. If you specify an accumulation statement to add all of the duplicated variables to the master observation, then the duplicated value is not overwritten.
See Also

- “Modifying” on page 414
- Table 21.2 on page 416
- “%SYSRC Autocall Macro” in SAS Macro Language: Reference
- “SELECT Statement” in SAS DATA Step Statements: Reference
- “REPLACE Statement” in SAS DATA Step Statements: Reference
- “OUTPUT Statement” in SAS DATA Step Statements: Reference
Indexes in SAS

A SAS index is an optional component of a SAS data set that enables SAS to access observations in the SAS data set quickly and efficiently. The purpose of SAS indexes is to optimize WHERE-clause processing and to facilitate BY-group processing.

For information about SAS indexes, see the documents that are listed in the following table.

<table>
<thead>
<tr>
<th>Engine</th>
<th>Documentation</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V9 engine</td>
<td>“Indexes” in SAS V9 LIBNAME Engine: Reference</td>
<td>Examples in the linked document show how to create SAS indexes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You can use an index to optimize WHERE or BY processing.</td>
</tr>
<tr>
<td>SPD Engine</td>
<td>“Features That Boost Processing Performance” in SAS Scalable Performance Data Engine: Reference</td>
<td>You can use the same language elements as with the V9 engine to create or use SAS indexes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indexes are stored differently than with the V9 engine. Additional language elements for the SPD Engine enable more control over index usage. In addition, the engine provides an automatic sort for BY processing, so an unsorted data set does not require an index.</td>
</tr>
<tr>
<td>Engine</td>
<td>Documentation</td>
<td>Usage Notes</td>
</tr>
<tr>
<td>------------------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CAS engine</td>
<td>&quot;Indexing&quot; in SAS Cloud Analytic Services: Fundamentals</td>
<td>You can use one of several CAS actions to create an index. The syntax is different from the V9 engine. You can use an index to improve WHERE processing, especially on very large tables.</td>
</tr>
<tr>
<td>SAS/ACCESS engines</td>
<td>SAS/ACCESS software documentation on support.sas.com</td>
<td>SAS/ACCESS engines do not create or use SAS indexes. You can request for the engine to use an existing DBMS index, or to use a DBMS column as a key variable. If an appropriate DBMS index is found, then the join WHERE clause is passed down to the DBMS, which is very efficient. If the index is not found or is not appropriate, then all data is transferred to SAS for the join process.</td>
</tr>
</tbody>
</table>
Using Arrays

Definitions for Array Processing

Rules for Referencing Arrays

A Conceptual View of Arrays

Syntax for Defining and Referencing an Array

Processing Simple Arrays

Variations on Basic Array Processing

Multidimensional Arrays: Creating and Processing

Specifying Array Bounds

Examples

Definitions for Array Processing
array
is a temporary grouping of SAS variables that are arranged in a particular order and identified by an array-name. The array exists only for the duration of the current DATA step. The array-name distinguishes it from any other arrays in the same DATA step; it is not a variable.

Note: Arrays in SAS are different from those in many other programming languages. In SAS, an array is not a data structure. An array is just a convenient way of temporarily identifying a group of variables.

array processing
is a method that enables you to perform the same tasks for a series of related variables.

array reference
is a method to reference the elements of an array.

one-dimensional array
is a simple grouping of variables that, when processed, results in output that can be represented in simple row format.

multidimensional array
is a more complex grouping of variables that, when processed, results in output that could have two or more dimensions, such as columns and rows.

Basic array processing involves the following steps:

- grouping variables into arrays
- selecting a current variable for an action
- repeating an action

Rules for Referencing Arrays

Rules for Referencing Arrays

Before you make any references to an array, an ARRAY statement must appear in the same DATA step that you used to create the array. Once you have created the array, you can perform the following tasks:

- Use an array reference anywhere that you can write a SAS expression.
- Use an array reference as the arguments of some SAS functions.
- Use a subscript enclosed in braces, brackets, or parentheses to reference an array.
- Use the special array subscript asterisk (*) to refer to all variables in an array in an INPUT or PUT statement or in the argument of a function.

Note: You cannot use the asterisk with __TEMPORARY__ arrays.
An array definition is in effect only for the duration of the DATA step. If you want to use the same array in several DATA steps, you must redefine the array in each step. You can, however, redefine the array with the same variables in a later DATA step by using a macro variable. A macro variable is useful for storing the variable names that you need, as shown in this example:

```
%let list=NC SC GA VA;

data one;
  array state{*} &list;
  ... more SAS statements ...
run;

data two;
  array state{*} &list;
  ... more SAS statements ...
run;
```

**A Conceptual View of Arrays**

**One-Dimensional Array**

The following figure is a conceptual representation of two one-dimensional arrays, Misc and Mday.

![Figure 23.1 One-Dimensional Array](image)

Misc contains eight elements, the variables Misc1 through Misc8. To reference the data in these variables, use the form Misc\(n\), where \(n\) is the element number in the array. For example, Misc(6) is the sixth element in the array.

Mday contains seven elements, the variables Mday1 through Mday7. Mday(3) is the third element in the array.

**Two-Dimensional Array**

The following figure is a conceptual representation of the two-dimensional array Expenses.
The Expenses array contains ten groups of eight variables each. The ten groups (expense categories) comprise the first dimension of the array, and the eight variables (days of the week) comprise the second dimension. To reference the data in the array variables, use the form Expenses{\(m,n\)}, where \(m\) is the element number in the first dimension of the array, and \(n\) is the element number in the second dimension of the array. Expenses{6,4} references the value of dues for the fourth day (the variable is Dues4).

Syntax for Defining and Referencing an Array

To define a simple or a multidimensional array, use the ARRAY statement. The ARRAY statement has the following form:

\[
\text{ARRAY array-name \{number-of-elements\} \<$\array-elements\ <$initial-value-list\>;}$
\]

where

array-name
is a SAS name that identifies the group of variables.

number-of-elements
is the number of variables in the group. You must enclose this value in either parentheses (), braces {}, or brackets [].

$ specifies that the elements in the array are character elements.

length
specifies the length of the elements in the array that have not been previously assigned a length.
array-elements
is a list of the names of the variables in the group. All variables that are defined
in a given array must be of the same type, either all character or all numeric.

initial-value-list
is a list of the initial values for the corresponding elements in the array.

For complete information, see the “ARRAY Statement” in SAS DATA Step
Statements: Reference.

To reference an array that was previously defined in the same DATA step, use an
Array Reference statement. An array reference has the following form:

array-name {subscript}

where

array-name
is the name of an array that was previously defined with an ARRAY statement in
the same DATA step.

subscript
specifies the subscript, which can be a numeric constant, the name of a variable
whose value is the number, a SAS numeric expression, or an asterisk (*).

Note: Subscripts in SAS are 1-based by default, and not 0-based as they are in
some other programming languages.

For complete information, see the Array Reference statement in the SAS DATA Step
Statements: Reference.

--

Processing Simple Arrays

Grouping Variables in a Simple Array

The following ARRAY statement creates an array named Books that contains the
three variables Reference, Usage, and Introduction:

array books{3} Reference Usage Introduction;

When you define an array, SAS assigns each array element an array reference with
the form array-name(subscript), where subscript is the position of the variable in the
list. The following table lists the array reference assignments for the previous
ARRAY statement:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Array Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>books{1}</td>
</tr>
<tr>
<td>Usage</td>
<td>books{2}</td>
</tr>
</tbody>
</table>
Later in the DATA step, when you want to process the variables in the array, you can refer to a variable by either its name or its array reference. For example, the names Reference and Books{1} are equivalent.

### Using a DO Loop to Repeat an Action

To perform the same action several times, use an iterative DO loop. A simple iterative DO loop that processes an array has the following form:

```sas
DO index-variable = 1 TO number-of-elements-in-array;
   … more SAS statements …
END;
```

The loop is processed repeatedly (iterates) according to the instructions in the iterative DO statement. The iterative DO statement contains an `index-variable` whose name you specify and whose value changes at each iteration of the loop.

To execute the loop as many times as there are variables in the array, specify that the values of `index-variable` are 1 TO `number-of-elements-in-array`. SAS increases the value of `index-variable` by 1 before each new iteration of the loop. When the value exceeds the `number-of-elements-in-array`, SAS stops processing the loop. By default, SAS automatically includes `index-variable` in the output data set. Use a DROP statement or the DROP= data set option to prevent the index variable from being written to your output data set.

An iterative DO loop that executes three times and has an index variable named count has the following form:

```sas
do count=1 to 3;
   … more SAS statements …
end;
```

The first time that the loop processes, the value of count is 1; the second time, 2; and the third time, 3. At the beginning of the fourth iteration, the value of count is 4, which exceeds the specified range and causes SAS to stop processing the loop.

### Using a DO Loop to Process Selected Elements in an Array

To process particular elements of an array, specify those elements as the range of the iterative DO statement. For example, the following statement creates an array Days that contains seven elements:

```sas
array days{7} D1-D7;
```

The following DO statements process selected elements of the array Days:
Table 23.2  DO Statement Processing

<table>
<thead>
<tr>
<th>DO Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>do i=2 to 4;</td>
<td>processes elements 2 through 4</td>
</tr>
<tr>
<td>do i=1 to 7 by 2;</td>
<td>processes elements 1, 3, 5, and 7</td>
</tr>
<tr>
<td>do i=3,5;</td>
<td>processes elements 3 and 5</td>
</tr>
</tbody>
</table>

Selecting the Current Variable

You must tell SAS which variable in the array to use in each iteration of the loop. Recall that you identify variables in an array by their array references and that you use a variable name, a number, or an expression as the subscript of the reference. Therefore, you can write programming statements so that the index variable of the DO loop is the subscript of the array reference (for example, array-name(index-variable)). When the value of the index variable changes, the subscript of the array reference (and therefore the variable that is referenced) also changes.

The following example uses the index variable count as the subscript of array references inside a DO loop:

```
array books{3} Reference Usage Introduction;
do count=1 to 3;
   if books{count}=. then books{count}=0;
end;
```

When the value of count is 1, SAS reads the array reference as Books{1} and processes the IF-THEN statement on Books{1}, which is the variable Reference. When count is 2, SAS processes the statement on Books{2}, which is the variable Usage. When count is 3, SAS processes the statement on Books{3}, which is the variable Introduction.

The statements in the example tell SAS to
- perform the actions in the loop three times
- replace the array subscript count with the current value of count for each iteration of the IF-THEN statement
- locate the variable with that array reference and process the IF-THEN statement on it
- replace missing values with zero if the condition is true.

The following DATA step defines the array Book and processes it with a DO loop.

```
options linesize=80 pagesize=60;
data changed(drop=count);
   input Reference Usage Introduction;
   array book{3} Reference Usage Introduction;
do count=1 to 3;
   if book{count}=. then book{count}=0;
end;
```
The following output shows the CHANGED data set.

Output 23.1 Using an Array Statement to Process Missing Data Values

<table>
<thead>
<tr>
<th>Obs</th>
<th>Reference</th>
<th>Usage</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45</td>
<td>63</td>
<td>113</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>75</td>
<td>150</td>
</tr>
<tr>
<td>3</td>
<td>62</td>
<td>0</td>
<td>98</td>
</tr>
</tbody>
</table>

Defining the Number of Elements in an Array

When you define the number of elements in an array, you can either use an asterisk enclosed in braces ({*}), brackets ([*]), or parentheses (**) to count the number of elements or to specify the number of elements. You must list each array element if you use the asterisk to designate the number of elements. In the following example, the array C1Temp references five variables with temperature measures.

array c1temp{*} c1t1 c1t2 c1t3 c1t4 c1t5;

If you specify the number of elements explicitly, you can omit the names of the variables or array elements in the ARRAY statement. SAS then creates variable names by concatenating the array name with the numbers 1, 2, 3, and so on. If a variable name in the series already exists, SAS uses that variable instead of creating a new one. In the following example, the array c1t references five variables: c1t1, c1t2, c1t3, c1t4, and c1t5.

array c1t{5};

Rules for Referencing Arrays

Before you make any references to an array, an ARRAY statement must appear in the same DATA step that you used to create the array. Once you have created the array, you can perform the following tasks:
Use an array reference anywhere that you can write a SAS expression.

Use an array reference as the arguments of some SAS functions.

Use a subscript enclosed in braces, brackets, or parentheses to reference an array.

Use the special array subscript asterisk (*) to refer to all variables in an array in an INPUT or PUT statement or in the argument of a function.

Note: You cannot use the asterisk with _TEMPORARY_ arrays.

An array definition is in effect only for the duration of the DATA step. If you want to use the same array in several DATA steps, you must redefine the array in each step. You can, however, redefine the array with the same variables in a later DATA step by using a macro variable. A macro variable is useful for storing the variable names that you need, as shown in this example:

```sas
%let list=NC SC GA VA;
data one;
  array state{*} &list;
  ... more SAS statements ...
run;

data two;
  array state{*} &list;
  ... more SAS statements ...
run;
```

**Variations on Basic Array Processing**

**Determining the Number of Elements in an Array Efficiently**

The DIM function in the iterative DO statement returns the number of elements in a one-dimensional array or the number of elements in a specified dimension of a multidimensional array, when the lower bound of the dimension is 1. Use the DIM function to avoid changing the upper bound of an iterative DO group each time you change the number of elements in the array.

The form of the DIM function is as follows:

```
DIMn(array-name)
```

where \( n \) is the specified dimension that has a default value of 1.

You can also use the DIM function when you specify the number of elements in the array with an asterisk. Here are some examples of the DIM function:

- \( \text{do } i=1 \text{ to dim(days)}; \)
- \( \text{do } i=1 \text{ to dim4(days) by 2}; \)
DO WHILE and DO UNTIL Expressions

Arrays are often processed in iterative DO loops that use the array reference in a DO WHILE or DO UNTIL expression. In this example, the iterative DO loop processes the elements of the array named Trend.

```sas
data test;
  array trend{5} x1-x5;
  input x1-x5 y;
  do i=1 to 5 while(trend{i}<y);
    /* more SAS statements */
  end;
  datalines;
  /* data lines */
;```

Using Variable Lists to Define an Array Quickly

SAS reserves the following three names for use as variable list names:

- `_CHARACTER_`
- `_NUMERIC_`
- `_ALL_`

You can use these variable list names to reference variables that have been previously defined in the same DATA step. The `_CHARACTER_` variable lists character values only. The `_NUMERIC_` variable lists numeric values only. The `_ALL_` variable lists either all character or all numeric values, depending on how you previously defined the variables.

For example, the following INPUT statement reads in variables X1 through X3 as character values using the $8. informat, and variables X4 through X5 as numeric variables. The following ARRAY statement uses the variable list `_CHARACTER_` to include only the character variables in the array. The asterisk indicates that SAS determines the subscript by counting the variables in the array.

```sas
input (X1-X3) ($8.) X4-X5;
array item {*} _character_;```

You can use the `_NUMERIC_` variable in your program (for example, you need to convert currency). In this application, you do not need to know the variable names. You need only to convert all values to the new currency.

For more information about variable lists, see the “ARRAY Statement” in SAS DATA Step Statements: Reference.
Multidimensional Arrays: Creating and Processing

Grouping Variables in a Multidimensional Array

To create a multidimensional array, place the number of elements in each dimension after the array name in the form \{n, … \} where \(n\) is required for each dimension of a multidimensional array.

From right to left, the rightmost dimension represents columns; the next dimension represents rows. Each position farther left represents a higher dimension. The following ARRAY statement defines a two-dimensional array with two rows and five columns. The array contains ten variables: five temperature measures (t1 through t5) from two cities (c1 and c2):

\[
\text{array temprg}\{2,5\} \text{ c1t1-c1t5 c2t1-c2t5;}
\]

SAS places variables into a multidimensional array by filling all rows in order, beginning at the upper left corner of the array (known as row-major order). You can think of the variables as having the following arrangement:

\[
\begin{align*}
\text{c1t1} & \quad \text{c1t2} & \quad \text{c1t3} & \quad \text{c1t4} & \quad \text{c1t5} \\
\text{c2t1} & \quad \text{c2t2} & \quad \text{c2t3} & \quad \text{c2t4} & \quad \text{c2t5}
\end{align*}
\]

To refer to the elements of the array later with an array reference, you can use the array name and subscripts. The following table lists some of the array references for the previous example:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Array Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1t1</td>
<td>temprg{1,1}</td>
</tr>
<tr>
<td>c1t2</td>
<td>temprg{1,2}</td>
</tr>
<tr>
<td>c2t2</td>
<td>temprg{2,2}</td>
</tr>
<tr>
<td>c2t5</td>
<td>temprg{2,5}</td>
</tr>
</tbody>
</table>

Using Nested DO Loops

Multidimensional arrays are usually processed inside nested DO loops. For example, the following is one form that processes a two-dimensional array:
DO index-variable-1=1 TO number-of-rows;
    DO index-variable-2=1 TO number-of-columns;
        ... more SAS statements ...
    END;
END;

An array reference can use two or more index variables as the subscript to refer to two or more dimensions of an array. Use the following form:

array-name \{index-variable-1, ..., index-variable-n\}

The following example creates an array that contains ten variables—five temperature measures (t1 through t5) from two cities (c1 and c2). The DATA step contains two DO loops.

- The outer DO loop (DO I=1 TO 2) processes the inner DO loop twice.
- The inner DO loop (DO J=1 TO 5) applies the ROUND function to all the variables in one row.

For each iteration of the DO loops, SAS substitutes the value of the array element corresponding to the current values of I and J.

```sas
options linesize=80 pagesize=60;

data temps;
    array temprg{2,5} c1t1-c1t5 c2t1-c2t5;
    input c1t1-c1t5 /
        c2t1-c2t5;
    do i=1 to 2;
        do j=1 to 5;
            temprg{i,j}=round(temprg{i,j});
        end;
    end;
    datalines;
    89.5 65.4 75.3 77.7 89.3
    73.7 87.3 89.9 98.2 35.6
    75.8 82.1 98.2 93.5 67.7
    101.3 86.5 59.2 35.6 75.7
;
proc print data=temps;
    title 'Temperature Measures for Two Cities';
run;
```

The following data set Temps contains the values of the variables rounded to the nearest whole number.

**Output 23.2 Using a Multidimensional Array**

<table>
<thead>
<tr>
<th></th>
<th>c1t1</th>
<th>c1t2</th>
<th>c1t3</th>
<th>c1t4</th>
<th>c1t5</th>
<th>c2t1</th>
<th>c2t2</th>
<th>c2t3</th>
<th>c2t4</th>
<th>c2t5</th>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>90</td>
<td>65</td>
<td>75</td>
<td>78</td>
<td>89</td>
<td>74</td>
<td>87</td>
<td>90</td>
<td>98</td>
<td>36</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>76</td>
<td>82</td>
<td>98</td>
<td>94</td>
<td>68</td>
<td>101</td>
<td>87</td>
<td>53</td>
<td>36</td>
<td>76</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>
The previous example can also use the DIM function to produce the same result:

do
i=1 to dim1(temprg);
  do j=1 to dim2(temprg);
    temprg{i,j}=round(temprg[i,j]);
  end;
end;

The value of DIM1(TEMPRG) is 2; the value of DIM2(TEMPRG) is 5.

Specifying Array Bounds

Identifying Upper and Lower Bounds

Typically, in an ARRAY statement, the subscript in each dimension of the array ranges from 1 to $n$, where $n$ is the number of elements in that dimension. Thus, 1 is the lower bound and $n$ is the upper bound of that dimension of the array. For example, in the following array, the lower bound is 1 and the upper bound is 4:

array new{4} Jackson Poulenc Andrew Parson;

In the following ARRAY statement, the bounds of the first dimension are 1 and 2 and those of the second dimension are 1 and 5:

array test{2,5} test1-test10;

Bounded array dimensions have the following form:

{<lower-1>:<upper-1>,...,<lower-n>:<upper-n>}

Therefore, you can also write the previous ARRAY statements as follows:

array new{1:4} Jackson Poulenc Andrew Parson;
array test{1:2,1:5} test1-test10;

For most arrays, 1 is a convenient lower bound, so you do not need to specify the lower bound. However, specifying both the lower and the upper bounds is useful when the array dimensions have beginning points other than 1.

In the following example, ten variables are named Year76 through Year85. The following ARRAY statements place the variables into two arrays named First and Second:

array first{10} Year76-Year85;
array second{76:85} Year76-Year85;

In the first ARRAY statement, the element first{4} is variable Year79, first{7} is Year82, and so on. In the second ARRAY statement, element second{79} is Year79 and second{82} is Year82.

To process the array names Second in a DO group, make sure that the range of the DO loop matches the range of the array as follows:

do i=76 to 85;
  if second{i}=9 then second{i}=.;
end;
Determining Array Bounds: LBOUND and HBOUND Functions

You can use the LBOUND and HBOUND functions to determine array bounds. The LBOUND function returns the lower bound of a one-dimensional array or the lower bound of a specified dimension of a multidimensional array. The HBOUND function returns the upper bound of a one-dimensional array or the upper bound of a specified dimension of a multidimensional array.

The form of the LBOUND and HBOUND functions is as follows:

\[
\text{LBOUND}(n)(\text{array-name}) \\
\text{HBOUND}(n)(\text{array-name})
\]

where

\( n \)

is the specified dimension and has a default value of 1.

You can use the LBOUND and HBOUND functions to specify the starting and ending values of the iterative DO loop to process the elements of the array named Second:

\[
do \ i=\text{lbound}(\text{second}) \space \text{to} \space \text{hbound}(\text{second}); \\
\quad \text{if} \ \text{second}(i)=9 \space \text{then} \ \text{second}(i)=. ; \\
end;
\]

In this example, the index variable in the iterative DO statement ranges from 76 to 85.

When to Use the HBOUND Function Instead of the DIM Function

The following ARRAY statement defines an array containing a total of five elements, a lower bound of 72, and an upper bound of 76. It represents the calendar years 1972 through 1976:

\[
\text{array} \ \text{years}(72:76) \ \text{first} \ \text{second} \ \text{third} \ \text{fourth} \ \text{fifth} ;
\]

To process the array named YEARS in an iterative DO loop, make sure that the range of the DO loop matches the range of the array as follows:

\[
\text{do} \ i=\text{lbound(years)} \ \text{to} \ \text{hbound(years)} ; \\
\quad \text{if} \ \text{years}(i)=99 \ \text{then} \ \text{years}(i)=. ; \\
end;
\]

The value of LBOUND(YEARS) is 72; the value of HBOUND(YEARS) is 76.

For this example, the DIM function would return a value of 5, the total count of elements in the array YEARS. Therefore, if you used the DIM function instead of the HBOUND function for the upper bound of the array, the statements inside the DO loop would not have executed.
Specifying Bounds in a Two-Dimensional Array

The following list contains 40 variables named X60 through X99. They represent the years 1960 through 1999.

<table>
<thead>
<tr>
<th>X60</th>
<th>X61</th>
<th>X62</th>
<th>X63</th>
<th>X64</th>
<th>X65</th>
<th>X66</th>
<th>X67</th>
<th>X68</th>
<th>X69</th>
</tr>
</thead>
<tbody>
<tr>
<td>X70</td>
<td>X71</td>
<td>X72</td>
<td>X73</td>
<td>X74</td>
<td>X75</td>
<td>X76</td>
<td>X77</td>
<td>X78</td>
<td>X79</td>
</tr>
<tr>
<td>X80</td>
<td>X81</td>
<td>X82</td>
<td>X83</td>
<td>X84</td>
<td>X85</td>
<td>X86</td>
<td>X87</td>
<td>X88</td>
<td>X89</td>
</tr>
<tr>
<td>X90</td>
<td>X91</td>
<td>X92</td>
<td>X93</td>
<td>X94</td>
<td>X95</td>
<td>X96</td>
<td>X97</td>
<td>X98</td>
<td>X99</td>
</tr>
</tbody>
</table>

The following ARRAY statement arranges the variables in an array by decades. The rows range from 6 through 9, and the columns range from 0 through 9.

```
array X{6:9,0:9} X60-X99;
```

In array X, variable X63 is element X{6,3} and variable X89 is element X{8,9}. To process array X with iterative DO loops, use one of these methods:

- **Method 1:**
  ```
  do i=6 to 9;
  do j=0 to 9;
    if X{i,j}=0 then X{i,j}=.;
  end;
  end;
  ```

- **Method 2:**
  ```
  do i=lbound1(X) to hbound1(X);
  do j=lbound2(X) to hbound2(X);
    if X{i,j}=0 then X{i,j}=.;
  end;
  end;
  ```

Both examples change all values of 0 in variables X60 through X99 to missing. The first example sets the range of the DO groups explicitly. The second example uses the LBOUND and HBOUND functions to return the bounds of each dimension of the array.

Examples

Example: Using Character Variables in an Array

You can specify character variables and their lengths in ARRAY statements. The following example groups variables into two arrays, NAMES and CAPITALS.

```
The dollar sign ($) tells SAS to create the elements as character variables. If the variables have already been declared as character variables, a dollar sign in the array is not necessary.

2 The INPUT statement reads all the variables in array NAMES.

3 The statement inside the DO loop uses the UPCASE function to change the values of the variables in array NAMES to uppercase. The statement then stores the uppercase values in the variables in the CAPITALS array.

The following output shows the TEXT data set.

**Output 23.3 Using Character Variables in an Array**

### Names Changed from Lowercase to Uppercase

<table>
<thead>
<tr>
<th>Obs</th>
<th>n1</th>
<th>n2</th>
<th>n3</th>
<th>n4</th>
<th>n5</th>
<th>c1</th>
<th>c2</th>
<th>c3</th>
<th>c4</th>
<th>c5</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smithers</td>
<td>Michaels</td>
<td>Gonzalez</td>
<td>Hurt</td>
<td>Frank</td>
<td>Smithers</td>
<td>Michaels</td>
<td>Gonzalez</td>
<td>Hurt</td>
<td>Frank</td>
<td>6</td>
</tr>
</tbody>
</table>

---

**Example: Assigning Initial Values to the Elements of an Array**

This example creates variables in the array Test and assigns them the initial values 90, 80, and 70. It reads values into another array named Score and compares each element of Score to the corresponding element of Test.

```sas
options linesize=80 pagesize=60;

data score1(drop=i);
  array test{3} t1-t3 (90 80 70); /* 1 */
  array score{3} s1-s3;
  input id score{*}; /* 2 */
  do i=1 to 3;
    if score{i}>=test{i} then /* 3 */
      do;
        NewScore=score{i};
        outputScore=score{i}; /* 4 */
      end;
  end;
end;
```
Assign the initial values 90, 80, and 70 to the Test array and assign the variables s1, s2, and s3 to the Score array.

The INPUT statement reads a value for the variable named ID and then reads values for all the variables in the Score array.

If the value of the element in Score is greater than or equal to the value of the element in Test, the variable NewScore is assigned the value in the element Score.

The OUTPUT statement writes the observation to the SAS data set.

The following output shows the Score1 data set.

Output 23.4 Assigning Initial Values to the Elements of an Array

Data Set SCORE1

<table>
<thead>
<tr>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>id</th>
<th>NewScore</th>
</tr>
</thead>
<tbody>
<tr>
<td>90</td>
<td>80</td>
<td>70</td>
<td>99</td>
<td>60</td>
<td>82</td>
<td>1234</td>
<td>99</td>
</tr>
<tr>
<td>90</td>
<td>80</td>
<td>70</td>
<td>99</td>
<td>60</td>
<td>82</td>
<td>1234</td>
<td>82</td>
</tr>
<tr>
<td>90</td>
<td>80</td>
<td>70</td>
<td>80</td>
<td>85</td>
<td>75</td>
<td>5678</td>
<td>85</td>
</tr>
<tr>
<td>90</td>
<td>80</td>
<td>70</td>
<td>80</td>
<td>85</td>
<td>75</td>
<td>5678</td>
<td>75</td>
</tr>
</tbody>
</table>

Example: Creating an Array for Temporary Use in the Current DATA Step

When elements of an array are constants that are needed only for the duration of the DATA step, you can omit variables from an array group and instead use temporary array elements. You refer to temporary data elements by the array name and dimension. Although they behave like variables, temporary array elements do not have names, and they do not appear in the output data set. Temporary array elements are automatically retained, instead of being reset to missing at the beginning of the next iteration of the DATA step.

To create a temporary array, use the _TEMPORARY_ argument. The following example creates a temporary array named Test:

```
options linesize=80 pagesize=60;

data score2(drop=i);
```
array test{3} _temporary_ (90 80 70);
array score{3} s1-s3;
input id score{*};
  do i=1 to 3;
    if score{i}>=test{i} then do;
      NewScore=score{i};
      output;
    end;
  end;
datalines;
1234  99 60 82
5678  80 85 75
;

proc print noobs data=score2;
  title 'Data Set SCORE2';
run;

The following output shows the Score2 data set.

Output 23.5 Using _TEMPORARY_ Arrays

<table>
<thead>
<tr>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>id</th>
<th>NewScore</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>60</td>
<td>82</td>
<td>1234</td>
<td>99</td>
</tr>
<tr>
<td>99</td>
<td>60</td>
<td>82</td>
<td>1234</td>
<td>82</td>
</tr>
<tr>
<td>80</td>
<td>85</td>
<td>75</td>
<td>5678</td>
<td>85</td>
</tr>
<tr>
<td>80</td>
<td>85</td>
<td>75</td>
<td>5678</td>
<td>75</td>
</tr>
</tbody>
</table>

Example: Performing an Action on All Numeric Variables

This example multiplies all of the numeric variables in array Test by 3.

options nodate pageno=1 linesize=80 pagesize=60;

data sales;
  infile datalines;
  input Value1 Value2 Value3 Value4;
  datalines;
  11 56 58 61
  22 51 57 61
  22 49 53 58
;
data convert(drop=i);
  set sales;
  array test{*} _numeric_;
do i=1 to dim(test);
    test[i] = (test[i]*3);
end;
run;

proc print data=convert;
title 'Data Set CONVERT';
run;

The following output shows the CONVERT data set.

**Output 23.6**  Output from Using a _NUMERIC_ Variable List

<table>
<thead>
<tr>
<th>Obs</th>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
<th>Value4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33</td>
<td>168</td>
<td>174</td>
<td>183</td>
</tr>
<tr>
<td>2</td>
<td>66</td>
<td>153</td>
<td>171</td>
<td>183</td>
</tr>
<tr>
<td>3</td>
<td>66</td>
<td>147</td>
<td>159</td>
<td>174</td>
</tr>
</tbody>
</table>
Types of Errors in SAS

Summary of Types of Errors That SAS Recognizes

SAS performs error processing during both the compilation and the execution phases of SAS processing. You can debug SAS programs by understanding processing messages in the SAS log and then fixing your code. You can use the DATA Step Debugger to detect logic errors in a DATA step during execution.

SAS recognizes five types of errors.
### Table 24.1  Types of Errors

<table>
<thead>
<tr>
<th>Type of Error</th>
<th>When This Error Occurs</th>
<th>When the Error Is Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>syntax</td>
<td>when programming statements do not conform to the rules of the SAS language</td>
<td>compile time</td>
</tr>
<tr>
<td>semantic</td>
<td>when the language element is correct, but the element might not be valid for a particular usage</td>
<td>compile or execution time</td>
</tr>
<tr>
<td>execution-time</td>
<td>when SAS attempts to execute a program and execution fails</td>
<td>execution time</td>
</tr>
<tr>
<td>data</td>
<td>when data values are invalid</td>
<td>execution time</td>
</tr>
<tr>
<td>macro-related</td>
<td>when you use the macro facility incorrectly</td>
<td>macro compile time or execution time, DATA, or PROC step compile time or execution time</td>
</tr>
</tbody>
</table>

### Syntax Errors

Syntax errors occur when program statements do not conform to the rules of the SAS language. Here are some examples of syntax errors:

- misspelled SAS keyword
- unmatched quotation marks
- missing a semicolon
- invalid statement option
- invalid data set option

When SAS encounters a syntax error, it attempts to correct the error by interpreting the intent of the code. Then SAS continues processing your program based on these assumptions. If SAS cannot correct the error, it prints an error message to the log. If you do not want SAS to correct syntax errors, you can set the NOAUTOCORRECT system option. For more information, see “AUTOCORRECT System Option” in SAS System Options: Reference.

In the following example, the DATA statement is misspelled, and SAS prints a warning message to the log. Because SAS could interpret the misspelled word, the program runs and produces output.

```sas
date temp; /* 1 */
   x=1;
run;
```
Some errors are explained fully by the message that SAS prints in the log. Other error messages are not as easy to interpret because SAS is not always able to detect exactly where the error occurred. For example, when you fail to end a SAS statement with a semicolon, SAS does not always detect the error at the point where it occurs. The free-format nature of SAS statements (they can begin and end anywhere) can cause this issue. In the following example, the semicolon at the end of the DATA statement is missing. SAS prints the word ERROR in the log, identifies the possible location of the error, prints an explanation of the error, and stops processing the DATA step.

```
data temp
    x=1;
run;
```

```
proc print data=temp;
run;
```

```
data is misspelled as date.
```

**Example Code 24.1** SAS Log: Syntax Error (Misspelled Key Word)
Example Code 24.2  SAS Log: Syntax Error (Missing Semicolon)

```sas
67   data temp
68      x=1;
-  
22  76
ERROR 22-322: Syntax error, expecting one of the following: a name, a quoted string, (, /, ;, _DATA_, _LAST_, _NULL_.
ERROR 76-322: Syntax error, statement will be ignored.
69   run;
NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
    real time           0.01 seconds
    cpu time            0.01 seconds
70  71   proc print data=temp;
72   run;
NOTE: There were 1 observations read from the data set WORK.TEMP.
NOTE: PROCEDURE PRINT used (Total process time):
    real time           0.00 seconds
    cpu time            0.00 seconds
73   proc printto; run;
```

Whether subsequent steps are executed depends on which method you use to run SAS, as well as your operating environment.

Note: You can add these lines to your code to fix unmatched comment tags, unmatched quotation marks, and missing semicolons:

```sas
/* ';' * ';' */
quit;
run;
```

Semantic Errors

Semantic errors occur when the form of the elements in a SAS statement is correct, but the elements are not valid for that usage. Semantic errors are detected at compile time and can cause SAS to enter syntax check mode. (For a description of syntax check mode, see “Syntax Check Mode” on page 529.)

Examples of semantic errors include the following:

- specifying the wrong number of arguments for a function
- using a numeric variable name where only a character variable is valid
- using invalid references to an array
- a variable is not initialized

In the following example, SAS detects an invalid reference to the array all at compile time:

```sas
data _null_;
```sas
array all{*} x1-x5;
all=3;
datalines;
1 1.5
. 3
2 4.5
3 2 7
3 . .
;
run;
```

**Example Code 24.3  SAS Log: Semantic Error (invalid Reference to an Array)**

```sas
data _null_;  
array all{*} x1-x5;  
ERROR: invalid reference to the array all.  
all=3;  
datalines;  
NOTE: The SAS System stopped processing this step because of errors.  
NOTE: DATA statement used (Total process time):  
real time 0.15 seconds  
cpu time 0.01 seconds  
;
run;  
```

Here is another example of a semantic error that occurs at compile time. In this DATA step, the libref SomeLib has not been previously assigned in a LIBNAME statement.

```sas
data test;  
set somelib.old;  
run;
```

**Example Code 24.4  SAS Log: Semantic Error (Libref Not Previously Assigned)**

```sas
data test;  
ERROR: Libname SomeLib is not assigned.  
set somelib.old;  
run;  
NOTE: The SAS System stopped processing this step because of errors.  
WARNING: The data set WORK.TEST may be incomplete. When this step was stopped there were 0 observations and 0 variables.
```

SAS writes a message like "NOTE: SAS went to a new line when input statement reached past the end of a line." when a semantic error occurs at execution time. This note is written to the SAS log when FLOWOVER is used and all the variables in the INPUT statement cannot be fully read.

The detection of a variable that is not initialized is another semantic error. By default, SAS does not report an error, but writes a note to the SAS log. If a variable is not initialized and the system option VARINITCHK=ERROR has been issued, SAS stops processing a DATA step and writes an error message to the SAS log.
Execution-Time Errors

Definition

Execution-time errors occur when SAS executes a program that processes data values. Most execution-time errors produce warning messages or notes in the SAS log but allow the program to continue executing. The location of an execution-time error is usually given as line and column numbers in a note or error message.

Note: When you run SAS in noninteractive mode, more serious errors can cause SAS to enter syntax check mode and stop processing the program.

Common execution-time errors include the following:

- invalid arguments to functions
- invalid mathematical operations (for example, division by 0)
- observations in the wrong order for BY-group processing
- reference to a nonexistent member of an array (occurs when the array's subscript is out of range)
- open and close errors on SAS data sets and other files in INFILE and FILE statements
- INPUT statements that do not match the data lines (for example, an INPUT statement in which you list the wrong columns for a variable or fail to indicate that the variable is a character variable)

Out-of-Resources Condition

An execution-time error can also occur when you encounter an out-of-resources condition, such as a full disk, or insufficient memory for a SAS procedure to complete. When these conditions occur, SAS attempts to find resources for current use. For example, SAS might ask the user for permission to perform these actions in out-of-resource conditions:

- Delete temporary data sets that might no longer be needed.
- Free the memory in which macro variables are stored.

When an out-of-resources condition occurs in a windowing environment, you can use the SAS CLEANUP system option to display a requestor panel. The requestor panel enables you to choose how to resolve the error. When you run SAS in batch, noninteractive, or interactive line mode, the operation of CLEANUP depends on your operating environment. For more information, see “CLEANUP System Option” in SAS System Options: Reference and in the SAS documentation for your operating system.

Examples

In the following example, an execution-time error occurs when SAS uses data values from the second observation to perform the division operation in the
assignment statement. Division by 0 is an invalid mathematical operation and causes an execution-time error.

```
data inventory;
  input Item $ 1-14 TotalCost 15-20
  UnitsOnHand 21-23;
  UnitCost=TotalCost/UnitsOnHand;
datalines;
  Hammers          440      55
  Nylon cord       35       0
  Ceiling fans     1155     30
;
proc print data=inventory;
  format TotalCost dollar8.2 UnitCost dollar8.2;
run;
```

**Example Code 24.5** SAS Log: Execution-Time Error (division by 0)
SAS executes the entire step, assigns a missing value for the variable UnitCost in the output, and writes the following to the SAS log:

- a note that describes the error
- the values that are stored in the input buffer
- the contents of the program data vector when the error occurred
- a note explaining the error

Note that the values that are listed in the program data vector include the _N_ and _ERROR_ automatic variables. These automatic variables are assigned temporarily to each observation and are not stored with the data set.

In the following example of an execution-time error, the program processes an array and SAS encounters a value of the array's subscript that is out of range. SAS prints an error message to the log and stops processing.

```sas
data test;
  array all{*} x1-x3;
  input I measure;
  if measure > 0 then
    all{I} = measure;
datalines;
1 1.5
. 3
2 4.5
;
proc print data=test;
run;
```
Example Code 24.6  SAS Log: Execution-Time Error (Subscript Out of Range)

```sas
163 data test;
164   array all{*} x1-x3;
165   input I measure;
166   if measure > 0 then
167     all{I} = measure;
168   datalines;
169   ERROR: Array subscript out of range at line 167 column 7.
170   RULE: ------+----1----+----2----+----3----+----4----+----5---
171   3
172   x1=.  x2=.  x3=.  I=.  measure=3  _ERROR_=1  _N_=2
173  NOTE: The SAS System stopped processing this step because of errors.
174  WARNING: The data set WORK.TEST may be incomplete. When this step was stopped there were 1 observations and 5 variables.
175  WARNING: Data set WORK.TEST was not replaced because this step was stopped.
176  NOTE: DATA statement used (Total process time):
177     real time           0.00 seconds
178     cpu time            0.00 seconds
```

```
179 ;
180 proc print data=test;
181 run;
182 NOTE: No variables in data set WORK.TEST.
183 NOTE: PROCEDURE PRINT used (Total process time):
184     real time           0.00 seconds
185     cpu time            0.00 seconds
```

186 proc printto; run;

Data Errors

**Definition**

Data errors occur when some data values are not appropriate for the SAS statements that you have specified in the program. For example, if you define a variable as numeric, but the data value is actually character, SAS generates a data error. SAS detects data errors during program execution and continues to execute the program, and does the following:

- writes an invalid data note to the SAS log.
- prints the input line and column numbers that contain the invalid value in the SAS log. Unprintable characters appear in hexadecimal. To help determine column numbers, SAS prints a rule line above the input line.
- prints the observation under the rule line.
- sets the automatic variable _ERROR_ to 1 for the current observation.

In this example, a character value in the Number variable results in a data error during program execution:

```
data age;
```
input Name $ Number;
datalines;
Sue 35
Joe xx
Steve 22
;
proc print data=age;
run;

The SAS log shows that there is an error in line 8, position 5–6 of the program.

**Example Code 24.7  SAS Log: Data Error**

```sas
234 data age;
235     input Name $ Number;
236     datalines;
238        Joe xx
Name=Joe Number=. _ERROR_=1 _N_=2
NOTE: Invalid data for Number in line 238 5-6.
RULE:      ----+----1----+----2----+----3----+----4----+----5---
238        Joe xx
Name=Joe Number=. _ERROR_=1 _N_=2
NOTE: The data set WORK.AGE has 3 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
240 ;
241
242 proc print data=age;
243 run;

NOTE: Writing HTML Body file: sashtml2.htm
NOTE: There were 3 observations read from the data set WORK.AGE.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.04 seconds
      cpu time            0.00 seconds
```

**Output 24.2  SAS Output: Data Error**

You can also use the INVALIDDATA= system option to assign a value to a variable when your program encounters invalid data. For more information, see the INVALIDDATA= system option in *SAS System Options: Reference*.

**Format Modifiers for Error Reporting**

The INPUT statement uses the ? and the ?? format modifiers for error reporting. The format modifiers control the amount of information that is written to the SAS log. Both the ? and the ?? modifiers suppress the invalid data message. However,
the `??` modifier also sets the automatic variable `_ERROR_` to 0. For example, these two sets of statements are equivalent:

- `input x ?? 10-12;`
- `input x ? 10-12; _error_=0;`

In either case, SAS sets the invalid values of X to missing values.

Macro-related Errors

Several types of macro-related errors exist:

- macro compile time and macro execution-time errors, generated when you use the macro facility itself
- errors in the SAS code produced by the macro facility

For more information about macros, see *SAS Macro Language: Reference*.

Error Processing in SAS

Syntax Check Mode

Overview of Syntax Check Mode

Syntax check mode enables SAS to flag syntax errors or stop processing when it encounters a DATA step that has a syntax error. SAS can enter syntax check mode only if your program creates a data set. For example, if you use the DATA _NULL_ statement, then SAS cannot enter syntax check mode because it does not create a data set.

In syntax check mode, SAS internally sets the OBS= option to 0 and the REPLACE/NOREPLACE option to NOREPLACE. When these options are in effect, SAS acts as follows:

- reads the remaining statements in the DATA step or PROC step
- checks that statements are valid SAS statements
- executes global statements
- writes errors to the SAS log
- creates the descriptor portion of any output data sets that are specified in program statements
- does not write any observations to new data sets that SAS creates
- does not execute most of the subsequent DATA steps or procedures in the program (exceptions include PROC DATASETS and PROC CONTENTS)
Note: Any data sets that are created after SAS has entered syntax check mode do not replace existing data sets with the same name.

When syntax checking is enabled, SAS underlines the point where it detects a syntax or semantic error in a DATA step and identifies the error by number. SAS then enters syntax check mode and remains in this mode until the program finishes executing. When SAS enters syntax check mode, all DATA step statements and PROC step statements are validated.

Enabling Syntax Check Mode

Use these methods to enable SAS to enter syntax check mode.

- Set the "DMSSYNCHK System Option" in SAS System Options: Reference system option in the windowing environment.
- Set the "SYNTAXCHECK System Option" in SAS System Options: Reference system option in batch or non-interactive mode.

Note: SYNTAXCHECK is the default value in batch or non-interactive mode.

To disable syntax check mode, use the NOSYNTAXCHECK and NODMSSYNCHK system options.

You can place an OPTIONS statement that contains the SYNTAXCHECK system option or the DMSSYNCHK system option before the step that you want to apply it to. If you place the OPTIONS statement inside a step, then SYNTAXCHECK or DMSSYNCHK does not take effect until the beginning of the next step.

Processing Multiple Errors

Depending on the type and severity of the error, the method that you use to run SAS, and your operating environment, SAS either stops program processing or flags errors and continues processing. SAS continues to check individual statements in procedures after it finds certain types of errors. In some cases SAS can detect multiple errors in a single statement and might issue more error messages for a given situation. This is likely to occur if the statement containing the error creates an output SAS data set.

The following example illustrates a statement with two errors:

```sas
data temporary;
   Item1=4;
run;

proc print data=temporary;
   var Item1 Item2 Item3;
run;
```
SAS Log: Multiple Program Errors

Example Code 24.8

```sas
273  data temporary;
274     Item1=4;
275  run;
NOTE: The data set WORK.TEMPORARY has 1 observations and 1
     variables.
NOTE: DATA statement used (Total process time):
     real time           0.01 seconds
     cpu time            0.01 seconds
276
277  proc print data=temporary;
ERROR: Variable ITEM2 not found.
ERROR: Variable ITEM3 not found.
278     var Item1 Item2 Item3;
279  run;
NOTE: The SAS System stopped processing this step because of
     errors.
NOTE: PROCEDURE PRINT used (Total process time):
     real time           0.52 seconds
     cpu time            0.00 seconds
280  proc printto; run;
```

SAS displays two error messages, one for the variable Item2 and one for the variable Item3.

When you are running debugged production programs that are unlikely to encounter errors, you might want to force SAS to abend after a single error occurs. You can use the ERRORABEND system option to do this.

Checkpoint Mode and Restart Mode

Overview of Checkpoint Mode and Restart Mode

When used together, checkpoint mode and restart mode create an environment where batch programs that terminate before completing can be resubmitted without rerunning steps or labeled code sections that have already completed. Execution resumes with either the DATA or PROC step or the labeled code section that was executing when the failure occurred.

A labeled code section is the SAS code that begins with `label:` outside of a DATA or PROC step and ends with the RUN statement that precedes the next `label:` that is outside of a DATA or PROC step. Labels must be unique. Consider using labeled code sections when you want to group DATA or PROC steps that might need to be grouped together because the data for one is dependent on the other.

The following example program has two labeled code sections. The first labeled code section begins with the label `readSortData:` and ends with the `run;` statement for `proc sort data=mylib.mydata;`. The second labeled code section starts with the label `report:` and ends with the `run;` statements for `proc report data=mylib.mydata;`.

```sas
readSortData:
data mylib.mydata;
...more sas code...
run;
```
proc sort data=mylib.mydata;
...more sas code...
run;

report:
proc report data=mylib.mydata;
...more sas code...
run;
endReadSortReport:

Note: The use of label: in checkpoint mode and restart mode is valid only outside of a DATA or PROC statement. Checkpoint mode and restart mode for labeled code sections are not valid for labels within a DATA step or macros.

Checkpoint mode and restart mode can be enabled for either DATA and PROC steps or for labeled code sections, but not both simultaneously. To use checkpoint mode and restart mode on a step-by-step basis, use the step checkpoint mode and the step restart mode. To use checkpoint mode and restart mode based on groups of code sections, use the label checkpoint mode and the label restart mode. Each group of code is identified by a unique label. If you use labels, all steps in a SAS program must belong to a labeled code section.

When checkpoint mode is enabled, SAS records information about DATA and PROC steps or labeled code sections in a checkpoint library. When a batch program terminates prematurely, you can resubmit the program in restart mode to complete execution. In restart mode, global statements are re-executed, macro definitions are recompiled, and macros are re-executed. SAS reads the data in the checkpoint library to determine which steps or labeled code sections completed. Program execution resumes with the step or the label that was executing when the failure occurred.

The checkpoint-restart data contains information only about the DATA and PROC steps or the labeled code sections that completed and the step or labeled code sections that did not complete. The checkpoint-restart data does not contain the following information:
- information about macro variables and macro definitions
- information about SAS data sets
- information that might have been processed in the step or labeled code section that did not complete

Note: Checkpoint mode is not valid for batch programs that contain the DM statement to submit commands to SAS. If checkpoint mode is enabled and SAS encounters a DM statement, checkpoint mode is disabled and the checkpoint catalog entry is deleted.

As a best practice, if you use labeled code sections, add a label at the end of your program. When the program completes successfully, the label is recorded in the checkpoint-restart data. If the program is submitted again in restart mode, SAS knows that the program has already completed successfully.

If a DATA or PROC step must be re-executed, you can add the global statement CHECKPOINT EXECUTE_ALWAYS immediately before the step. This statement tells SAS to always execute the following step without considering the checkpoint-restart data. It is applicable only to the step that follows the statement. For more
You enable checkpoint mode and restart mode for DATA and PROC steps by using system options when you start the batch program in SAS:

- **STEPCHKPT** system option enables checkpoint mode, which indicates to SAS to record checkpoint-restart data
- **STEPCHKPTLIB** system option identifies a user-specified checkpoint-restart library
- **STEPRESTART** system option enables restart mode, ensuring that execution resumes with the DATA or PROC step indicated by the checkpoint-restart library.

You enable checkpoint mode and the restart mode for labeled code sections by using these system options when you start the batch program in SAS:

- **LABELCHKPT** system option enables checkpoint mode for labeled code sections, which indicates to SAS to record checkpoint-restart data.
- **LABELCHKPTLIB** system option identifies a user-specified checkpoint-restart library
- **LABELRESTART** system option enables restart mode, ensuring that execution resumes with the labeled code section indicated by the checkpoint-restart library.

If you use the Work library as your checkpoint-restart library, you can use the chkptclean system option to have the files in the Work library erased after a successful execution of your batch program.

For information, see the following system options in *SAS System Options: Reference*:

- “STEPCHKPT System Option” in *SAS System Options: Reference*
- “STEPCHKPTLIB= System Option” in *SAS System Options: Reference*
- “STEPRESTART System Option” in *SAS System Options: Reference*
- “LABELCHKPT System Option” in *SAS System Options: Reference*
- “LABELCHKPTLIB= System Option” in *SAS System Options: Reference*
- “LABELRESTART System Option” in *SAS System Options: Reference*
- “CHKPTCLEAN System Option” in *SAS System Options: Reference*

**Requirements for Using Checkpoint Mode and Restart Mode**

In order for checkpoint mode and restart mode to work successfully, the number and order of the DATA and PROC steps or labeled code sections in the batch program must not change between SAS invocations. By specifying the ERRORABEND and ERRORCHECK system options when SAS starts, SAS terminates for most error conditions in order to maintain valid checkpoint-restart data.

The checkpoint-restart library can be a user-specified library or, if no library is specified, the checkpoint-restart data is saved to the Work library. Always start SAS with the NOWORKTERM and NOWORKINIT system options regardless of whether the checkpoint-restart data is saved to a user-specified library or to the Work library. SAS writes the name of the Work library to the SAS log.

**Operating Environment Information:** Under UNIX and z/OS operating environments, consider always assigning a checkpoint-restart library when you use...
the STEPCHKPT option or the LABELCHKPT option. If your site sets the CLEANWORK utility to run at regular intervals, data in the Work library might be lost. Under z/OS, it might not be practical for your site to reuse the Work library in a batch session.

The labels for labeled code sections must be unique. If SAS enters restart mode for a label that is a duplicate label, SAS starts at the first label. The code between the duplicate labels might rerun needlessly.

Setting Up and Executing Checkpoint Mode and Restart Mode

To set up checkpoint mode and restart mode, make the following modifications to your batch program:

- Add the CHECKPOINT EXECUTE_ALWAYS statement before any DATA and PROC steps that you want to execute each time the batch program is submitted.
- If your checkpoint-restart library is a user-defined library, you must add the LIBNAME statement that defines the checkpoint-restart libref as the first statement in the batch program. If you use the Work library as your checkpoint library, no LIBNAME statement is necessary.

Once the batch program has been modified, you start the program using the appropriate system options:

- For checkpoint-restart data that is saved in the Work library, start a batch SAS session that specifies these system options:
  - SYSIN, if required in your operating environment, names the batch program.
  - STEPCHKPT or LABELCHKPT enables checkpoint mode.
  - NOWORKTERM saves the Work library when SAS ends.
  - NOWORKINIT does not initialize the Work library when SAS starts.
  - ERRORCHECK STRICT puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.
  - ERRORABEND specifies whether SAS terminates for most errors.
  - CHKPTCLEAN specifies whether to erase files in the Work library and delete the Work library if the batch program runs successfully.

In the Windows operating environment, the following SAS command starts a batch program in checkpoint mode using the Work library as the checkpoint-restart library:

```
sas -sysin 'c:\mysas\myprogram.sas'-stepchkpt -noworkterm -noworkinit -errorcheck strict -errorabend -chkptclean
```

- For checkpoint-restart data that is saved in a user-specified library, start a batch SAS session that includes these system options:
  - SYSIN, if required in your operating environment, names the batch program.
  - STEPCHKPT or LABELCHKPT enables checkpoint mode.
  - STEPCHKPTLIB or LABELCHKPTLIB specifies the libref of the library where SAS saves the checkpoint-restart data.
  - NOWORKTERM saves the Work library when SAS ends.
  - NOWORKINIT does not initialize the Work library when SAS starts.
ERRORCHECK STRICT puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.

ERRORABEND specifies whether SAS terminates for most errors.

In the Windows operating environment, the following SAS command starts a batch program in checkpoint mode using a user-specified checkpoint-restart library:

```
sas -sysin 'c:\mysas\myprogram.sas' -labelchkpt -labelchkptlib mylibref
     -noworkterm -noworkinit -errorcheck strict -errorabend
```

In this case, the first statement in MyProgram.sas is the LIBNAME statement that defines the MyLibref libref.

### Restarting Batch Programs

To resubmit a batch SAS session using the checkpoint-restart data that is saved in the Work library, include these system options when SAS starts:

- **SYSIN**, if required in your operating environment, names the batch program.
- **STEPCHKPT** or **LABELCHKPT** continues checkpoint mode.
- **STEPRESTART** or **LABELRESTART** enables restart mode, indicating to SAS to use the checkpoint-restart data.
- **NOWORKINIT** starts SAS using the Work library from the previous SAS session.
- **NOWORKTERM** saves the Work library when SAS ends.
- **ERRORCHECK STRICT** puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.
- **ERRORABEND** specifies whether SAS terminates for most errors.
- **CHKPTCLEAN** specifies whether to erase files in the Work library if the batch program runs successfully.

In the Windows operating environment, the following SAS command resubmits a batch program whose checkpoint-restart data was saved to the Work library:

```
sas -sysin 'c:\mysas\mysasprogram.sas' -stepchkpt -steprestart -noworkinit
     -noworkterm -errorcheck strict -errorabend -chkptclean
```

By specifying the **NOWORKTERM** system options and either the **STEPCHKPT** or **LABELCHKPT** system option, checkpoint mode continues to be enabled once the batch program restarts.

To resubmit a batch SAS session using the checkpoint-restart data that is saved in a user-specified library, include these system options when SAS starts:

- **SYSIN**, if required in your operating environment, names the batch program.
- **STEPCHKPT** or **LABELCHKPT** continues checkpoint mode.
- **STEPRESTART** or **LABELRESTART** enables restart mode, indicating to SAS to use the checkpoint-restart data.
- **STEPCHKPTLIB** or **LABELCHKPTLIB** specifies the libref of the checkpoint-restart library.
- **NOWORKTERM** saves the Work library when SAS ends.
- **NOWORKINIT** does not initialize the Work library when SAS starts.
ERRORCHECK STRICT puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.

ERRORABEND specifies whether SAS terminates for most errors.

In the Windows operating environment, the following SAS command resubmits a batch program whose checkpoint-restart data was saved to a user-specified library:

```
sas -sysin 'c:\mysas\mysasprogram.sas' -labelchkpt -labelrestart -labelchklib -noworkterm -noworkinit mylibref -errorcheck strict -errorabend
```

### Using System Options to Control Error Handling

You can use the following system options to control error handling (resolve errors) in your program:

- **BYERR**
  - Specifies whether SAS produces errors when the SORT procedure attempts to process a _NULL_ data set.

- **CHKPTCLEAN**
  - In checkpoint mode or reset mode, specifies whether to erase files in the Work directory if a batch program executes successfully.

- **DKRICOND=**
  - Specifies the level of error detection to report when a variable is missing from an input data set during the processing of a DROP=, KEEP=, and RENAME= data set option.

- **DKROCOND=**
  - Specifies the level of error detection to report when a variable is missing from an output data set during the processing of a DROP=, KEEP=, and RENAME= data set option.

- **DSNFERR**
  - When a SAS data set cannot be found, specifies whether SAS issues an error message.

- **ERRORABEND**
  - Specifies whether SAS responds to errors by terminating.

- **ERRORCHECK=**
  - Specifies whether SAS enters syntax-check mode when errors are found in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.

- **ERRORS=**
  - Specifies the maximum number of observations for which SAS issues complete error messages.

- **FMTERR**
  - When a variable format cannot be found, specifies whether SAS generates an error or continues processing.

- **INVALIDDATA=**
  - Specifies the value that SAS assigns to a variable when invalid numeric data is encountered.

- **LABELCHKPT**
  - Specifies whether SAS checkpoint-restart data is to be recorded for a batch program that contains labeled code sections.
LABELCHKPTLIB
  specifies the libref of the library where checkpoint-restart data is saved for
  labeled code sections.

LABELRESTART
  specifies whether to execute a batch program by using checkpoint-restart data
  for labeled code sections.

MERROR
  specifies whether SAS issues a warning message when a macro-like name does not match a macro keyword.

QUOTELENMAX
  if a quoted string exceeds the maximum length allowed, specifies whether SAS writes a warning message to the SAS log.

SERROR
  specifies whether SAS issues a warning message when a macro variable reference does not match a macro variable.

STEPCHKPT
  specifies whether checkpoint-restart data is to be recorded for a batch program.

STEPCHKPTLIB=
  specifies the libref of the library where checkpoint-restart data is saved.

STEPRESTART
  specifies whether to execute a batch program by using checkpoint-restart data.

VARINITCHK=
  specifies whether to stop or continue processing a DATA step when a variable is not initialized. You can also specify the type of message that is written to the SAS log.

VNFERR
  specifies whether SAS issues an error or warning when a BY variable exists in one data set but not another data set. SAS issues only these errors or warnings when processing the SET, MERGE, UPDATE, or MODIFY statements.

For more information about SAS system options, see SAS System Options: Reference.

Using Return Codes

In some operating environments, SAS passes a return code to the system, but the way in which return codes are accessed is specific to your operating environment. See Return Codes and Completion Status in SAS 9.4 Companion for Windows for more information.

Operating Environment Information: For more information about return codes, see the SAS documentation for your operating environment.

Other Error-Checking Options

To help determine your programming errors, you can use the following methods:
Error Checking When Using Indexes to Randomly Access or Update Data

The Importance of Error Checking

When reading observations with the SET statement and KEY= option or with the MODIFY statement, error checking is imperative for several reasons. The most important reason is that these tools use nonsequential access methods. Therefore, there is no guarantee that an observation will be located that satisfies the request. Error checking enables you to direct execution to specific code paths, depending on the outcome of the I/O operation. Your program continues execution for expected conditions and terminate execution when unexpected results occur.

Error-Checking Tools

Two tools have been created to make error checking easier when you use the MODIFY statement or the SET statement with the KEY= option to process SAS data sets:

- _IORC_ automatic variable
SYSRC autocall macro

_IORC_ is created automatically when you use the MODIFY statement or the SET statement with KEY=. The value of _IORC_ is a numeric return code that indicates the status of the I/O operation from the most recently executed MODIFY or SET statement with KEY=. Checking the value of this variable enables you to detect abnormal I/O conditions and to direct execution down specific code paths instead of having the application terminate abnormally. For example, if the KEY= variable value does match between two observations, you might want to combine them and output an observation. If they do not match, however, you might want to only write a note to the log.

Because the values of the _IORC_ automatic variable are internal and subject to change, the SYSRC macro was created to enable you to test for specific I/O conditions while protecting your code from future changes in _IORC_ values. When you use SYSRC, you can check the value of _IORC_ by specifying one of the mnemonics listed in the following table.

### Table 24.2 Most Common Mnemonic Values of _IORC_ for DATA Step Processing

<table>
<thead>
<tr>
<th>Mnemonic Value</th>
<th>Meaning of Return Code</th>
<th>When Return Code Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>_DSENMR</td>
<td>The Transaction data set observation does not exist in the Master data set.</td>
<td>MODIFY with BY is used and no match occurs.</td>
</tr>
<tr>
<td>_DSEMTR</td>
<td>Multiple Transaction data set observations with the same BY variable value do not exist in the Master data set.</td>
<td>MODIFY with BY is used and consecutive observations with the same BY values do not find a match in the first data set. In this situation, the first observation that fails to find a match returns _DSENMR. The subsequent observations return _DSEMTR.</td>
</tr>
<tr>
<td>_DSENOM</td>
<td>No matching observation was found in the Master data set.</td>
<td>SET or MODIFY with KEY= finds no match.</td>
</tr>
<tr>
<td>_SENOCHN</td>
<td>The output operation was unsuccessful.</td>
<td>the KEY= option in a MODIFY statement contains duplicate values.</td>
</tr>
<tr>
<td>_SOK</td>
<td>The I/O operation was successful.</td>
<td>a match is found.</td>
</tr>
</tbody>
</table>
Example 1: Routing Execution When an Unexpected Condition Occurs

Overview

This example shows how to prevent an unexpected condition from terminating the DATA step. The goal is to update a master data set with new information from a transaction data set. This application assumes that there are no duplicate values for the common variable in either data set.

Note: This program works as expected only if the master and transaction data sets contain no consecutive observations with the same value for the common variable. For an explanation of the behavior of MODIFY with KEY= when duplicates exist, see the MODIFY statement in *SAS DATA Step Statements: Reference*.

Input Data Sets

The Transaction data set contains three observations: two updates to information in Master and a new observation about PartNumber value 6 that needs to be added. Master is indexed on PartNumber. There are no duplicate values of PartNumber in Master or Transaction. The following shows the Master and the Transaction input data sets:

<table>
<thead>
<tr>
<th>Master</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>PartNumber</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

Original Program

The objective is to update the Master data set with information from the Transaction data set. The program reads Transaction sequentially. Master is read directly, not sequentially, using the MODIFY statement and the KEY= option. Only observations with matching values for PartNumber, which is the KEY= variable, are read from Master.

```
data master; 1
   set transaction; 2
   modify master key=PartNumber; 3
   Quantity = Quantity + AddQuantity; 4
run; 5
```

1 Open the Master data set for update.
2 Read an observation from the Transaction data set.
Match observations from the Master data set based on the values of PartNumber.

Update the information about Quantity by adding the new values from the Transaction data set.

Resulting Log

This program has correctly updated one observation but it stopped when it could not find a match for PartNumber value 6. The following lines are written to the SAS log:

```
ERROR: No matching observation was found in Master data set.
PartNumber=6 AddQuantity=16 Quantity=70 _ERROR_=1
_IORC_=1230015 _N_=2
NOTE: The SAS System stopped processing this step because of errors.
NOTE: The data set WORK.MASTER has been updated. There were 1 observations rewritten, 0 observations added and 0 observations deleted.
```

Resulting Data Set

The Master file was incorrectly updated. The updated master has five observations. One observation was updated correctly, a new one was not added, and a second update was not made. The following shows the incorrectly updated Master data set:

```
Master

OBS  PartNumber  Quantity
  1     1          10
  2     2          20
  3     3          30
  4     4          54
  5     5          50
```

Revised Program

The objective is to apply two updates and one addition to Master. This action prevents the DATA step from stopping when it does not find a match in Master for the PartNumber value 6 in Transaction. By adding error checking, this DATA step is allowed to complete normally and produce a correctly revised version of Master. This program uses the _IORC_ automatic variable and the SYSRC autocall macro in a SELECT group to check the value of the _IORC_ variable. If a match is found, the program executes the appropriate code.

```
data master;
  set transaction;
  modify master key=PartNumber;
  select(_iorc_);
    when(%sysrc(_sok)) do;
      Quantity = Quantity + AddQuantity;
      replace;
    end;
    when(%sysrc(_dsenom)) do;
```
```sas
Quantity = AddQuantity;
_error_ = 0;
output;
end;
otherwise do;
  put 'ERROR: Unexpected value for _IORC_ = ' _iorc_;
  put 'Program terminating. DATA step iteration # ' _n_;
  put _all_;
  stop;
end;
end;
run;
```

1. Open the Master data set for update.
2. Read an observation from the Transaction data set.
3. Match observations from the Master data set based on the value of PartNumber.
4. Take the correct course of action based on whether a matching value for PartNumber is found in Master. Update Quantity by adding the new values from Transaction. The SELECT group directs execution to the correct code. When a match occurs (_SOK), update Quantity and replace the original observation in Master. When there is no match (_DSENOM), set Quantity equal to the AddQuantity amount from Transaction, and append a new observation. _ERROR_ is reset to 0 to prevent an error condition that would write the contents of the program data vector to the SAS log. When an unexpected condition occurs, write messages and the contents of the program data vector to the log, and stop the DATA step.

**Resulting Log**

The DATA step executed without error and observations were appropriately updated and added. The following lines are written to the SAS log:

```
NOTE: The data set WORK.MASTER has been updated. There were
  2 observations rewritten, 1 observations added and 0
  observations deleted.
```

**Correctly Updated Master Data Set**

Master contains updated quantities for PartNumber values 2 and 4 and a new observation for PartNumber value 6. The following shows the correctly updated Master data set:

```
  OBS  PartNumber  Quantity
    1     1        10
    2     2        32
    3     3        30
    4     4        54
    5     5        50
    6     6        16
```
Example 2: Using Error Checking on All Statements That Use KEY=

Overview
This example shows how important it is to use error checking on all statements that use the KEY= option when reading data.

Input Data Sets
The Master and Description data sets are both indexed on PartNumber. The Order data set contains values for all parts in a single order. Only Order contains the PartNumber value 8. The following shows the Master, Order, and Description input data sets:

<table>
<thead>
<tr>
<th>Master</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS  PartNumber</td>
<td>Quantity</td>
</tr>
<tr>
<td>1    1</td>
<td>10</td>
</tr>
<tr>
<td>2    2</td>
<td>20</td>
</tr>
<tr>
<td>3    3</td>
<td>30</td>
</tr>
<tr>
<td>4    4</td>
<td>40</td>
</tr>
<tr>
<td>5    5</td>
<td>50</td>
</tr>
</tbody>
</table>

| OBS  PartNumber |
| 1    2          |
| 2    4          |
| 3    1          |
| 4    3          |
| 5    8          |
| 6    5          |
| 7    6          |

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS  PartNumber</td>
</tr>
<tr>
<td>1    4</td>
</tr>
<tr>
<td>2    3</td>
</tr>
<tr>
<td>3    2</td>
</tr>
<tr>
<td>4    6</td>
</tr>
</tbody>
</table>

Original Program with Logic Error
The objective is to create a data set that contains the description and number in stock for each part in a single order, except for the parts that are not found in either of the two input data sets, Master and Description. A transaction data set contains the part numbers of all parts in a single order. One data set is read to retrieve the description of the part and another is read to retrieve the quantity that is in stock.

The program reads the Order data set sequentially and then uses SET with the KEY= option to read the Master and Description data sets directly. This reading is based on the key value of PartNumber. When a match occurs, an observation that contains all the necessary information for each value of PartNumber in Order is written. This first attempt at a solution uses error checking for only one of the two SET statements that use KEY= to read a data set.
data combine;  
length PartDescription $ 15;  
set order;  
set description key=PartNumber;  
set master key=PartNumber;  
select(_iorc_);  
when(%sysrc(_sok)) do;  
  output;  
end;  
when(%sysrc(_dsenom)) do;  
  PartDescription = 'No description';  
  _error_ = 0;  
  output;  
end;  
otherwise do;  
  put 'ERROR: Unexpected value for _IORC_= ' _iorc_;  
  put 'Program terminating.';  
  put _all_;  
  stop;  
end;  
run;

1 Create the Combine data set.

2 Read an observation from the Order data set. Read an observation from the Description and the Master data sets based on a matching value for PartNumber, the key variable. Note that no error checking occurs after an observation is read from Description.

3 Take the correct course of action, based on whether a matching value for PartNumber is found in Master or Description. (This logic is based on the erroneous assumption that this SELECT group performs error checking for both of the preceding SET statements that contain the KEY= option. It actually performs error checking for only the most recent one.) The SELECT group directs execution to the correct code. When a match occurs (_SOK), the value of PartNumber in the observation that is being read from Master matches the current PartNumber value from Order. So, output an observation. When there is no match (_DSENOM), no observations in Master contain the current value of PartNumber, so set the value of PartDescription appropriately and output an observation. _ERROR_ is reset to 0 to prevent an error condition that would write the contents of the program data vector to the SAS log. When an unexpected condition occurs, write messages and the contents of the program data vector to the log, and stop the DATA step.

Resulting Log

This program creates an output data set but executes with one error. The following lines are written to the SAS log:

PartNumber=1 PartDescription=Nuts Quantity=10 _ERROR_=1 _IORC_=0 _N_=3
PartNumber=5 PartDescription=No description Quantity=50 _ERROR_=1 _IORC_=0 _N_=6

NOTE: The data set WORK.COMBINE has 7 observations and 3 variables.
Resulting Data Set

The following shows the incorrectly created Combine data set. Observation 5 should not be in this data set. PartNumber value 8 does not exist in either Master or Description, so no Quantity should be listed for it. Also, observations 3 and 7 contain descriptions from observations 2 and 6, respectively.

<table>
<thead>
<tr>
<th>OBS</th>
<th>PartNumber</th>
<th>PartDescription</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Screws</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Nuts</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Nuts</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Bolts</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>No description</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>No description</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>No description</td>
<td>50</td>
</tr>
</tbody>
</table>

Revised Program

To create an accurate output data set, this example performs error checking on both SET statements that use the KEY= option:

```sas
data combine(drop=Foundes);  
  length PartDescription $ 15;  
  set order;  
  Foundes = 0;  
  set description key=PartNumber;  
  select(_iorc_);  
    when(%sysrc(_sok)) do;  
      Foundes = 1;  
    end;  
    when(%sysrc(_dsenom)) do;  
      PartDescription = 'No description';  
      _error_ = 0;  
    end;  
  otherwise do;  
    put 'ERROR: Unexpected value for _IORC_= ' _iorc_;  
    put 'Program terminating. Data set accessed is Description';  
    put _all_;  
    _error_ = 0;  
    stop;  
  end;  
end;  
set master key=PartNumber;  
select(_iorc_);  
  when(%sysrc(_sok)) do;  
    output;  
  end;  
  when(%sysrc(_dsenom)) do;  
    if not Foundes then do;  
      _error_ = 0;  
      put 'WARNING: PartNumber ' PartNumber 'is not in'  
        ' Description or Master.';  
    end;  
end;```
else do;
    Quantity = 0;
    _error_ = 0;
    output;
end;
end;
otherwise do;
    put 'ERROR: Unexpected value for _IORC_ = ' _iorc_;
    put 'Program terminating. Data set accessed is Master';
    put _all_;
    _error_ = 0;
    stop;
end;
end; /* ends the SELECT group */
run;

1 Create the Combine data set.

2 Read an observation from the Order data set.

3 Create the variable Foundes so that its value can be used later to indicate when a PartNumber value has a match in the Description data set.

4 Read an observation from the Description data set, using PartNumber as the key variable.

5 Take the correct course of action based on whether a matching value for PartNumber is found in Description. The SELECT group directs execution to the correct code based on the value of _IORC_. When a match occurs (_SOK), the value of PartNumber in the observation that is being read from Description matches the current value from Order. Foundes is set to 1 to indicate that Description contributed to the current observation. When there is no match (_DSENOM), no observations in Description contain the current value of PartNumber, so the description is set appropriately. _ERROR_ is reset to 0 to prevent an error condition that would write the contents of the program data vector to the SAS log. Any other _IORC_ value indicates that an unexpected condition has been met, so messages are written to the log and the DATA step is stopped.

6 Read an observation from the Master data set, using PartNumber as a key variable.

7 Take the correct course of action based on whether a matching value for PartNumber is found in Master. When a match is found (_SOK) between the current PartNumber value from Order and from Master, write an observation. When a match is not found (_DSENOM) in Master, test the value of Foundes. If Foundes is not true, then a value was not found in Description either, so write a message to the log but do not write an observation. If Foundes is true, however, the value is in Description but not Master. So write an observation but set Quantity to 0. Again, if an unexpected condition occurs, write a message and stop the DATA step.

Resulting Log

The DATA step executed without error. Six observations were correctly created and the following message was written to the log:

WARNING: PartNumber 8 is not in Description or Master.
NOTE: The data set WORK.COMBINE has 6 observations
and 3 variables.

Correctly Created Combine Data Set

The following shows the correctly updated Combine data set. Note that Combine does not contain an observation with the PartNumber value 8. This value does not occur in either Master or Description.

<table>
<thead>
<tr>
<th>OBS</th>
<th>PartNumber</th>
<th>PartDescription</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Screws</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Nuts</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>No description</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Bolts</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>No description</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Washers</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples

Use the Debugger Tool to Debug Logic Errors in the DATA Step

Example Code

The following example shows how to invoke the DATA step debugger. The DEBUG option is specified in the DATA statement after the font slash (/).

```sas
data mydata2 / debug;
  set mydata1;
run;
```

Key Ideas

- The DATA step debugger is a set of commands that enables you to debug logic errors in the SAS DATA step.
- To use the DATA step debugger, specify the DEBUG option in the DATA statement.
- You can use the DATA step debugger to debug logic errors in the SAS DATA step.
- The DATA step debugger enables you to issue commands to execute DATA step statements one by one and then pause to display the resulting variables' values in a window.
- The DATA step debugger is not supported for a DATA step that is running in CAS.
Route Execution When an Unexpected Condition Occurs

Example Code

This example shows how to prevent an unexpected condition from terminating the DATA step. The goal is to update a master data set with new information from a transaction data set. This application assumes that there are no duplicate values for the common variable in either data set.

Note: This program works as expected only if the master and transaction data sets contain no consecutive observations with the same value for the common variable. For an explanation of the behavior of MODIFY with KEY= when duplicates exist, see the MODIFY statement in *SAS DATA Step Statements: Reference*.

The Transaction data set contains three observations: two updates to information in Master and a new observation about PartNumber value 6 that needs to be added. Master is indexed on PartNumber. There are no duplicate values of PartNumber in Master or Transaction. The following shows the Master and the Transaction input data sets:

<table>
<thead>
<tr>
<th>Master</th>
<th>Transaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>PartNumber</td>
</tr>
<tr>
<td>-----</td>
<td>------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

The objective is to update the Master data set with information from the Transaction data set. The program reads Transaction sequentially. Master is read directly, not sequentially, using the MODIFY statement and the KEY= option. Only observations with matching values for PartNumber, which is the KEY= variable, are read from Master.

```sas
data master; /* 1 */
  set transaction; /* 2 */
  modify master key=PartNumber; /* 3 */
  Quantity = Quantity + AddQuantity; /* 4 */
run;
```

1 Open the Master data set for update.
Read an observation from the Transaction data set.

Match observations from the Master data set based on the values of PartNumber.

Update the information about Quantity by adding the new values from the Transaction data set.

This program has correctly updated one observation but it stopped when it could not find a match for PartNumber value 6. The following lines are written to the SAS log:

```
ERROR: No matching observation was found in Master data set.
PartNumber=6 AddQuantity=16 Quantity=70 _ERROR_=1
_IORC_=1230015 _N_=2
NOTE: The SAS System stopped processing this step because of errors.
NOTE: The data set WORK.MASTER has been updated. There were 1 observations rewritten, 0 observations added and 0 observations deleted.
```

Resulting Data Set: The Master file was incorrectly updated. The updated master has five observations. One observation was updated correctly, a new one was not added, and a second update was not made. The following shows the incorrectly updated Master data set:

```
Master

OBS      PartNumber  Quantity
 1          1           10
 2          2           20
 3          3           30
 4          4           54
 5          5           50
```

The objective is to apply two updates and one addition to Master. This action prevents the DATA step from stopping when it does not find a match in Master for the PartNumber value 6 in Transaction. By adding error checking, this DATA step is allowed to complete normally and produce a correctly revised version of Master. This program uses the _IORC_ automatic variable and the SYSRC autocall macro in a SELECT group to check the value of the _IORC_ variable. If a match is found, the program executes the appropriate code.

```
data master;                        /* 1 */
  set transaction;                   /* 2 */
  modify master key=PartNumber;      /* 3 */
  select(_iorc_);                     /* 4 */
    when(%sysrc(_sok)) do;
      Quantity = Quantity + AddQuantity;
      replace;
    end;
    when(%sysrc(_dsenom)) do;
      Quantity = AddQuantity;
      _error_ = 0;
      output;
    end;
  otherwise do;
    put 'ERROR: Unexpected value for _IORC= ' _iorc_;
    put 'Program terminating. DATA step iteration # ' _n_;  
    put_all;
  end;
```

Use Error Checking on All Statements That Use KEY=

Example Code

This example shows how to create a numeric variable using an assignment statement. Because the value on the right side of the expression is an integer, the variable type will be implicitly set to a numeric data type <etc.>

```
data example;
  set example
    var1 = 1;
run;
```

Overview

This example shows how important it is to use error checking on all statements that use the KEY= option when reading data.

Input Data Sets

The Master and Description data sets are both indexed on PartNumber. The Order data set contains values for all parts in a single order. Only Order contains the PartNumber value 8. The following shows the Master, Order, and Description input data sets:

<table>
<thead>
<tr>
<th>Master</th>
<th>ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBS</td>
<td>PartNumber</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Original Program with Logic Error

The objective is to create a data set that contains the description and number in stock for each part in a single order, except for the parts that are not found in either of the two input data sets, Master and Description. A transaction data set contains the part numbers of all parts in a single order. One data set is read to retrieve the description of the part and another is read to retrieve the quantity that is in stock.

The program reads the Order data set sequentially and then uses SET with the KEY= option to read the Master and Description data sets directly. This reading is based on the key value of PartNumber. When a match occurs, an observation that contains all the necessary information for each value of PartNumber in Order is written. This first attempt at a solution uses error checking for only one of the two SET statements that use KEY= to read a data set.

```
data combine;                       /* 1 */
  length PartDescription $ 15;
  set order;                       /* 2 */
  set description key=PartNumber;  /* 2 */
  set master key=PartNumber;       /* 2 */
  select(_iorc_);                  /* 3 */
  when(%sysrc(_sok)) do;
    output;
  end;
  when(%sysrc(_dsenom)) do;
    PartDescription = 'No description';
    _error_ = 0;
    output;
  end;
  otherwise do;
    put 'ERROR: Unexpected value for _IORC_= ' _iorc_;
    put 'Program terminating.';
    put _all_;
    stop;
  end;
run;
```

1 Create the Combine data set.

2 Read an observation from the Order data set. Read an observation from the Description and the Master data sets based on a matching value for
PartNumber, the key variable. Note that no error checking occurs after an
observation is read from Description.

Take the correct course of action, based on whether a matching value for
PartNumber is found in Master or Description. (This logic is based on the
erroneous assumption that this SELECT group performs error checking for both
of the preceding SET statements that contain the KEY= option. It actually
performs error checking for only the most recent one.) The SELECT group
directs execution to the correct code. When a match occurs (_SOK), the value of
PartNumber in the observation that is being read from Master matches the
current PartNumber value from Order. So, output an observation. When there is
no match (_DSENOM), no observations in Master contain the current value of
PartNumber, so set the value of PartDescription appropriately and output an
observation. _ERROR_ is reset to 0 to prevent an error condition that would
write the contents of the program data vector to the SAS log. When an
unexpected condition occurs, write messages and the contents of the program
data vector to the log, and stop the DATA step.

Resulting Log

This program creates an output data set but executes with one error. The following
lines are written to the SAS log:

```sas
PartNumber=1 PartDescription=Nuts Quantity=10 _ERROR_=1
_ERROR_=0 _N_=3
PartNumber=5 PartDescription=No description Quantity=50
_ERROR_=1 _IORC_=0 _N_=6

NOTE: The data set WORK.COMBINE has 7 observations and 3 variables.
```

Resulting Data Set

The following shows the incorrectly created Combine data set. Observation 5 should
not be in this data set. PartNumber value 8 does not exist in either Master or
Description, so no Quantity should be listed for it. Also, observations 3 and 7 contain
descriptions from observations 2 and 6, respectively.

<table>
<thead>
<tr>
<th>OBS</th>
<th>PartNumber</th>
<th>PartDescription</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Screws</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Nuts</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Nuts</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Bolts</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>No description</td>
<td>30</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>No description</td>
<td>50</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>No description</td>
<td>50</td>
</tr>
</tbody>
</table>

Revised Program

To create an accurate output data set, this example performs error checking on both
SET statements that use the KEY= option:

```sas
data combine(drop=Foundes); /* 1 */
length PartDescription $ 15;
set order; /* 2 */
```
Foundes = 0;                     /* 3 */
set description key=PartNumber;  /* 4 */
select(_iorc_);                  /* 5 */
  when(%sysrc(_sok)) do;
    Foundes = 1;
  end;
  when(%sysrc(_dsenom)) do;
    PartDescription = 'No description';
    _error_ = 0;
  end;
  otherwise do;
    put 'ERROR: Unexpected value for _IORC_= ' _iorc_;
    put 'Program terminating. Data set accessed is Description';
    put _all_;
    _error_ = 0;
    stop;
  end;
end;
set master key=PartNumber;      /* 6 */
select(_iorc_);                  /* 7 */
  when(%sysrc(_sok)) do;          /*ends the SELECT group*/
    output;
  end;
  when(%sysrc(_dsenom)) do;
    if not Foundes then do;
      _error_ = 0;
      put 'WARNING: PartNumber ' PartNumber 'is not in' 
           ' Description or Master.';
    end;
    else do;
      Quantity = 0;
      _error_ = 0;
      output;
    end;
  end;
  otherwise do;
    put 'ERROR: Unexpected value for _IORC_= ' _iorc_;
    put 'Program terminating. Data set accessed is Master';
    put _all_;
    _error_ = 0;
    stop;
  end;
run;

1  Create the Combine data set.
2  Read an observation from the Order data set.
3  Create the variable Foundes so that its value can be used later to indicate when
   a PartNumber value has a match in the Description data set.
4  Read an observation from the Description data set, using PartNumber as the key
   variable.
5  Take the correct course of action based on whether a matching value for
   PartNumber is found in Description. The SELECT group directs execution to the
correct code based on the value of _IORC_. When a match occurs (_SOK), the
value of PartNumber in the observation that is being read from Description
matches the current value from Order. Foundes is set to 1 to indicate that Description contributed to the current observation. When there is no match (_DSENOM), no observations in Description contain the current value of PartNumber, so the description is set appropriately. _ERROR_ is reset to 0 to prevent an error condition that would write the contents of the program data vector to the SAS log. Any other _IORC_ value indicates that an unexpected condition has been met, so messages are written to the log and the DATA step is stopped.

6 Read an observation from the Master data set, using PartNumber as a key variable.

7 Take the correct course of action based on whether a matching value for PartNumber is found in Master. When a match is found (_SOK) between the current PartNumber value from Order and from Master, write an observation. When a match is not found (_DSENOM) in Master, test the value of Foundes. If Foundes is not true, then a value was not found in Description either, so write a message to the log but do not write an observation. If Foundes is true, however, the value is in Description but not Master. So write an observation but set Quantity to 0. Again, if an unexpected condition occurs, write a message and stop the DATA step.

Resulting Log

The DATA step executed without error. Six observations were correctly created and the following message was written to the log:

WARNING: PartNumber 8 is not in Description or Master.
NOTE: The data set WORK.COMBINE has 6 observations and 3 variables.

Correctly Created Combine Data Set

The following shows the correctly updated Combine data set. Note that Combine does not contain an observation with the PartNumber value 8. This value does not occur in either Master or Description.

<table>
<thead>
<tr>
<th>OBS</th>
<th>PartNumber</th>
<th>PartDescription</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>Screws</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Nuts</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>No description</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>Bolts</td>
<td>30</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>No description</td>
<td>50</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Washers</td>
<td>0</td>
</tr>
</tbody>
</table>
# Optimizing System Performance

**Definitions for Optimizing System Performance**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collecting and Interpreting Performance Statistics</td>
<td>555</td>
</tr>
<tr>
<td>Using the FULLSTIMER and STIMER System Options</td>
<td>556</td>
</tr>
<tr>
<td>Interpreting FULLSTIMER and STIMER Statistics</td>
<td>557</td>
</tr>
<tr>
<td>Techniques for Optimizing I/O</td>
<td>557</td>
</tr>
<tr>
<td>Overview of Techniques for Optimizing I/O</td>
<td>557</td>
</tr>
<tr>
<td>Using WHERE Processing</td>
<td>558</td>
</tr>
<tr>
<td>Using DROP and KEEP Statements</td>
<td>559</td>
</tr>
<tr>
<td>Using LENGTH Statements</td>
<td>559</td>
</tr>
<tr>
<td>Using the OBS= and FIRSTOBS= Data Set Options</td>
<td>559</td>
</tr>
<tr>
<td>Creating SAS Data Sets</td>
<td>559</td>
</tr>
<tr>
<td>Using Indexes</td>
<td>560</td>
</tr>
<tr>
<td>Accessing Data through SAS Views</td>
<td>560</td>
</tr>
<tr>
<td>Using Engines Efficiently</td>
<td>560</td>
</tr>
<tr>
<td>Setting System Options to Improve I/O Performance</td>
<td>561</td>
</tr>
<tr>
<td>Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views</td>
<td>563</td>
</tr>
<tr>
<td>Using the SASFILE Statement</td>
<td>563</td>
</tr>
<tr>
<td>Using the DATASETS Procedure to Modify Attributes</td>
<td>564</td>
</tr>
<tr>
<td>Storing Variables as Characters</td>
<td>564</td>
</tr>
<tr>
<td>Techniques for Optimizing Memory Usage</td>
<td>564</td>
</tr>
<tr>
<td>System Options</td>
<td>564</td>
</tr>
<tr>
<td>Using the BY Statement with PROC MEANS</td>
<td>565</td>
</tr>
<tr>
<td>Techniques for Optimizing CPU Performance</td>
<td>565</td>
</tr>
<tr>
<td>Reducing CPU Time By Using More Memory or Reducing I/O</td>
<td>565</td>
</tr>
<tr>
<td>Storing a Compiled Program for Computation-Intensive DATA Steps</td>
<td>565</td>
</tr>
<tr>
<td>Reducing Search Time for SAS Executable Files</td>
<td>566</td>
</tr>
<tr>
<td>Specifying Variable Lengths</td>
<td>566</td>
</tr>
<tr>
<td>Using Parallel Processing</td>
<td>566</td>
</tr>
<tr>
<td>Reducing CPU Time By Modifying Program Compilation Optimization</td>
<td>567</td>
</tr>
<tr>
<td>Calculating Data Set Size</td>
<td>567</td>
</tr>
</tbody>
</table>

## Definitions for Optimizing System Performance

Performance statistics are measurements of the total input and output operations (I/O), memory, and CPU time used to process individual DATA steps or PROC steps. You can obtain...
these statistics by using SAS system options that can help you to measure your job's initial performance and to determine how to improve performance.

System performance is measured by the overall amount of I/O, memory, and CPU time that your system uses to process SAS programs. By using the techniques discussed in the following sections, you can reduce or reallocate your usage of these three critical resources to improve system performance. You might not be able to take advantage of every technique for every situation, but you can choose the ones that are most appropriate for a particular situation.

Collecting and Interpreting Performance Statistics

Using the FULLSTIMER and STIMER System Options

The FULLSTIMER and STIMER system options control the printing of performance statistics in the SAS log. These options produce different results, depending on your operating environment. See the SAS documentation for your operating environment for details about the output that SAS generates for these options.

The following output shows an example of the FULLSTIMER output in the SAS log, as produced in a UNIX operating environment.

Example Code 25.1 Sample Results of Using the FULLSTIMER Option in a UNIX Operating Environment

```
NOTE: DATA statement used:
real time           0.19 seconds
user cpu time       0.06 seconds
system cpu time     0.01 seconds
Memory                            460k
Semaphores    exclusive 194 shared 9 contended 0
SAS Task context switches         1   splits 0
```

The STIMER option reports a subset of the FULLSTIMER statistics. The following example shows STIMER output in the SAS log.

Example Code 25.2 Sample Results of Using the STIMER Option in a UNIX Operating Environment

```
NOTE: DATA statement used:
real time           1.16 seconds
cpu time            0.09 seconds
```

Operating Environment Information: See the documentation for your operating environment for information about how STIMER differs from FULLSTIMER in your operating environment. The information that these options display varies depending
Interpreting FULLSTIMER and STIMER Statistics

Several types of resource usage statistics are reported by the STIMER and FULLSTIMER options, including real time (elapsed time) and CPU time. Real time represents the clock time it took to execute a job or step; it is heavily dependent on the capacity of the system and the current load. As more users share a particular resource, less of that resource is available to you. CPU time represents the actual processing time required by the CPU to execute the job, exclusive of capacity and load factors. If you must wait longer for a resource, your CPU time does not increase, but your real-time increases. It is not advisable to use real time as the only criterion for the efficiency of your program. The reason is that you cannot always control the capacity and load demands on your system. A more accurate assessment of system performance is CPU time, which decreases more predictably as you modify your program to become more efficient.

The statistics reported by FULLSTIMER relate to the three critical computer resources: I/O, memory, and CPU time. Under many circumstances, reducing the use of any of these three resources usually results in better throughput of a particular job and a reduction of real time used. However, there are exceptions, as described in the following sections.

Techniques for Optimizing I/O

Overview of Techniques for Optimizing I/O

I/O is one of the most important factors for optimizing performance. Most SAS jobs consist of repeated cycles of reading a particular set of data to perform various data analysis and data manipulation tasks. To improve the performance of a SAS job, you must reduce the number of times SAS accesses disk or tape devices.

To do this, you can modify your SAS programs to process only the necessary variables and observations by:

- using WHERE processing
- using DROP and KEEP statements
- using LENGTH statements
- using the OBS= and FIRSTOBS= data set options

You can also modify your programs to reduce the number of times it processes the data internally by:

- creating SAS data sets
- using indexes
- accessing data through SAS views
- using engines efficiently
- using PROC DATASETS when modifying variable attributes
- storing numeric values as characters
- using techniques to optimize memory usage

You can reduce the number of data accesses by processing more data each time a device is accessed by:

- setting the `ALIGNSASIOFILES`, `BUFNO=`, `BUFSIZE=`, `CATCACHE=`, `COMPRESS=`, `DATAPAGESIZE=`, `STRIPESIZE=`, `UBUFNO=`, and `UBUFSIZE=` system options
- using the `SASFILE` global statement to open a SAS data set and allocate enough buffers to hold the entire data set in memory

When using SAS DATA step views, you can improve performance by:

- specifying the `VBUFSIZE=` system option
- specifying the `OBSBUF=` data set option

**Note:** Sometimes you might be able to use more than one method, making your SAS job even more efficient.

---

**Using WHERE Processing**

You might be able to use a WHERE statement in a procedure to perform the same task as a DATA step with a subsetting IF statement. The WHERE statement can eliminate extra DATA step processing when performing certain analyses because unneeded observations are not processed.

For example, the following DATA step creates the data set Seatbelt. This data set contains only those observations from the Auto.Survey data set for which the value of Seatbelt is `YES`. The new data set is then printed.

```sas
libname auto 'SAS-library';
data seatbelt;
  set auto.survey;
  if seatbelt='yes';
run;
```

```sas
proc print data=seatbelt;
run;
```

However, you can get the same output from the PROC PRINT step without creating a data set if you use a WHERE statement in the PRINT procedure, as in the following example:

```sas
proc print data=auto.survey;
  where seatbelt='yes';
run;
```

The WHERE statement can save resources by eliminating the number of times that you process the data. In this example, you might be able to use less time and memory by eliminating the DATA step. Also, you use less I/O because there is no
intermediate data set. Note that you cannot use a WHERE statement in a DATA step that reads raw data.

The extent of savings that you can achieve depends on many factors, including the size of the data set. It is recommended that you test your programs to determine the most efficient solution. For more information, see Chapter 20, “WHERE Statement Processing”.

Using DROP and KEEP Statements

Another way to improve efficiency is to use DROP and KEEP statements to reduce the size of your observations. When you create a temporary data set and include only the variables that you need, you can reduce the number of I/O operations that are required to process the data. For more information, see “DROP Statement” in SAS DATA Step Statements: Reference and “KEEP Statement” in SAS DATA Step Statements: Reference.

Using LENGTH Statements

You can also use LENGTH statements to reduce the size of your observations. When you include only the necessary storage space for each variable, you can reduce the number of I/O operations that are required to process the data. Before you change the length of a numeric variable, however, see “LENGTH Statement” in SAS DATA Step Statements: Reference. For more information, see “LENGTH Statement” in SAS DATA Step Statements: Reference.

Using the OBS= and FIRSTOBS= Data Set Options

You can also use the OBS= and FIRSTOBS= data set options to reduce the number of observations processed. When you create a temporary data set and include only the necessary observations, you can reduce the number of I/O operations that are required to process the data. See “FIRSTOBS= Data Set Option” in SAS Data Set Options: Reference and “OBS= Data Set Option” in SAS Data Set Options: Reference for more information.

Creating SAS Data Sets

If you process the same raw data repeatedly, it is usually more efficient to create a SAS data set. SAS can process SAS data sets more efficiently than it can process raw data files.

Another consideration involves whether you are using data sets created with previous releases of SAS. If you frequently process data sets created with previous releases, it is sometimes more efficient to convert that data set to a new one by creating it in the most recent version of SAS. See “Cross-Release Compatibility and Migration” in SAS Language Reference: Concepts for more information.
Using Indexes

An index is an optional file that you can create for a SAS data file to provide direct access to specific observations. The index stores values in ascending value order for a specific variable or variables and includes information as to the location of those values within observations in the data file. In other words, an index enables you to locate an observation by the value of the indexed variable.

Without an index, SAS accesses observations sequentially in the order in which they are stored in the data file. With an index, SAS accesses the observation directly. Therefore, by creating and using an index, you can access an observation faster.

In general, SAS can use an index to improve performance in these situations:

- For WHERE processing, an index can provide faster and more efficient access to a subset of data.
- For BY processing, an index returns observations in the index order, which is in ascending value order, without using the SORT procedure.
- For the SET and MODIFY statements, the KEY= option enables you to specify an index in a DATA step to retrieve particular observations in a data file.

Note: An index exists to improve performance. However, an index conserves some resources at the expense of others. Therefore, you must consider costs associated with creating, using, and maintaining an index. See Chapter 22, “Using Indexes” for more information about indexes and deciding whether to create one.

Accessing Data through SAS Views

You can use the SQL procedure or a DATA step to create SAS views of your data. A SAS view is a stored set of instructions that subsets your data with fewer statements. Also, you can use a SAS view to group data from several data sets without creating a new one, saving both processing time and disk space. For more information, see Chapter 17, “SAS Views” and the Base SAS Procedures Guide.

For information about optimizing system performance with SAS views, see “Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views” on page 563.

Using Engines Efficiently

If you do not specify an engine in a LIBNAME statement, SAS must perform extra processing steps in order to determine which engine to associate with the SAS library. SAS must look at all of the files in the directory until it has enough information to determine which engine to use. For example, the following statement is efficient because it explicitly tells SAS to use a specific engine for the libref Fruits:

```sas
/* Engine specified. */
```
libname fruits v9 'SAS-library';

The following statement does not explicitly specify an engine. In the output, notice the Note about mixed engine types that is generated:

/* Engine not specified. */

libname fruits 'SAS-library';

Example Code 25.3  SAS Log Output from the LIBNAME Statement

NOTE: Directory for library FRUITS contains files of mixed engine types.
NOTE: Libref FRUITS was successfully assigned as follows:
   Engine:       V9
   Physical Name: SAS-library

z/OS Specifics: In the z/OS operating environment, you do not need to specify an engine for certain types of libraries.

See Chapter 13, “SAS Engines” for more information about SAS engines.

Setting System Options to Improve I/O Performance

The following SAS system options can help you reduce the number of disk accesses that are needed for SAS files, though they might increase memory usage and the SAS data set size:

ALIGNSASIOFILES
   A SAS data set consists of a header that is followed by one or more pages of data. Normally, the header is 1K on Windows and 8K on UNIX. The ALIGNSASIOFILES system option forces the header to be the same size as the data pages so that the data pages are aligned to boundaries that allow for more efficient I/O. The page size is set using the BUFSIZE= option.

   For more information, see “ALIGNSASIOFILES System Option” in SAS System Options: Reference and the SAS documentation for your operating environment.

BUFNO=
   SAS uses the BUFNO= option to adjust the number of open page buffers when it processes a SAS data set. Increasing this option's value can improve your application's performance by allowing SAS to read more data with fewer passes; however, your memory usage increases. Experiment with different values for this option to determine the optimal value for your needs.

   Note: You can also use the CBUFNO= system option to control the number of extra page buffers to allocate for each open SAS catalog.

   For more information, see “BUFNO= System Option” in SAS System Options: Reference and the SAS documentation for your operating environment.

BUFSIZE=
   When the BASE engine creates a data set, it uses the BUFSIZE= option to set the permanent page size for the data set. The page size is the amount of data that can be transferred for an I/O operation to one buffer. The default value for BUFSIZE= is determined by your operating environment. Note that the default is
set to optimize the sequential access method. To improve performance for direct (random) access, you should change the value for **BUFSIZE=**.

Whether you use your operating environment's default value or specify a value, the engine always writes complete pages regardless of how full or empty those pages are.

If you know that the total amount of data is going to be small, you can set a small page size with the **BUFSIZE=** option, so that the total data set size remains small and you minimize the amount of wasted space on a page. In contrast, if you know that you are going to have many observations in a data set, you should optimize **BUFSIZE=** so that as little overhead as possible is needed. Note that each page requires some additional overhead.

Large data sets that are accessed sequentially benefit from larger page sizes because sequential access reduces the number of system calls that are required to read the data set. Note that because observations cannot span pages, typically there is unused space on a page.

“Calculating Data Set Size” on page 567 discusses how to estimate data set size.

For more information, see “**BUFSIZE= System Option**” in SAS System Options: Reference and the SAS documentation for your operating environment.

**CATCACHE=**

SAS uses this option to determine the number of SAS catalogs to keep open at one time. Increasing its value can use more memory, although this might be warranted if your application uses catalogs that are needed relatively soon by other applications. (The catalogs closed by the first application are cached and can be accessed more efficiently by subsequent applications.)

For more information, see “**CATCACHE= System Option**” in SAS System Options: Reference and the SAS documentation for your operating environment.

**COMPRESS=**

One further technique that can reduce I/O processing is to store your data as compressed data sets by using the **COMPRESS=** data set option. However, storing your data this way means that more CPU time is needed to decompress the observations as they are made available to SAS. But if your concern is I/O and not CPU usage, compressing your data might improve the I/O performance of your application.

For more information, see “**COMPRESS= System Option**” in SAS System Options: Reference.

**DATAPAGESIZE=**

Beginning with SAS 9.4, the optimal buffer page size is increased to improve I/O performance. The increase in page size might increase the size of the data set or utility file. If you find that the current optimization processes are not ideal for your SAS session, you can use **DATAPAGESIZE=COMPAT93** to use the optimization processes that were used prior to SAS 9.4.

For more information, see “**DATAPAGESIZE= System Option**” in SAS System Options: Reference.

**STRIPESIZE=**

When data is stored in a RAID (Redundant Array of Independent Disks) device, you can use the **STRIPESIZE=** system option to set the I/O buffer size for a directory to be the size of a RAID stripe. SAS data sets or utility files that are created in the directory have a page size that matches the RAID stripe size. Using this option can improve the performance of individual disk.
For more information, see “STIPESIZE= System Option” in SAS System Options: Reference.

UBUFNO=
The UBUFNO= system option sets the number of utility buffers that SAS uses to process data sets.

For more information, see “UBUFNO= System Option” in SAS System Options: Reference.

UBUFSIZE=
The UBUFSIZE= option sets the page size for utility files that SAS uses to process data sets. You can improve the number of disk accesses when values of the UBUFSIZE= option and the BUFSIZE= option are the same.

For more information, see “UBUFSIZE= System Option” in SAS System Options: Reference.

VBUFSIZE=
The VBUFSIZE= option sets the size of the view buffer. View performance can be improved by setting the size of the view buffer large enough to hold more generated observations. For more information, see “VBUFSIZE= System Option” in SAS System Options: Reference and “Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views” on page 563.

Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views

When working with SAS DATA step views, specifying either the OBSBUF= data set option or the VBUSIZE= system option can improve processing efficiency by reducing task switching. The VBUFSIZE= system option enables you to specify the size of the view buffer based on number of bytes. The default buffer size is 65536. The OBSBUF= data set option sets the view buffer size based on a specified number of observations. In either case, setting the view buffer so that it can hold more generated observations speeds up execution time by reducing task switching.

For more information about the VBUFSIZE= system option, see “VBUFSIZE= System Option” in SAS System Options: Reference. For more information about the OBSBUF= data set option, see “OBSBUF= Data Set Option” in SAS Data Set Options: Reference.

Using the SASFILE Statement

The SASFILE global statement opens a SAS data set and allocates enough buffers to hold the entire data set in memory. Once it is read, data is held in memory, available to subsequent DATA and PROC steps, until either a second SASFILE statement closes the file and frees the buffers or the program ends, which automatically closes the file and frees the buffers.

Using the SASFILE statement can improve performance by

- reducing multiple open and close operations (including allocation and freeing of memory for buffers) to process a SAS data set to one open and close operation
- reducing I/O processing by holding the data in memory
If your SAS program consists of steps that read a SAS data set multiple times and you have an adequate amount of memory so that the entire file can be held in real memory, the program should benefit from using the SASFILE statement. Also, SASFILE is especially useful as part of a program that starts a SAS server such as a SAS/SHARE server. For more information about the SASFILE global statement, see the SAS DATA Step Statements: Reference.

Using the DATASETS Procedure to Modify Attributes

Using the DATASETS procedure to modify variable attributes is more efficient than using a DATA step, as long as this task is the only one PROC DATASETS has to perform. The DATASETS procedure processes only the data descriptor information of a data set. A DATA step processes an entire data set. For more information, see “DATASETS Procedure” in Base SAS Procedures Guide.

Storing Variables as Characters

SAS uses eight bytes of storage for each numeric value processed in the DATA step and one byte for each character. If you are not going to perform calculations on a variable that contains numbers, you can save storage by defining the variable as a character variable. When you reduce the amount of storage that is necessary for each variable, you reduce the number of I/O operations.

Techniques for Optimizing Memory Usage

System Options

If memory is a critical resource, several techniques can reduce your dependence on increased memory. However, most of them also increase I/O processing or CPU usage.

You can use the MEMSIZE= system option to increase the amount of memory available to SAS and therefore decrease processing time. By increasing memory, you reduce processing time because the amount of time spent on paging, or reading pages of data into memory, is reduced.

The SORTSIZE= and SUMSIZE= system options enable you to limit the amount of memory that is available to sorting and summarization procedures.

You can also make tradeoffs between memory and other resources, as discussed in "Reducing CPU Time By Modifying Program Compilation Optimization" on page 567. To use the I/O subsystem most effectively, you must use more and larger buffers. However, these buffers share space with the other memory demands of your SAS session.
**Operating Environment Information:** The MEMSIZE= system option is not available in some operating environments. If MEMSIZE= is available in your operating environment, it might not increase memory. See the documentation for your operating environment for more information.

**Using the BY Statement with PROC MEANS**

When you use the CLASS statement and class variables in PROC MEANS, the memory requirements can be substantial. SAS keeps a copy of unique values of each class variable in memory. If PROC MEANS encounters insufficient memory for the summarization of all class variables, you can save memory by using the CLASS and BY statement together to analyze the data by classes.

For more information, see “Comparison of the BY and CLASS Statements” in *Base SAS Procedures Guide*.

**Techniques for Optimizing CPU Performance**

**Reducing CPU Time By Using More Memory or Reducing I/O**

Executing a single stream of code takes approximately the same amount of CPU time each time that code is executed. Optimizing CPU performance in these instances is usually a tradeoff. For example, you can reduce CPU time by using more memory. This allows more information to be read and stored in one operation. However, less memory is available to other processes.

Also, because the CPU performs all the processing that is needed to perform an I/O operation, an option or technique that reduces the number of I/O operations can also have a positive effect on CPU usage.

**Storing a Compiled Program for Computation-Intensive DATA Steps**

Another technique that can improve CPU performance is to store a DATA step that is executed repeatedly as a compiled program rather than as SAS statements. This is especially true for large DATA step jobs that are not I/O-intensive. For more information about storing compiled DATA steps, see “Stored Compiled DATA Step Programs”.
Reducing Search Time for SAS Executable Files

The PATH= system option specifies the list of directories (or libraries, in some operating environments) that contain SAS executable files. Your default configuration file specifies a certain order for these directories. You can rearrange the directory specifications in the PATH= option so that the most commonly accessed directories are listed first. Place the least commonly accessed directories last.

Operating Environment Information: The PATH= system option is not available in some operating environments. See the documentation for your operating environment for more information.

Specifying Variable Lengths

When SAS processes the program data vector, it typically moves the data in one large operation rather than by individual variables. When data is properly aligned (in 8-byte boundaries), data movement can occur in as little as two clock cycles (a single load followed by a single store). SAS moves unaligned data by more complex means, at worst, a single byte at a time. This would be at least eight times slower for an 8-byte variable.

Many high-performance RISC (Reduced Instruction Set Computer) processors pay a very large performance penalty for movement of unaligned data. When possible, leave numeric data at full width (eight bytes). Note that SAS must widen short numeric data for any arithmetic operation. On the other hand, short numeric data can save both memory and I/O. You must determine which method is most advantageous for your operating environment and situation.

Note: Alignment can be especially important when you process a data set by selecting only specific variables or when you use WHERE processing.

Using Parallel Processing

SAS System 9 supports a new wave of SAS functionality related to parallel processing. Parallel processing means that processing is handled by multiple CPUs simultaneously. This technology takes advantage of SMP computers and provides performance gains for two types of SAS processes: threaded I/O and threaded application processing.

For information, see Chapter 26, “Using Parallel Processing”.
Reducing CPU Time By Modifying Program Compilation Optimization

When SAS compiles a program, the code is optimized to remove redundant instructions, missing value checks, and repetitive computations for array subscripts. The code detects patterns of instruction and replaces them with more efficient sequences, and also performs optimizations that pertain to the SAS register. In most cases, performing the code-generation optimization is preferable. If you have a large DATA step program, performing code generation optimization can result in a significant increase in compilation time and overall execution time.

You can reduce or turn off the code generation optimization by using the CGOPTIMIZE= system option. Set the code generation optimization that you want SAS to perform using these CGOPTIMIZE= system option values:

- **0** performs no optimization during code compilation.
- **1** specifies to perform stage 1 optimization. Stage 1 optimization removes redundant instructions, missing value checks, and repetitive computations for array subscripts; detects patterns of instructions and replaces them with more efficient sequences.
- **2** specifies to perform stage 2 optimization. Stage 2 performs optimizations that pertain to the SAS register. Performing stage 2 optimization on large DATA step programs can result in a significant increase in compilation time.
- **3** specifies to perform full optimization, which is a combination of stages 1 and 2. This is the default value.

For more information, see “CGOPTIMIZE= System Option” in SAS System Options: Reference.

Calculating Data Set Size

If you have already applied optimization techniques but still experience lengthy processing times or excessive memory usage, the size of your data sets might be very large. In that case, further improvement might not be possible.

You can estimate the size of a data set by creating a dummy data set that contains the same variables as your data set. Run the CONTENTS procedure, which shows the size of each observation. Multiply the size by the number of observations in your data set to obtain the total number of bytes that must be processed. You can compare processing statistics with smaller data sets to determine whether the performance of the large data sets is in proportion to their size. If not, further optimization might still be possible.

**Note:** When you use this technique to calculate the size of a data set, you obtain only an estimate. Internal requirements, such as the storage of variable names, might cause the actual data set size to be slightly different.
Overview

SAS introduced *threading* technology starting in SAS 9 with the introduction of several Base SAS procedures that had been enhanced to execute, in part, in multiple *threads*. SAS has continued to develop and enhance products and components that take advantage of the threaded processing capabilities provided by proprietary internal subsystems. Threading is available on a variety of platforms from a local desktop with multiple CPUs to high-performance platform servers. These high-performance servers include large multi-core symmetric multi-processor (SMP) systems and massively parallel processing (MPP) appliances typically configured as a distributed *cluster*. Many SAS components that execute on these platforms take advantage of threading technology.

With SAS 9.4M5, when you license SAS Viya, you can access SAS Cloud Analytic Services (CAS), a distributed server environment that supports multithreaded, in-memory processing. See "What is SAS Cloud Analytic Services?" for more information.

Previous releases of Base SAS 9.4 support programs written in the SAS DS2 programming language or the SAS Federated SQL language. These languages can
take advantage of threading. Many other SAS products also use threading technology. For example, the SAS High-Performance Analytics procedures, SAS Stored Processes, and SAS Embedded Process either execute or generate code that executes in high-performance distributed computing environments.

What Is Threading Technology in SAS?

Threading technology provides multiple paths of execution within an operating environment. Each path of execution is called a thread, and each thread can handle a program task or data transfer. The result is multiple program tasks and data I/O operations performed at the same time, in parallel. A thread requires a context (like a register set and a program counter), a segment of code to execute, and some amount of memory to use in the process. A threading operating environment might have multiple CPUs but only one core per CPU. Other more high-performance configurations might include multiple CPUs with multiple cores per CPU and even multiple threads per core. In situations in which each CPU might execute only one thread at a time, the CPU’s ability to quickly switch between threads provides near-simultaneous execution.

Threaded execution in SAS software includes one or both of these two general techniques.

- **Threaded I/O** means that data (frequently in very high volume) is delivered to an application in threads so that the application is continually processing, not waiting on data. In Base SAS and SAS/STAT, several procedures take advantage of threaded reads. Also Base SAS includes the SPD engine that reads from a data set that is partitioned to optimize for threaded input to the application. The SAS High-Performance Analytics procedures require very rapid data delivery. They require threaded reads from data distributed across a computing cluster to deliver huge amounts of data to the application (which is also processing on the cluster) and then write the data in parallel to the data storage appliance. SAS 9.4M5 includes access to SAS Viya, which supports distributed, in-memory, multithreaded processing. See “What is SAS Cloud Analytic Services?” in SAS Language Reference: Concepts for more information about SAS Cloud Analytic Services with SAS Viya.

- **Threaded application processing** means that the application itself is structured to perform certain tasks in parallel on multiple-CPU machines. Threaded application processing enables the application to process large amounts of data to be processed more quickly because multiple threads execute on smaller segments of data. Applications can be designed to take advantage of machines with multiple CPUs whether it is a local four-way desktop or a server-class machine. The SAS High-Performance Analytics Server executes on appliances that distribute both the data and copies of the application across the appliance nodes so that the data is co-located with the application processing.

With SAS 9.4 and SAS Analytics 12.1, customers can access a wide variety of products and components that use threading to support ever-increasing amounts of data as well as computationally intensive algorithms and models. Base SAS and Foundation SAS threading technologies support all of these.
How Is Threading Controlled in SAS?

Many SAS components take advantage of threading technologies automatically. Mechanisms within SAS can detect certain environment variables and either use or not use threading depending on the application’s likely performance. Some environment variables and system options can be configured by the administrator. Data set options, where available, can also be specified to affect I/O or application processing in threads. Because many components might be using threads automatically, it is likely that thread usage would continue, even if the specific options to control threading were turned off.

The system options THREADS, NOTHREADS, and CPUCOUNT influence threading throughout SAS where threading is not automatic. Some products have additional options for controlling threading such as DBSLICE in SAS/ACCESS. The system option THREADS is the default in all products so that threading can occur wherever use of threading is possible and performance is improved. NOTHREADS disables threading in Base SAS or SAS clients and in products that execute in a symmetric multi-processor environment. Procedure statement options are provided to override the system options when necessary.

Certain procedures in SAS products such as SAS/STAT, SAS/OR, SAS/ETS, SAS Enterprise Miner, and SAS High-Performance Analytics Server procedures can execute in either SMP mode on a SAS client, or in massively parallel processing mode in the distributed computing environment. In SMP mode, NOTHREADS is honored, if set; if THREADS is set, CPUCOUNT defaults to the number of threads available for processing on the client, but can be adjusted. In MPP mode, threads are always assumed. NOTHREADS is ignored and threading is always enabled (unless you execute from the client SAS session or SAS Enterprise Miner).

However, in MPP mode, NOTHREADS has no effect. In most of these products, thread controls and execution mode are specified in the PERFORMANCE statement. Refer to the SAS System Options: Reference along with the specific SAS product documentation for information about the threading technologies used in that product or component.

Threading in Base SAS

Some threading is automatic in Base SAS. In addition, the THREADS option is the default for all Base SAS components that support threaded reads or threaded application processing.

Base Language

SAS uses threading technology to build indexes on SAS data files. An index can speed performance in SAS Language WHERE processing, BY-group processing, SET and MODIFY statements, and ARRAY processing in a DO loop. The sorting algorithm in Base SAS, which is used in building an index, is thread-enabled by default but can be disabled with NOTHREADS. For more information, see SAS System Options: Reference along with the specific SAS
product documentation for information about the threading technologies used in that product or component.

Thread-Enabled Base SAS Procedures

Certain Base SAS procedures have algorithms that can take advantage of thread-enabled processing. These procedures are thread-enabled to split parts of the procedure algorithm so that it executes some parts of the algorithm in threads. For example, the SORT procedure is thread-enabled so that the sorting takes place in available threads and each thread sorts a part of the data. The procedure then quickly generates the data set in sorted order from the multiple threads. These procedures can also read data in threads.

The number of threads and CPUs available to the procedures is specified by system or procedure options CPUCOUNT and THREADS|NOTHREADS. NOTHREADS specifies not to use threaded processing for running SAS applications that support it. THREADS is the default. When NOTHREADS is in effect, CPUCOUNT is ignored. Base SAS thread-enabled procedures are the following:

- MEANS
- REPORT
- SORT
- SUMMARY
- TABULATE
- SQL

For details, see “Threaded Processing for Base SAS Procedures” in Base SAS Procedures Guide. For details of the thread-enabled SQL procedure, see the SAS SQL Procedure User’s Guide. Details of SAS System Options, see the SAS System Options: Reference.

Some procedures in SAS/STAT software are also thread-enabled and most of them can run in either SMP or MPP mode. In SMP mode, NOTHREADS and CPUCOUNT are honored. In MPP mode, the PERFORMANCE statement provides the options to control threading. These are the thread-enabled SAS/STAT procedures:

- ADAPTIVEREG
- FMM
- GLM
- GLMSELECT
- LOESS
- MIXED
- QUANTLIFE
- QUANTREG
- QUANTSELECT
- ROBUSTREG

See the SAS/STAT Procedures Guide for details for each procedure.

SAS Scalable Performance Data Engine

The SAS Scalable Performance Data Engine, which is included in Base SAS, is engineered to exploit SMP hardware capabilities. The SAS Scalable
Performance Data Engine uses partitioned data sets that are optimized for reading data in threads. The partition size can be configured with the SAS Scalable Performance Data Engine PARTSIZE option. THREADNUM and SPDEMAXTHREADS control threading for optimum threaded reads. The Base SAS NOTHREADS and CPUCOUNT system options have no effect on SPD Engine threaded reads. They remain in effect for the SAS thread-enabled procedures executing on the SPD Engine data set. SPD Engine indexes are also created in threads in parallel automatically without regard to NOTHREADS, if set. You can use SPDEINDEXSORTSIZE= to optimize threaded index creation. The SPD Engine is described in the SAS Scalable Performance Data Engine: Reference.

SAS FedSQL Language
SAS FedSQL is a SAS proprietary SQL implementation based on the ANSI SQL:1999 standard. It provides support for ANSI SQL data types and other ANSI compliance features. The core strength of SAS FedSQL is its ability to execute federated queries across a heterogeneous database environment and return a single result set. FedSQL queries are automatically optimized with multi-threaded algorithms in order to resolve large-scale operations. In addition, FedSQL can execute outside of a SAS session, for example in the SAS Federation Server and SAS Scalable Performance Data Server environments. The NOTHREADS and CPUCOUNT options have no effect on FedSQL processing.

The FedSQL procedure, which submits FedSQL programs for execution, is included. See the SAS FedSQL Language Reference for complete information.

SAS DS2 Programming Language
DS2 is a SAS proprietary programming language that is appropriate for advanced data manipulation and data modeling applications. DS2 is included with Base SAS and intersects with the SAS DATA step but also supports additional data types, ANSI SQL types, programming structure elements, user-defined methods, and packages. The DS2 SET statement accepts embedded FedSQL syntax and the runtime-generated queries can exchange data interactively between DS2 and any supported database. This allows SQL preprocessing of input tables which effectively combines the power of the two languages.

DS2 programs are thread-enabled by using the THREAD statement on a program coded for parallel execution. The NOTHREADS and CPUCOUNT options have no effect. See the SAS 9.4 DS2 Language Reference for details about whether your DATA step programs would benefit from being converted to DS2.

The DS2 procedure, which submits thread-enabled DS2 programs to the SAS Embedded Process for execution is also included. A high-performance version of the DS2 procedure, PROC HPDS2, submits DS2 language statements to the separately licensed High-Performance Analytics Server for processing. See the SAS High-Performance Analytics Server Usage Guide for documentation on this and other high-performance versions of certain SAS procedures.

DS2 can execute outside of a SAS session. For example:
- SAS Federation Server
- SAS Scalable Performance Data Server
- MPP computing environments such as the SAS In-Database Scoring Accelerator, the SAS Embedded Process environment, and SAS High-Performance Analytics Server distributed environment
SAS Logging
The SAS Logging Facility ignores the NOTHREADS and CPUCOUNT options. It handles all incoming logging events in threads. The client identity that is associated with the current thread or task is reported in the log. The logging facility supports many SAS products and components, but it is included with Base SAS. See the SAS Logging: Configuration and Programming Reference.

SAS Code Analyzer
The SAS Code Analyzer (SCAPROC procedure) runs an existing SAS program (executing the program as usual) when generating metadata about the SAS job that are recorded comments. PROC SCAPROC captures information about the job step, I/O information such as file dependencies, and macro symbol usage information from a running SAS job. The output is a SAS program containing comments with the dependencies described in the comments. An application can read this text and create SAS metadata or determine a process flow based on these dependencies. For example, developers for SAS Data Integration Studio can use the information emitted by the SAS Code Analyzer to reverse engineer legacy SAS jobs. It can also be used with SAS Grid Manager. When the saved job is run on the grid, SAS Grid Manager automatically assigns the identified subtasks to a grid node. For more information, see the SCAPROC procedure documentation in the Base SAS Procedures Guide.

SAS/ACCESS Engines
SAS/ACCESS engines are LIBNAME engines that provide Read, Write, and Update access to more than 60 relational and nonrelational databases, PC files, data warehouse appliances, and distributed file systems. These engines are not part of Base SAS but they depend on Base SAS. They are licensed separately or are included in many product bundles such as SAS BI Server or SAS Activity-Based Management. Many bundles offer the customer a choice of two out of the many SAS/ACCESS engines available.

SAS/ACCESS engines enable SAS programs to connect to a DBMS as if it were a SAS data set. This takes advantage of performance-related DBMS features and benefits including bulk load support, temporary table support, and native SQL support with Explicit Pass-Through. If the DBMS is a parallel server, the engine accesses the DBMS data in parallel by using multiple threads to connect to the DBMS server. If your SAS program is executing a thread-enabled SAS procedure with these SAS/ACCESS engines, even greater gains in performance are likely.

In SAS/ACCESS, threaded reads partition the result set across multiple threads. Unlike threaded processing in Base SAS procedures, threaded reads in SAS/ACCESS are not dependent on the number of processors on a machine. Instead, the result set is retrieved on multiple connections between SAS and the DBMS. SAS causes the DBMS to partition the result set by appending a WHERE clause to the SQL statement. When this happens, a single SQL statement becomes multiple SQL statements, one on each thread. The DBMS reads the partitions one per thread also.

The amount of scalability that is provided with the SAS/ACCESS engines depends on the efficiency of parallelization implemented in the DBMS itself. However, SAS/ACCESS engines have options available in the LIBNAME statement that enable tuning of the threaded implementation within the SAS/ACCESS engines. The options that control threaded reads in SAS/ACCESS are DBSLICE,
DBSLICEPARM, THREADS|NOTHREADS, and whether BY, OBS, or KEY options are used in a PROC or DATA step. Refer to the SAS/ACCESS for Relational Databases documentation for more information.

SAS Scalable Performance Data Server


SAS Intelligence Platform

The SAS Intelligence Platform is an infrastructure for creating, managing, and distributing enterprise intelligence. This infrastructure supports SAS solutions for industries such as financial services, life sciences, health care, retail, and manufacturing. The SAS Intelligence Platform is not part of Base SAS but instead it relies on SAS Foundation, which includes Base SAS and these components:

- SAS Management Console for defining metadata
- SAS Business Intelligence Server and SAS Data Integration Server technologies
- SAS Integration Technologies, which provides the following:
  - the SAS servers and supporting services such as SAS Stored Process Servers
  - application messaging
  - SAS BI Web Services
  - publishing framework
  - SAS Foundation Services

The SAS Intelligence Platform Overview discusses individual components and references a wide spectrum of related SAS Intelligence Platform documentation.

These are the SAS servers in the Intelligence Platform:

- SAS Workspace Server
- SAS Stored Process Server
- SAS Pooled Workspace Server
- SAS OLAP Server
- SAS Metadata Server

Each server is initiated with a pool of active threads. These threads are controlled by the server and are used by server processes (for example, handling incoming
requests). If the NOTHREADS and CPUCOUNT options are specified, they are ignored, except during the execution of submitted code that includes a SAS procedure that honors these options.

For the SAS Metadata Server, thread usage is controlled by default settings for the object server parameters (THREADSMAX and THREADSMIN) and for the metadata server configuration option, MACACTIVETHREADS. Administrators can override these settings in order to fine-tune performance. See the SAS Intelligence Platform: System Administrator’s Guide for details and examples. The THREADMAX and THREADMIN object server parameters are rarely used for servers other than the SAS Metadata Server.

In the intelligent platform middle tier (which is an infrastructure for web applications), incoming requests are processed on threads. These threads are defined using the job execution service. The threads are not constrained by the NOTHREADS or CPUCOUNT options. Both the number of job queue threads and number of job execution threads can be specified. Refer to the SAS Intelligence Platform: Middle-Tier Administration Guide.

SAS MP CONNECT is a part of SAS/CONNECT software that is bundled with the intelligence platform. It supports parallel processing by establishing a connection between multiple SAS sessions and enabling each of the sessions to asynchronously execute tasks in parallel. By establishing connections to processes on the same local computer, the application can use network resources to process in parallel and coordinate all the results into the client SAS session. Many SAS processes use multiple processors on an SMP computer, but they can also be executed on multiple remote single or multiprocessor computers on a network. Threads are always assumed to be available.

Some SAS High-Performance Analytics products can execute on the SAS Intelligence Platform if it is configured as an MPP environment. For example, SAS Grid Manager (discussed in the next section) handles workload management for SAS applications that execute in SMP configurations. It can also manage applications that are coded for parallel execution and distributed across the nodes of the SAS High-Performance Analytics Server.

SAS Visual Analytics is a web-based suite of high-performance analytics applications that executes on the SAS Intelligence Platform if it is configured as an MPP environment. This execution environment for SAS Visual Analytics is documented in the SAS Intelligence Platform: Middle-Tier Administration Guide. (SAS Visual Analytics is discussed further in the next section. See “SAS High-Performance Analytics Portfolio of Products” on page 576.)

SAS High-Performance Analytics Portfolio of Products

SAS High-Performance Analytics products are engineered to make high use of threads. These products are not part of Base SAS or SAS Foundation. However, they rely on and extend Base SAS functionality to provide capabilities that support rapid analysis of huge volumes of data.

Some SAS High-Performance Analytics products can work in concert with, or are directly integrated with, other SAS applications and solutions. For example, you can configure SAS Grid Manager to distribute the workload from SAS Enterprise Guide
executing on the SAS Intelligence Platform. The SAS Grid Manager can be used to manage the workload of SAS jobs on the SAS High-Performance Analytic Server running on a DBMS appliance such as EMC Greenplum or Teradata. And SAS Visual Analytics can be used to explore data that is consumed by SAS Enterprise Miner executing in a SAS Grid environment.

SAS High-Performance Analytics technologies include the following:

- SAS Grid Manager
- SAS In-Database
- SAS In-Memory Analytics technologies, which include:
  - SAS High-Performance Analytics Server
  - SAS Visual Analytics
  - SAS High-Performance Risk Management
  - other SAS high-performance products and solutions

SAS Grid Manager

SAS Grid Manager provides scalable workload management and prioritization, high-availability processing, and optimized performance across various SAS processes and services that execute in threads on a grid of configured servers. By coordinating the resources of separate servers into a centrally managed SAS environment, SAS Grid Manager can provide accelerated processing for SAS jobs.

SAS programs, that are grid-enabled to run in multiple independent steps within the overall program flow, execute in threads. The threaded execution typically occurs at the DATA step or procedure boundaries. Because the programs are specifically written to execute in threads, it is assumed that the THREADS option is set and the CPUCOUNT option is greater than one. SAS Grid Manager enables subtasks of individual SAS jobs to run in parallel on different parts of the grid and share a pool of resources. The threaded subtasks can further benefit from threaded processing performed by any of the thread-enabled procedures executed from within the code. Programs that have been analyzed for parallel processing using the SAS Code Analyzer can be run on the grid with SAS Grid Manager. This automatically assigns the identified subtasks to a grid node.

Many SAS products, such as SAS Enterprise Guide, SAS Enterprise Miner, SAS Data Integration Studio, SAS Web Report Studio, SAS Marketing Automation, and SAS Marketing Optimization are integrated with SAS Grid Manager. An option within the application or in the SAS Metadata enables the integration. SAS Data Integration Studio and SAS Enterprise Miner have code generation engines that can recognize opportunities for parallelization and generate the appropriate code to submit to SAS Grid Manager to execute in parallel across the grid.

Other SAS components, such as the SAS Intelligence Platform, use SAS Grid Manager to determine the optimal SAS server for processing by distributing server workloads across multiple computers on a network. SAS Grid Manager divides jobs into separate processes that run in parallel across multiple servers. SAS Grid Manager is documented in *Grid Computing for SAS*.
SAS In-DATABASE Technology

SAS In-Database technology provides faster execution of commands and functions by sending them to execute inside the databases by SAS applications. This process avoids data movement and conversion but also takes advantage of the threading capabilities of the host database.

SAS In-Database runs on a variety of threaded DBMS and hardware or software appliance architectures such as Teradata, Greenplum, Aster Data, Netezza, and DB2. Because the threading is controlled by the database, NOTTHREADS and CPUCOUNT options have no effect.

In-Database processes are divided into multiple parallel tasks within the database, each working with small subsets of the overall data to be processed. As the results of the smaller subsets are derived, they are consolidated and returned to the SAS application. Typically, execution times are much shorter than if the data were transferred to the SAS server.

In-Database technology uses existing database functionality and standard SAS functionality to extend that SAS functionality into the database. For example, Base SAS procedures such as SORT or SUMMARY can be executed within the database environment with the inclusion of simple options to the procedure’s code. These procedures are “translated” into native database functions for execution within the database environment. As an example of extending SAS functionality into the database, scoring code generated by SAS Enterprise Miner can be exported as functions. Once exported, these functions can be executed by either SAS processes or called from third-party or database-specific applications using standard SQL statements.

SAS In-Database technology is described in the SAS In-Database Products: Administrators Guide and SAS In-Database Products: User’s Guide. SAS In-Database components include scoring and analytic accelerators for most of the supported databases.

SAS In-Memory Analytics Technology

SAS In-Memory Analytics technology takes advantage of the large number of threads and high level of memory that is available in some specially configured DBMS appliances such as Teradata and EMC Greenplum and on commodity hardware using Hadoop Distributed File System (HDFS). The Teradata and EMC Greenplum appliances are assembled as computing clusters using specific massively parallel processing (MPP) techniques. With SAS In-Memory Analytics technology, all of the data to be processed is distributed across the cluster and loaded into memory before the analytic procedure begins. This is in distinct contrast to traditional processing where data is loaded in blocks as they are needed. In addition, SAS High-Performance Analytics procedures are engineered to execute on hundreds of threads and each thread is responsible for a small subset of the overall data to be processed. Faster analysis of large data sets results in greater refinement of analytic models.
Several members of the SAS High-Performance Analytics family of products are based on SAS In-Memory Analytics technology, including the SAS High-Performance Analytics Server, SAS Visual Analytics, and SAS High-Performance Risk. For more detail on the array of SAS In-Memory Analytics components, please see the In-Memory Analytics website: Products & Solutions/In-Memory Analytics.

SAS High-Performance Analytics Server
The SAS High-Performance Analytics Server is engineered to run in threads and provides high-performance analytic procedures that focus on predictive model development with computationally intensive calculations. These procedures are drawn from the libraries of SAS/STAT, SAS/QC, and SAS/ETS. The procedures execute in the MPP computing environment provided by EMC Greenplum and Teradata appliances and Hadoop clusters. High-performance MPP configurations typically have a minimum 1.5TB of memory and upward of 192 cores with multiple threads per core.

The SAS High-Performance Analytics procedures are invoked on the requesting SAS client where a Base SAS session is executing. This can be the traditional Display Manager System, SAS Enterprise Guide, or through the SAS High-Performance Data Mining tab in SAS Enterprise Miner. In the MPP environment, the SAS client communicates with the SAS High-Performance Analytics Server nodes where a thin SAS environment executes a copy of the requested SAS procedure or DS2 code. Once completed, the analytic results are returned to the requesting application on the SAS client.

The PERFORMANCE statement in the SAS High-Performance Analytics procedures enables you to specify parameters to control threading and the mode of processing, SMP (client mode) or MPP (distributed mode). In SMP mode (which is a SAS session on the client machine), the CPUCOUNT default is the number of CPUs on the client machine. CPUCOUNT and NOTHREADS options can override the SAS system options. In this environment, if the procedure executes in MPP mode, then the CPUCOUNT option default is that the number of threads is determined by the number of CPUs on the appliance nodes. The NTHREADS option available only in the PERFORMANCE statement throttles the number of threads.

The SAS High-Performance Analytics Server and procedures are documented in the SAS High-Performance Server Administration Guide. The SAS High-Performance Analytics Server relies on the SAS LASR Analytic Server to provide a highly scalable and reliable analytics infrastructure that is optimized for large volumes of data and complex computations.

SAS Visual Analytics
SAS Visual Analytics runs on SAS High-Performance Analytics Server and on a SAS Intelligence Platform if it is configured as an MPP environment. SAS Visual Analytics provides the ability for business users and business analysts to perform a wide variety of tasks using visualizations specifically designed for exploring big data and deriving value. This enables you to go beyond descriptive statistics and use more specialized analytics in a very scalable way. This provides more accurate insights into the future and aids decision making. For example, users can visually explore data, execute analytic correlations on billions of rows of data in just seconds, and visually present results. This helps quickly identify patterns, trends, and relationships in data that were not evident before.

SAS Visual Analytics can be combined with the SAS High-Performance Analytics Server and SAS In-Database to provide a high-performance model lifecycle. For example, use SAS Visual Analytics to explore the data and decide which information to use. Use SAS High-Performance Analytics Server to build...
your models, and then use SAS In-Database to push the models into an appropriate database for scoring.

SAS Visual Analytics requires a dedicated and specialized configuration of blade hardware such as Teradata or EMC Greenplum appliances or Hadoop HDFS configured as an MPP cluster. This environment is always threaded. SAS options CPUCOUNT and NOTHREADS have no effect. Instead, the NTHREADS option in the PERFORMANCE statement provides a way to throttle thread usage. See the SAS Visual Analytics: User’s Guide for product information.

SAS Visual Analytics relies on the SAS LASR Analytic Server to provide a highly scalable analytics infrastructure that is optimized for large volumes of data and complex computations.

SAS LASR Analytic Server
The SAS LASR Analytic Server is an analytic platform that provides a secure environment for concurrent access to data. It loads the data into memory across the computing nodes of a SAS High-Performance Analytics Server. The SAS LASR Analytic Server executes on the SAS High-Performance Analytics Server root node with worker nodes across the appliance that read data into memory in parallel very fast. If the data is not from a co-located data provider, then the data is read from the DBMS appliance or Hadoop cluster and transferred to the root node of the SAS High-Performance Analytics Server. Then, it is loaded into the memory of the worker nodes. The SAS LASR Analytic Server is not influenced by CPUCOUNT or NOTHREADS. Instead, the NTHREADS option in the PERFORMANCE statement throttles thread usage. Refer to the SAS LASR Analytic Server: Administration Guide for details.

For more SAS In-Memory Analytics products, see Products & Solutions/In-Memory Analytics.

SAS High-Performance Analytics Product Integration

The SAS High-Performance Analytics portfolio supports many SAS solutions and products. Some SAS products are integrated with SAS Grid Manager and (to some extent) other high-performance analytics products in order to take advantage of parallel processing.

SAS Enterprise Guide:
SAS Enterprise Guide executes as a SAS client and provides a front end to SAS servers to execute SAS programs. Users create programs that take advantage of the parallel processing capabilities in Base SAS (for example, thread-enabled procedures and stored processes). SAS Enterprise Guide can detect whether SAS Grid Manager is managing the environment to provide workload balancing, resource assignment, and job prioritization. SAS Enterprise Guide can run Process Flow branches in parallel on different grid nodes. It enables parallel execution of tasks on the same server, and you can run tasks at the project or individual level in a SAS Grid Manager environment. See the SAS Enterprise Guide for details.

SAS Data Integration Studio
SAS Data Integration Studio runs on the SAS Data Integration Server and automatically takes advantage of grid computing if SAS Grid Manager is
SAS Data Integration Studio 3.4 was enhanced to automatically generate SAS applications that are enabled to execute on a SAS Grid Manager managed grid. Users can produce grid-enabled SAS applications without any programming knowledge or knowledge of the underlying grid infrastructure.

These SAS applications detect the existence of a SAS Grid Manager environment at run time and distribute the execution accordingly. These grid-enabled applications can be saved as SAS stored processes and subsequently executed by the SAS Intelligence Platform components including SAS Web Report Studio, SAS Information Map Studio, and the SAS Add-In for Microsoft Office. (These applications need SAS BI or SAS Enterprise BI servers, which depend on SAS Foundation).

SAS Data Integration Studio can execute with in-memory products SAS High-Performance Analytics Server and SAS Visual Analytics Server configured with the SAS Intelligence Platform as an MPP environment, which is always threaded. SAS Data Integration Studio provides High-Performance Analytics transformations for SAS LASR Analytic Servers or HDFS. NOTHREADS and CPUCOUNT options have no effect. See the SAS Data Integration Studio: User’s Guide for details. The SAS Data Integration Server is administered as part of the SAS Intelligence Platform.

SAS Risk Dimensions
The iterative workflow in SAS Risk Dimensions is similar to that in SAS Data Integration Studio; they both execute the same analysis over different subsets of the data. This workflow makes them ideal for taking advantage of SAS Grid Manager to distribute the processing across the grid. For more information, see SAS Risk Dimensions User’s Guide.

SAS Enterprise Miner
SAS Enterprise Miner can automatically generate SAS applications that are enabled to execute on a SAS Grid Manager grid. These SAS applications detect the presence of a SAS Grid Manager environment at run time and distribute the execution accordingly. The applications can also be saved as SAS stored processes and subsequently executed by the SAS Business Intelligence components such as SAS Web Report Studio. The DMINE, DMREG, and DMDB procedures are thread-enabled with THREADS as the default. NOTHREADS disables multithreaded computation.

SAS Enterprise Miner includes a key set of high-performance statistical and data mining procedures for tasks such as data binning, imputation, scoring, and transformations, and others. These procedures execute in the highly threaded SAS High-Performance Analytics Server. In MPP mode, SAS Enterprise Miner distributes data, memory, and computation across the server nodes to build predictive models. If the SAS High-Performance Analytics Server is installed and specific Hadoop HPDM code is enabled, an HPDM tab is available in SAS Enterprise Miner that permits execution on the server nodes. Scoring code from Enterprise Miner can be run inside the database using SAS In-Database technology. For more information, see SAS Enterprise Miner: Administration and Configuration.
SAS Viya

With SAS 9.4M5, you can license SAS Viya, software that offers a variety of high performance products and access to SAS Cloud Analytic Services. For more information, see *An Introduction to SAS Viya Programming*. 
PART 5

Creating Output

Chapter 27
Output ......................................................... 585

Chapter 28
Printing ....................................................... 611
Definitions for SAS Output

SAS output is the result of executing SAS programs. Most SAS procedures and some DATA step programs produce output. A SAS program can produce some or all of the following types of output:
destination

A designation within a SAS program that the Output Delivery System (ODS) uses to generate a specific type of output. Or, simply put, it is how and where ODS routes your output. For example, ODS can route your output to a browser as HTML, to a file, or to your terminal or display as a simple list report. For more information, see “Overview of How ODS Works” in SAS Output Delivery System: Procedures Guide. The destination of your output depends on the following:

- your operating environment
- your mode of running SAS
- your version of SAS

program results

Contain the programmatic results from SAS procedures and SAS DATA step applications. These results can be sent to a file or printed as a report. There are a variety of options, formats, statements, and commands available in SAS to customize your output. The Output Delivery System enables you to specify output destinations to control how your output is stored, table definitions to control how your output is structured, and style templates to control the stylistic elements of your output. For more information, see SAS Output Delivery System: User’s Guide.

Here are a few examples of the types of output that you can get from running SAS programs:

- a SAS data set
- an HTML file for web viewing
- a simple listing report
- RTF output suitable for viewing in Microsoft Word
- SVG output suitable for viewing by mobile devices
- an ODS Document for specifying multiple destinations at one time
- output that is formatted for a high-resolution printer such as PostScript and PDF
- output formatted in various markup languages (in addition to HTML)

SAS log output

SAS log contains a description of the SAS session and lists the lines of source code that were executed.

You can write specific information to the SAS log (such as variable values or text strings) by using the SAS statements that are described in “Writing to the Log in All Modes” on page 595.

The log is also used by some of the SAS procedures that perform utility functions, for example, the DATASETS and OPTIONS procedures. See the Base SAS Procedures Guide.

Because the SAS log provides a journal of program processing, it is an essential debugging tool. However, certain system options must be in effect to make the log effective for debugging your SAS programs. “Customizing the Log” on page 597 describes several SAS system options that you can use.
SAS console log created when the regular SAS log is not active, for recording information, warnings, and error messages. When the SAS log is active, the SAS console log is used only for fatal system initialization errors or late termination messages.

Note: For more information about the destination of the SAS console log, see the SAS documentation for your operating environment.

SAS logging facility output contains log messages that you create using the SAS logging facility. The SAS logging facility is optional. Logging facility messages can be created within SAS programs or they can be created by SAS for logging SAS server messages. Logging facility log messages are based on message categories such as authentication, administration, performance, or customized message categories in SAS programs. In SAS programs, you use logging facility functions, autocall macros, or DATA step component objects to create the logging facility environment.

Appenders are defined for the duration of a macro program or a DATA step. Loggers are defined for the duration of the SAS session.

For more information, see SAS Logging: Configuration and Programming Reference.

Default Output Destination

The default destination is dependent on the SAS version and mode of running SAS. The following table shows the default destinations for each method of operation based on SAS version:

Table 27.1 Comparison of Default Destinations for Output

<table>
<thead>
<tr>
<th>SAS Version</th>
<th>Mode of Running SAS</th>
<th>Viewer</th>
<th>ODS Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS 9.3 and later</td>
<td>SAS Windowing Environment</td>
<td>SAS Results Viewer or browser window</td>
<td>HTML</td>
</tr>
<tr>
<td></td>
<td>SAS Studio</td>
<td>Results tab or browser window</td>
<td>HTML5</td>
</tr>
<tr>
<td></td>
<td>Enterprise Guide</td>
<td>Project pane, process flow, or browser window</td>
<td>HTML5</td>
</tr>
<tr>
<td></td>
<td>batch mode</td>
<td>Depends on operating environment</td>
<td></td>
</tr>
</tbody>
</table>

Operating Environment Information: The default destination for SAS output is specific to your operating environment. Your configuration file and registry settings...
also affect where your output is sent. For specific information about the default output destination, see the SAS documentation for your operating environment:

- **UNIX**: “The Default Routings for the SAS Log and Procedure Output in UNIX Environments” in *SAS Companion for UNIX Environments*
- **z/OS**: “Destinations of SAS Output Files” in *SAS Companion for z/OS*
- **Windows**: “Managing SAS Output under Windows” in *SAS Companion for Windows*

For more information about the new defaults and ODS destinations, see the *SAS Output Delivery System: User’s Guide*.

---

**Change the Output Destination**

With SAS, there are many ways to control where your log, procedure, and DATA step output are sent. The method that you use depends on your operating system and the mode in which you are running SAS.

The following list describes some of the commonly used ODS statements and other SAS language elements that are used for routing output.

*Table 27.2  Ways to Change the Output Destination*

<table>
<thead>
<tr>
<th>Method to Use</th>
<th>Output Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINTTO procedure</td>
<td>routes DATA step, log, or procedure output from the system default destinations to the destination that you choose. The PRINTTO procedure defines non-ODS destinations.</td>
</tr>
<tr>
<td>FILENAME statement</td>
<td>associates a fileref with an external file or output device and enables you to specify file and device attributes</td>
</tr>
<tr>
<td>FILE command: Windows</td>
<td>stores the contents of the LOG or OUTPUT windows in files that you specify, when the command is issued from within the windowing environment.</td>
</tr>
<tr>
<td>ODS LISTING statement</td>
<td>opens, manages, or closes the LISTING destination.</td>
</tr>
<tr>
<td>ODS OUTPUT statement</td>
<td>produces a SAS data set from an output object and manages the selection and exclusion lists for the OUTPUT destination.</td>
</tr>
</tbody>
</table>
### Method to Use | Output Result
--- | ---
**ODS DOCUMENT statement** | produces an ODS document that enables you to restructure, navigate, and replay your data in different ways. It also enables you to specify multiple destinations without needing to rerun your analysis or repeat your database query.

**ODS HTML statement** | opens, manages, or closes the HTML destination, which produces HTML 4.0 output that contains embedded style sheets.

**ODS MARKUP statement** | opens, manages, or closes the MARKUP destination, which produces SAS output that is formatted using one of many different markup languages.

**ODS PRINTER statement** | opens, manages, or closes the PDF destination, which produces PDF output, a form of output that is read by Adobe Acrobat and other applications.

**ODS RTF statement** | opens, manages, or closes the RTF destination, which produces output written in Rich Text Format for use with Microsoft Word 2002.

**SAS system options** | redefine the destination of log and output for an entire SAS program. These system options are specified when you invoke SAS. The system options used to route output are the ALTLOG=, ALTPRINT=, LOG=, and PRINT= options.

For conceptual information about global ODS statements, see the following resources:

Operating Environment Information:

For information about changing the default output location for the z/OS and UNIX operating environments, see the following resources:
- UNIX: “Changing the Default Routings in UNIX Environments” in SAS Companion for UNIX Environments
Customize Output

Making Output Descriptive

There are many statements and system options available in SAS that enable you to customize your output. You can add informative titles, footnotes, and labels to customize your output and control how the information is laid out on the page.

The following list describes some of the statements and SAS system options that you can use:

<table>
<thead>
<tr>
<th>SAS Language Element</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTER</td>
<td>NOCENTER system option</td>
</tr>
<tr>
<td>DATE</td>
<td>NODATE system option</td>
</tr>
<tr>
<td>FOOTNOTE statement</td>
<td>prints footnotes at the bottom of each output page. You can also use the FOOTNOTES window for this purpose. You can also use the null footnote statement (footnote; to suppress a FOOTNOTE statement.</td>
</tr>
<tr>
<td>FORMCHAR= system option</td>
<td>specifies the default output formatting characters for some procedures such as CALENDAR, FREQ, REPORT, and TABULATE. This system option affects only the traditional LISTING output.</td>
</tr>
<tr>
<td>FORMDLIM= system option</td>
<td>specifies a character to delimit page breaks in SAS output for the LISTING destination.</td>
</tr>
</tbody>
</table>
### Using ODS to Customize the Style and Structure of Output

ODS does more than just enable you to control output destinations. It also enables you to customize the structure and style of your output. Since ODS uses table and style templates (definitions) to display procedure and DATA step results, you can control these results by creating customized table and style templates. You can also

<table>
<thead>
<tr>
<th>SAS Language Element</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LABEL statement</strong></td>
<td>associates descriptive labels with variables. With most procedure output, SAS writes the label rather than the variable name. The LABEL statement also provides descriptive labels when it is used with certain SAS procedures. See <em>Base SAS Procedures Guide</em> for information about using the LABEL statement with a specific procedure.</td>
</tr>
<tr>
<td><strong>LINESIZE=</strong> and <strong>PAGESIZE=</strong> system options</td>
<td>changes the default number of lines per page (page size) and characters per line (line size) for printed output. The default depends on the method of running SAS and the settings of certain SAS system options. Specify new page and line sizes in the OPTIONS statement or OPTIONS window. You can also specify line and page size for DATA step output in the FILE statement. The values that you use for the LINESIZE= and PAGESIZE= system options can significantly affect the appearance of the output that is produced by some SAS procedures.</td>
</tr>
<tr>
<td>**NUMBER</td>
<td>NONUMBER** and <strong>PAGENO=</strong> system options</td>
</tr>
<tr>
<td><strong>TITLE statement</strong></td>
<td>prints titles at the top of each output page. In the windowing environment, SAS prints the following title by default: The SAS System. You can use the TITLE statement or TITLES window to replace the default title or specify other descriptive titles for SAS programs. You can use the null title statement (title;) to suppress a TITLE statement.</td>
</tr>
</tbody>
</table>
modify existing style and table definitions if you do not want to create the definitions from scratch.


---

### Reformatting Values in Output

Certain SAS statements, procedures, and options enable you to print values using specified formats. In a windowing environment, you can use the Properties window to control how values are displayed. You can apply or change formats with the FORMAT and ATTRIB statements, or with the Properties window in a windowing environment.

The FORMAT procedure enables you to design your own formats and informats, giving you added flexibility in displaying values. See “FORMAT Procedure” in *Base SAS Procedures Guide* for more information about the FORMAT procedure, and *SAS System Options: Reference* for information about all other SAS system options.

---

### Printing Missing Values

SAS represents ordinary missing numeric values in a SAS listing as a single period and missing character values as a blank space. If you specify special missing values for numeric variables, SAS writes the letter or the underscore. For character variables, SAS writes a series of blanks equal to the length of the variable.

The MISSING= system option enables you to specify a character to print in place of the period for ordinary missing numeric values. For more information, see the “MISSING= System Option” in *SAS System Options: Reference*.

---

### The SAS Log

#### Structure of the Log

The SAS log is a record of everything that you do in your SAS session or with your SAS program. Depending on the setting of SAS system options, the method of running SAS, and the program statements that you specify, the log can include the following types of information:

- program statements
- names of data sets created by the program
- notes, warnings, or error messages encountered during program execution
- the number of variables and observations each data set contains
- processing time required for each step
Operating Environment Information: The SAS log appears differently depending on your operating environment. See the SAS documentation for your operating environment.

Example Code 27.1  Sample SAS Log

```
NOTE: Copyright (c) 2016 by SAS Institute Inc., Cary, NC, USA. 1
NOTE: SAS (r) Proprietary Software 9.4 (TS1B0) 2 Licensed to SAS Institute Inc., Site 1. 3
NOTE: This session is executing on the W32_7PRO platform. 4
NOTE: SAS initialization used:
   real time           4.19 seconds
   cpu time            0.85 seconds
   options pagesize=24
   linesize=64 pageno=1 nodate; 5
   data logsample; 6
   infile
   ! '\myserver\my-directory-path\sampledata.dat'; 7
   input LastName $ ID $ Gender $ Birth : date7. score1
   ! score2 score3 score4 score5 score6 score7 score8;
   format Birth mmddyy8.;
   run;
NOTE: The infile '\myserver\my-directory-path\sampledata.dat' is: 8
Filename='\myserver\my-directory-path\sampledata.dat',
RECFM=V,LRECL=256,File Size (bytes)=296,
Last Modified=08Jun2009:15:42:26,
Create Time=08Jun2009:15:42:26
NOTE: 5 records were read from the infile 9
'\myserver\my-directory-path\sampledata.dat'.
The minimum record length was 58.
The maximum record length was 59.
NOTE: The data set WORK.LOGSAMPLE has 5 observations and 12 variables. 10
NOTE: DATA statement used (Total process time):
   real time           0.21 seconds 11
   cpu time            0.03 seconds
9
10   proc sort data=logsample; 12
11      by LastName;
12   run;
NOTE: There were 5 observations read from the data set WORK.LOGSAMPLE.
NOTE: The data set WORK.LOGSAMPLE has 5 observations and 12 variables. 13
NOTE: PROCEDURE SORT used (Total process time):
   real time           0.01 seconds
   cpu time            0.01 seconds
13
14   proc print data=logsample; 14
15      by LastName;
16   run;
NOTE: There were 5 observations read from the data set WORK.LOGSAMPLE.
NOTE: PROCEDURE PRINT used (Total process time):
   real time           0.03 seconds
   cpu time            0.03 seconds
```

The following list corresponds to the circled numbers in the SAS log shown above:

1  copyright information
2  SAS release used to run this program
3  name and site number of the computer installation where the program ran
4  platform used to run the program
5 The OPTIONS statement sets a page size of 24, a line size of 64, sets the SAS output to page 1, and suppresses the date in the output.

6 SAS statements that make up the program (if the SAS system option SOURCE is enabled)

7 long statement continued to the next line. Note that the continuation line is preceded by an exclamation point (!), and that the line number does not change.

8 input file information—notes or warning messages about the raw data and where they were obtained (if the SAS system option NOTES is enabled)

9 the number and record length of records read from the input file (if the SAS system option NOTES is enabled)

10 SAS data set that your program created; notes that contain the number of observations and variables for each data set created (if the SAS system option NOTES is enabled)

11 reported performance statistics when the STIMER option or the FULLSTIMER option is set

12 procedure that sorts your data set

13 note about the sorted SAS data set

14 procedure that prints your data set

The SAS Log in Batch, Line, or Objectserver Modes

If the LOGCONFIGLOC= system option is not specified when SAS starts, you can configure the SAS log by using the LOG= system option or the LOGPARM= system option. These options can be specified in batch mode, line mode, or objectserver mode. If the LOGCONFIGLOC= system option is specified, logging is performed by the SAS logging facility and the LOGPARM= option is ignored. The LOG= option is honored only when the %S{App.Log} conversion character is specified in the logging configuration file. For information about these log system options, see "LOGPARM= System Option" in SAS System Options: Reference in the documentation for your operating environment: For information about the SAS logging facility, see SAS Logging: Configuration and Programming Reference.

The following sections discuss the log options that you can configure using the LOGPARM= system option and how you would name the SAS log for those options when the logging facility has not been initiated. The LOG= system option names the SAS log. The LOGPARM= system option enables you to append to or replace the SAS log, determine when to write to the SAS log, and start a new SAS log under certain conditions.

Appending to or Replacing the SAS Log

If you specify a destination for the SAS log in the LOG= system option, SAS verifies if a SAS log already exists. If the log does exist, you can specify how content is written to the SAS log by using the OPEN= option of the LOGPARM= system option.
In the following SAS command, both the LOG= and LOGPARM= system options are specified in order to replace an existing SAS log that is more than one day old:

```
sas -sysin "my-batch-program" -log "c:sas\SASlogs\mylog"
    -logparm open=replaceold
```

The OPEN= option is ignored when the ROLLOVER= option of the LOGPARM= system option is set to a specific size, n.

Specifying When to Write to the SAS Log

Content can be written to the SAS log either as the log content is produced or it can be buffered and written when the buffer is full. By default, SAS writes to the log when the log buffer is full. By buffering the log content, SAS performs more efficiently by writing to the log file periodically instead of writing one line at a time.

Windows Specifics: Under Windows, the buffered log contents are written periodically, using an interval specified by SAS.

You use the WRITE= option of the LOGPARM= system option to configure when the SAS log contents are written. Set LOGPARM="WRITE=IMMEDIATE" for the log content to be written as it is produced and set LOGPARM="WRITE=BUFFERED" for the log content to be written when the buffer is full.

Rolling Over the SAS Log

The SAS log can get very large for long running servers and for batch jobs. By using the LOGPARM= system option and the LOG= system options together, you can specify to roll over the SAS log to a new SAS log. When SAS rolls over the log, it closes the log and opens a new log.

The LOGPARM= system option controls when log files are opened and closed and the LOG= system option names the SAS log file. Logs can be rolled over automatically, when a SAS session starts, when the log has reached a specific size, or not at all. By using formatting directives in the SAS log name, each SAS log can be named with unique identifiers. For more information about how to roll over the SAS log, see the following examples:

- "Example : Use Directives to Name the SAS Log" on page 603
- "Example: Automatically Roll Over the SAS Log When Directives Change" on page 604
- "Example: Roll Over the SAS Log by SAS Session" on page 605
- "Example: Roll Over the SAS Log by the Log Size" on page 605

To not roll over the log at all, specify the LOGPARM= "ROLLOVER=NONE" option when SAS starts. Directives are not resolved and no rollover occurs. For example, if LOG="March#b.log", the directive #b does not resolve and the log name is March#b.log.

Writing to the Log in All Modes

In all modes, you can instruct SAS to write additional information to the log by using the following statements:
Table 27.4  Other Statements That Write Information to the SAS Log

<table>
<thead>
<tr>
<th>Statements</th>
<th>Descriptions</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA Statement with / NESTING Option</td>
<td>Specifies that a note is printed to the SAS log for the beginning and end of each DO-END and SELECT-END nesting level. This option enables you to debug mismatched DO-END and SELECT-END statements and is particularly useful in large programs where the nesting level is not obvious.</td>
<td>“DATA Statement” in SAS DATA Step Statements: Reference</td>
</tr>
<tr>
<td>ERROR Statement</td>
<td>Sets <em>ERROR</em> to 1. A message written to the SAS log is optional.</td>
<td>“ERROR Statement” in SAS DATA Step Statements: Reference</td>
</tr>
<tr>
<td>LIST Statement</td>
<td>Writes to the SAS log the input data record for the observation that is being processed.</td>
<td>“LIST Statement” in SAS DATA Step Statements: Reference</td>
</tr>
<tr>
<td>PUT Statement</td>
<td>Writes lines to the SAS log, to the SAS output window, or to an external location that is specified in the most recent FILE statement.</td>
<td>“PUT Statement” in SAS DATA Step Statements: Reference</td>
</tr>
<tr>
<td>%PUT Statement</td>
<td>Writes text or macro variable information to the SAS log.</td>
<td>“%PUT Macro Statement” in SAS Macro Language: Reference</td>
</tr>
<tr>
<td>PUTLOG Statement</td>
<td>Writes a message to the SAS log.</td>
<td>“PUTLOG Statement” in SAS DATA Step Statements: Reference</td>
</tr>
</tbody>
</table>

Use the PUT, PUTLOG, LIST, DATA, and ERROR statements in combination with conditional processing to debug DATA steps by writing selected information to the log.
Customizing the Log

Suppress the Contents of the Log

When you have large SAS production programs or an application that you run on a regular basis without changes, you might want to suppress part of the log. SAS system options enable you to suppress SAS statements and system messages, as well as to limit the number of error messages. All SAS system options remain in effect for the duration of your session or until you change the options. You should not suppress log messages until you have successfully executed the program without errors.

The following list describes some of the SAS system options that you can use to alter the contents of the log. For examples of how some of these options can be used, see “Examples: Suppress Output to the SAS Log”.

Table 27.5  Ways to Customize the Log

<table>
<thead>
<tr>
<th>SAS System Options</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPUID</td>
<td>NOCPUID</td>
</tr>
<tr>
<td>ECHO: Windows and UNIX</td>
<td>Specifies a message to be written to the SAS log while SAS initializes. The ECHO system option is valid only under the Windows and UNIX operating environments.</td>
</tr>
<tr>
<td>ECHOAUTO</td>
<td>NOECHOAUTO</td>
</tr>
<tr>
<td>ERRORS=</td>
<td>Specifies the maximum number of observations for which SAS issues complete error messages.</td>
</tr>
<tr>
<td>FULLSTATS: z/OS</td>
<td>Writes expanded statistics to the SAS log. The FULLSTATS system option is valid only under z/OS.</td>
</tr>
<tr>
<td>FULLSTIMER: Windows FULLSTIMER: UNIX</td>
<td>Writes a subset of system performance statistics to the SAS log.</td>
</tr>
<tr>
<td>ISPNOTES: z/OS</td>
<td>Specifies whether ISPF error messages are written to the SAS log. The ISPNOTES system option is valid only under the z/OS operating environment.</td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>HOSTINFOLONG</td>
<td>Writes additional operating environment information to the SAS log when SAS starts.</td>
</tr>
<tr>
<td>LOGPARM “OPEN=APPEND</td>
<td>REPLACER</td>
</tr>
<tr>
<td>MEMRPT: z/OS</td>
<td>Specifies whether memory usage statistics are written to the SAS log for each step. The MEMRPT system option is valid only under the z/OS operating environment.</td>
</tr>
<tr>
<td>MLOGIC</td>
<td>Writes macro execution trace information to the SAS log.</td>
</tr>
<tr>
<td>MLOGICNEST</td>
<td>Specifies whether to display the macro nesting information in the MLOGIC output in the SAS log.</td>
</tr>
<tr>
<td>MPRINT</td>
<td>NOMPRINT</td>
</tr>
<tr>
<td>MPRINTNEST</td>
<td>Specifies whether to display the macro nesting information in the MPRINT output in the SAS log.</td>
</tr>
<tr>
<td>MSGLEVEL=</td>
<td>Specifies the level of detail in messages that are written to the SAS log.</td>
</tr>
<tr>
<td>NEWS=</td>
<td>Specifies whether news information that is maintained at your site is written to the SAS log.</td>
</tr>
<tr>
<td>NOTE</td>
<td>NONOTES</td>
</tr>
<tr>
<td>OPLIST: Windows</td>
<td>Specifies whether the settings of the SAS system options are written to the SAS log.</td>
</tr>
<tr>
<td>OLIST: z/OS</td>
<td>Specifies whether overprinting of error messages to make them bold, is enabled.</td>
</tr>
<tr>
<td>OPLIST: UNIX</td>
<td>Specifies whether the SAS log and the listing file begin on a new page.</td>
</tr>
<tr>
<td>SAS System Option</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PRINTMSGLIST</td>
<td>NOPRINTMSGLIST</td>
</tr>
<tr>
<td>RTRACE: Windows RTRACE: UNIX</td>
<td>Produces a list of resources that are read or loaded during a SAS session and writes them to the SAS log if a location is not specified for the RTRACELOC= system option. The RTRACE system option is valid only for the Windows and UNIX operating environments.</td>
</tr>
<tr>
<td>SOURCE</td>
<td>NOSOURCE</td>
</tr>
<tr>
<td>SOURCE2</td>
<td>NOSOURCE2</td>
</tr>
<tr>
<td>SYMBOLGEN</td>
<td>NOSYMBOLGEN</td>
</tr>
<tr>
<td>VERBOSE: Windows VERBOSE: z/OS VERBOSE: UNIX</td>
<td>Specifies whether SAS writes to the batch log or to the computer monitor the values of the system options that are specified in the configuration file.</td>
</tr>
</tbody>
</table>

See *SAS System Options: Reference* for more information about how to use these and other SAS system options.

Customizing the Appearance of the Log

The following SAS statements and SAS system options enable you to customize the log. Customizing the log is helpful when you use the log for report writing or for creating a permanent record.

*Table 27.6* SAS Statements and System Options to Customize the Log

<table>
<thead>
<tr>
<th>System Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DETAILS system option</td>
<td>Specifies whether to include additional information when files are listed in a SAS library.</td>
</tr>
<tr>
<td>DMSLOGSIZE= system option</td>
<td>Specifies the maximum number of rows to display in the SAS log window.</td>
</tr>
<tr>
<td>DTRESET system option</td>
<td>Specifies whether to update the date and time in the SAS log and in the listing file.</td>
</tr>
</tbody>
</table>
FILE statement
Enables you to write the results of PUT statements to an external file. You can use LINESIZE= and PAGESIZE= options in the FILE statement to customize your report.

LINESIZE= system option
Specifies the line size (printer line width) for the SAS log and SAS output that are used by the DATA step and procedures.

MSGCASE system option: Windows
MSGCASE system option: UNIX
MSGCASE system option: z/OS
Specifies whether to display notes, warning, and error messages in uppercase letters or lowercase letters.

MISSING= system option
Specifies the character to be printed for missing numeric variable values.

NUMBER system option
Controls whether the page number is printed on the first title line of each page of printed output.

PAGE statement
Skips a specified number of lines in the SAS log.

PAGESIZE= system option
Specifies the number of lines that you can print per page of SAS output.

SKIP statement
Skips a specified number of lines in the SAS log.

STIMEFMT system option: Windows
STIMEFMT system option: UNIX
Specifies the format to use for displaying the read and CPU processing times when the STIMER system option is set. The STIMEFMT system option is valid under Windows and UNIX operating environments.

Operating Environment Information: The range of values for the FILE statement and for SAS system options depends on your operating environment. See the SAS documentation for your operating environment for more information.

Other System Options That Affect the SAS Log
The following system options pertain to the SAS log other than by the content and appearance of the SAS log:

Table 27.7 Other SAS System Options That Affect the SAS Log

ALTLOG system option: Windows
ALTLOG system option: UNIX ALTLOG system option: z/OS
Specifies the destination for a copy of the SAS log.
LOG system option: Windows LOG system option: UNIX LOG= system option: z/OS

Specifies the destination for the SAS log when SAS is run in batch mode.

LOGAPPLNAME= system option

Specifies a SAS session name that can be used for SAS logging.

Examples: Manage Output Destinations

Example: Create Default HTML Output

Example Code

The default output destination is HTML when running SAS programs. In SAS Studio, the results are displayed in the Results window.

title 'Student Weight';
  proc print data=sashelp.class;
    where weight>100;
  run;

Output 27.1 Default HTML Output

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
</tr>
<tr>
<td>4</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
</tr>
<tr>
<td>5</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
</tr>
<tr>
<td>8</td>
<td>Janet</td>
<td>F</td>
<td>15</td>
<td>62.5</td>
<td>112.5</td>
</tr>
<tr>
<td>14</td>
<td>Mary</td>
<td>F</td>
<td>15</td>
<td>66.5</td>
<td>112.0</td>
</tr>
<tr>
<td>15</td>
<td>Philip</td>
<td>M</td>
<td>16</td>
<td>72.0</td>
<td>150.0</td>
</tr>
<tr>
<td>16</td>
<td>Robert</td>
<td>M</td>
<td>12</td>
<td>64.8</td>
<td>128.0</td>
</tr>
<tr>
<td>17</td>
<td>Ronald</td>
<td>M</td>
<td>15</td>
<td>67.0</td>
<td>133.0</td>
</tr>
<tr>
<td>19</td>
<td>William</td>
<td>M</td>
<td>15</td>
<td>66.5</td>
<td>112.0</td>
</tr>
</tbody>
</table>

Note: If you previously closed the HTML destination, then your output is sent to the WORK directory by default. If you close the HTML destination and reopen it in the
same SAS session, then all output goes to the current directory. You do not have to specify ods html close; to view your output.

Key Ideas

- HTML is the default destination for output.
- The destination of your output depends on your operating environment, your mode of running SAS, and your version of SAS.

Example: Use ODS Statements to Change the Output Destination

Example Code

If you want to send your output to the LISTING destination, you can use ODS statements in your SAS programs to change the destination.

In this example, the output destination is changed from HTML to LISTING by specifying the ODS LISTING and ODS HTML CLOSE statements. By changing the output destination to LISTING, the output is automatically displayed as a list report in the SAS Output Window.

ods html close;
ods listing;
options nodate;
title 'Students';
  proc print data=sashelp.class;
    where weight>100;
  run;
ods html;
ods listing close;

Output 27.2  Listing Output in the Windowing Environment

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name</th>
<th>Sex</th>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Alfred</td>
<td>M</td>
<td>14</td>
<td>69.0</td>
<td>112.5</td>
</tr>
<tr>
<td>4</td>
<td>Carol</td>
<td>F</td>
<td>14</td>
<td>62.8</td>
<td>102.5</td>
</tr>
<tr>
<td>5</td>
<td>Henry</td>
<td>M</td>
<td>14</td>
<td>63.5</td>
<td>102.5</td>
</tr>
<tr>
<td>8</td>
<td>Janet</td>
<td>F</td>
<td>15</td>
<td>62.5</td>
<td>112.5</td>
</tr>
<tr>
<td>14</td>
<td>Mary</td>
<td>F</td>
<td>15</td>
<td>66.5</td>
<td>112.0</td>
</tr>
<tr>
<td>15</td>
<td>Phillip</td>
<td>M</td>
<td>16</td>
<td>72.0</td>
<td>150.0</td>
</tr>
<tr>
<td>16</td>
<td>Robert</td>
<td>M</td>
<td>12</td>
<td>64.8</td>
<td>128.0</td>
</tr>
<tr>
<td>17</td>
<td>Ronald</td>
<td>M</td>
<td>15</td>
<td>67.0</td>
<td>133.0</td>
</tr>
<tr>
<td>19</td>
<td>William</td>
<td>M</td>
<td>15</td>
<td>66.5</td>
<td>112.0</td>
</tr>
</tbody>
</table>
Key Ideas

- If you want a more permanent solution, you can change your settings so that every time you run SAS, your output is sent to the LISTING destination by default.

See Also


Examples: Rolling Over the SAS Log

Example : Use Directives to Name the SAS Log

Example Code

The following example shows how to use directives in the LOG= system option to include the year, the month, and the day in the SAS log name:

-log "c:\saslog\#Y#b#dsas.log"

When the SAS log is created on February 2, 2019, the name of the log is 2019Feb02sas.log.

Key Ideas

- Directives enable you to add information to the SAS log name such as the day, the hour, the system node name, or a unique identifier. You can include one or more directives in the name of the SAS log when you specify the log name in the LOG system option.
- A directive is a processing instruction that is used to uniquely name the SAS log.
- Directives resolve only when the value of the ROLLOVER= option of the LOGPARM= system option is set to AUTO or SESSION.
- If directives are specified in the log name and the value of the ROLLOVER option is NONE or a specific size, \( n \), the directive characters, such as \#b or \#Y, become part of the log name.
- Using the example above for the LOG system option, if the LOGPARM= system option specifies ROLLOVER=NONE, the name of the SAS log is \#Y\#b\#dsas.log.
See Also

- For a complete list of directives, see “LOGPARM= System Option” in SAS System Options: Reference

Example: Automatically Roll Over the SAS Log When Directives Change

Example Code

When the SAS log name contains one or more directives and the ROLLOVER= option of the LOGPARM= system option is set to AUTO, SAS closes the log and opens a new log when the directive values change. The new SAS log name contains the new directive values.

The table below shows some of the log names that are created when SAS is started on the second of the month at 6:15 AM, using this SAS command:

```sas
sas -objectserver -log "london#n#d#$H.log"
-logparm
"rollover=auto"
```

Key Ideas

- The directive #n inserts the system node name into the log name. #d adds the day of the month to the log name. #H adds the hour to the log name. The node name for this example is Thames.
- Under the Windows and UNIX operating environments, you can begin directives with either the % symbol or the # symbol, and use both symbols in the same directive.
- The log for this SAS session rolls over when the hour changes and when the day changes.

Table 27.8 Log Names for Rolled Over Logs

<table>
<thead>
<tr>
<th>Rollover Time</th>
<th>Log Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS initialization</td>
<td>londonThames0206.log</td>
</tr>
<tr>
<td>First rollover</td>
<td>londonThames0207.log</td>
</tr>
<tr>
<td>Last log of the day</td>
<td>londonThames0223.log</td>
</tr>
<tr>
<td>First log past midnight</td>
<td>londonThames0300.log</td>
</tr>
</tbody>
</table>
See Also

- For a complete list of directives, see “LOGPARM= System Option” in SAS System Options: Reference

Example: Roll Over the SAS Log by SAS Session

Example Code

To roll over the log at the start of a SAS session, specify the LOGPARM="ROLLOVER=SESSION" option when SAS starts. The example below creates a log filename that contains the user name that started the SAS session.

```sas
sas -log "%l.log"
   -logparm "rollover=session"
```

Key Ideas

- SAS resolves the system-specific directives that are specified in the LOG= system option and uses the resolved value to name the new log file.
- Under the Windows and UNIX operating environments, you can begin directives with either the % symbol or the # symbol, and use both symbols in the same directive.
- No roll over occurs during the SAS session, and the log file is closed at the end of the SAS session.

See Also

- “LOGPARM= System Option” in SAS System Options: Reference

Example: Roll Over the SAS Log by the Log Size

Example Code

To roll over the log when the log reaches a specific size, specify the LOGPARM="ROLLOVER=n" option when SAS starts.

```sas
sas -log "test%H%M.log"
   -logparm "rollover=80"
```
Key Ideas

- \( n \) is the maximum size of the log, in bytes, and it cannot be smaller than 10K (10,240) bytes.
- Under the Windows and UNIX operating environments, you can begin directives with either the % symbol or the # symbol, and use both symbols in the same directive.
- When the log reaches the specified size, SAS closes the log and appends the text “old” to the filename (for example, londonold.log). SAS opens a new log using the value of the LOG= option for the log name and ignores the OPEN= option statement in the LOGPARM system option.
- This is useful so that SAS never writes over an existing log file. Directives in log names are ignored for logs that roll over based on log size.
- To ensure unique log filenames between servers, SAS creates a lock file that is based on the log filename. The lock filename is \texttt{logname.lck}, where \texttt{logname} is the value of the LOG= option.
- If a lock file exists for a server log and another server specifies the same log name, a number is appended to the log and lock filenames for the second server. The numbers begin with 2 and increment by 1 for subsequent requests for the same log filename. For example, if a lock exists for the log file london.log, the second server log would be london2.log and the lock file would be london2.lck.

See Also

- “LOGPARM= System Option” in \textit{SAS System Options: Reference}

Examples: Suppress Output to the SAS Log

Example: Suppress the Printing of the Source Program to the SAS Log Using the NOSOURCE System Option

Example Code

In this example, the first DATA step prints the source code to the SAS log. The source code in this case contains a password, which is displayed in the SAS log when the DATA step executes:

```sas
data mytable;
  password="ABC123";
  x = 1;
run;
```
Example Code 27.2  SAS Log output that shows how the password is printed to the SAS log in the source code

You can suppress the printing of the source program to the log by using the NOSOURCE system option.

```
options nosource;
data mytable;
  password="ABC123";
run;
```

Example Code 27.3  SAS Log output that shows how the source code is suppressed from printing to the log

```
options nosource;

NOTE: The data set WORK.MYTABLE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
   real time           0.02 seconds
   cpu time            0.01 seconds
```

Key Ideas

- The SOURCE | NOSOURCE System Option specifies whether SAS writes source statements to the SAS log.
- The NOSOURCE System Option does not write SAS source statements to the SAS log.

See Also

- “SOURCE System Option” in SAS System Options: Reference

Example: Suppress the Printing of Variable Data to the SAS Log Using the NOLIST Option

Example Code

In this example, the NOSOURCE option suppresses the printing of the source program to the SAS log. However, because the second DATA step generates an error, the password data is printed to the SAS log in the log note.

```
options nosource;
data mytable;
  password="123ABC";
```
run;
data mytable2;
set mytable;
password=input(password,datetime.);
run;

**Example Code 27.4** Log output that shows how the password is printed to the SAS log in the NOTE

```
NOTE: The data set WORK.MYTABLE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
real time           0.02 seconds
cpu time            0.03 seconds

NOTE: Numeric values have been converted to character values at the places given by:
   (Line):(Column).
   123:14
NOTE: Invalid argument to function INPUT at line 123 column 14.
   password= _ERROR_=1 _N_=1
NOTE: Mathematical operations could not be performed at the following places. The results of the
   operations have been set to missing values.
   Each place is given by: (Number of times) at (Line):(Column).
   1 at 123:14
NOTE: There were 1 observations read from the data set WORK.MYTABLE.
NOTE: The data set WORK.MYTABLE2 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
real time           0.02 seconds
cpu time            0.03 seconds
```

You can suppress the printing variable data when there is an error by specifying the NOLIST option in the DATA statement.

```
data mytable2 / NOLIST;
set mytable;
password=input(password,datetime.);
run;
```

**Example Code 27.5** Log output that shows how the NOLIST option suppresses the printing of variable data to the SAS log when there is an error

```
NOTE: Numeric values have been converted to character values at the places given by:
   (Line):(Column).
   127:14
NOTE: Invalid argument to function INPUT at line 127 column 14.
NOTE: NOLIST option on the DATA statement suppressed output of variable listing.
NOTE: Mathematical operations could not be performed at the following places. The results of the
   operations have been set to missing values.
   Each place is given by: (Number of times) at (Line):(Column).
   1 at 127:14
NOTE: There were 1 observations read from the data set WORK.MYTABLE.
NOTE: The data set WORK.MYTABLE2 has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
real time           0.02 seconds
cpu time            0.00 seconds
```
Key Ideas

- The NOLIST DATA statement option suppresses the output of all variables to the SAS log when the value of _ERROR_ is 1.
- NOLIST must be the last option in the DATA statement.

See Also

- “DATA Statement” in SAS DATA Step Statements: Reference

Example: Suppress the Printing of All Notes to the SAS Log Using the NONOTES System Option

Example Code

These examples use the NONOTES system option to suppress the printing of notes to the SAS log.

data example;
   password="ABC123";
run;

**Example Code 27.6**  Log output showing how notes are printed to the SAS log

```sas
17   data example;
18      password="ABC123";
19   run;

NOTE: The data set WORK.EXAMPLE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time 0.02 seconds
      cpu time 0.01 seconds
```

You can suppress the printing of notes by using the NONOTES system option.

```sas
options NONOTES;
data example;
   password="ABC123";
run;
```

**Example Code 27.7**  Log output showing how notes are suppressed in the SAS log

```sas
20   options NONOTES;
21   data example;
22      password="ABC123";
23   run;
```
Key Ideas

- The NOTES system option specifies whether notes are written to the SAS log.
- NONOTES specifies that SAS does not write notes to the SAS log.

See Also

- "NOTES System Option" in SAS System Options: Reference
Universal Printing

Definition of Universal Printing

Universal Printing is a system that enables you to create a variety of document and graphic output formats. For example, you can use Universal Printing to create HTML files or PDF and RTF documents.

You can define printers with Universal Printing, and set options to control the printed output. In addition to creating the various document and graphic output types, you can send output to a printer.

SAS routes all printing through Universal Printing services. All Universal Printing features are controlled by system options, thereby enabling you to control many print features, even in batch mode.

See SAS Viya Universal Printing for details.
## Protecting Files

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of a Password</td>
<td>616</td>
</tr>
<tr>
<td>Assigning Passwords</td>
<td>617</td>
</tr>
<tr>
<td>Syntax</td>
<td>617</td>
</tr>
<tr>
<td>Assigning a Password with a DATA Step</td>
<td>617</td>
</tr>
<tr>
<td>Assigning a Password to an Existing Data Set</td>
<td>618</td>
</tr>
<tr>
<td>Assigning a Password with a Procedure</td>
<td>618</td>
</tr>
<tr>
<td>Assigning a Password with the SAS Windowing Environment</td>
<td>619</td>
</tr>
<tr>
<td>Assigning a Password outside of SAS</td>
<td>619</td>
</tr>
<tr>
<td>Removing or Changing Passwords</td>
<td>619</td>
</tr>
<tr>
<td>Using Password-Protected SAS Files in DATA and PROC Steps</td>
<td>619</td>
</tr>
<tr>
<td>How SAS Handles Incorrect Passwords</td>
<td>620</td>
</tr>
<tr>
<td>Assigning Complete Protection with the PW= Data Set Option</td>
<td>620</td>
</tr>
<tr>
<td>Encoded Passwords</td>
<td>621</td>
</tr>
<tr>
<td>Using Passwords with Views</td>
<td>622</td>
</tr>
<tr>
<td>Levels of Protection</td>
<td>622</td>
</tr>
<tr>
<td>PROC SQL Views</td>
<td>623</td>
</tr>
<tr>
<td>SAS/ACCESS Views</td>
<td>623</td>
</tr>
<tr>
<td>DATA Step Views</td>
<td>624</td>
</tr>
<tr>
<td>SAS Data File Encryption</td>
<td>624</td>
</tr>
<tr>
<td>About Encryption on SAS Data Files</td>
<td>624</td>
</tr>
<tr>
<td>SAS Proprietary Encryption</td>
<td>625</td>
</tr>
<tr>
<td>AES Encryption</td>
<td>626</td>
</tr>
<tr>
<td>AES Encryption and Referential Integrity Constraints</td>
<td>628</td>
</tr>
<tr>
<td>Passwords and Encryption with Generation Data Sets, Audit Trails, Indexes, and Copies</td>
<td>628</td>
</tr>
<tr>
<td>Blotting Passwords and Encryption Key Values</td>
<td>628</td>
</tr>
<tr>
<td>Check the SAS Log</td>
<td>629</td>
</tr>
<tr>
<td>Examples of Passwords and Encryption Keys That Are Not Blotted</td>
<td>629</td>
</tr>
<tr>
<td>Using Macros</td>
<td>630</td>
</tr>
<tr>
<td>Length of Passwords</td>
<td>630</td>
</tr>
<tr>
<td>Metadata-Bound Libraries</td>
<td>631</td>
</tr>
</tbody>
</table>
Definition of a Password

SAS software enables you to restrict access to members of SAS libraries by assigning passwords to the members. You can assign passwords to all member types except catalogs. You can specify three levels of protection: Read, Write, and Alter. When a password is assigned, it appears as uppercase Xs in the log.

Note: This document uses the terms SAS data file and SAS view to distinguish between the two types of SAS data sets. Passwords work differently for type VIEW than they do for type DATA. The term “SAS data set” is used when the distinction is not necessary.

read
protects against reading the file.

write
protects against changing the data in the file. For SAS data files, write protection prevents adding, modifying, or deleting observations.

alter
protects against deleting or replacing the entire file. For SAS data files, alter protection also prevents modifying variable attributes and creating or deleting indexes.

Alter protection does not require a password for Read or Write access; write protection does not require a password for Read access. For example, you can read an alter-protected or write-protected SAS data file without knowing the Alter or Write password. Conversely, read and write protection do not prevent any operation that requires alter protection. For example, you can delete a SAS data set that is read- or write-protected only without knowing the Read or Write password.

To protect a file from being read, written to, deleted, or replaced by anyone who does not have the proper authority, assign read, write, and alter protection. To allow others to read the file without knowing the password, but not change its data or delete it, assign just write and alter protection. To completely protect a file with one password, use the PW= data set option. For more information, see “Assigning Complete Protection with the PW= Data Set Option” on page 620.

Note: Because of how SAS opens files, you must specify the Read password to update a SAS data set that is only read-protected.

Note: The levels of protection differ somewhat for the member type VIEW. See “Using Passwords with Views” on page 622.
Assigning Passwords

Syntax

To set a password, first specify a SAS data set in one of the following:
- a DATA statement
- the MODIFY statement of the DATASETS procedure
- an OUT= option in some procedures
- the CREATE VIEW statement in PROC SQL
- the ToolBox

Then assign one or more password types to the data set. The data set might already exist, or the data set might be one that you create. The following is an example of syntax:

(password-type=password ... password-type=password>)

where password is a valid eight-character SAS name and password-type can be one of the following SAS data set options:
- ALTER=
- PW=
- READ=
- WRITE=

TIP Each password option must be coded on a separate line to ensure that they are properly blotted in the SAS log.

CAUTION

Keep a record of any passwords that you assign! If you forget or do not know the password, you cannot get the password from SAS.

Assigning a Password with a DATA Step

You can use data set options to assign passwords to unprotected members in the DATA step when you create a new SAS data file.

This example prevents deletion or modification of the data set without a password.

/* assign a write and an alter password to MYLIB.STUDENTS */
data mylib.students(write=yellow alter=red);
input name $ sex $ age;
datalines;
Amy f 25
... more data lines ...
;
This example prevents reading or deleting a stored program without a password and also prevents changing the source program.

/* assign a read and an alter password to the SAS view ROSTER */
data mylib.roster(read=green alter=red) / view=mylib.roster;
   set mylib.students;
run;
libname stored 'SAS-library-2';

/* assign a read and alter password to the program file SOURCE */
data mylib.schedule / pgm=stored.source(read=green alter=red);
   ... DATA step statements ...
run;

Note: When you replace a SAS data set that is alter-protected, the new data set inherits the Alter password. To change the Alter password for the new data set, use the MODIFY statement in the DATASETS procedure.

Assigning a Password to an Existing Data Set

You can use the MODIFY statement in the DATASETS procedure to assign passwords to unprotected members if the SAS data file already exists.

/* assign an alter password to STUDENTS */
proc datasets library=mylib;
   modify students(alter=red);
run;

Assigning a Password with a Procedure

You can assign a password after an OUT= data set specification in some procedures.

/* assign a write and an alter password to SCORE */
proc sort data=mylib.math
   out=mylib.score(write=yellow alter=red);
   by number;
run;

You can assign a password in a CREATE TABLE or a CREATE VIEW statement in PROC SQL.

/* assign an alter password to the SAS view BDAY */
proc sql;
   create view mylib.bday(alter=red) as
   query-expression;
Assigning a Password with the SAS Windowing Environment

You can create or change passwords for any data file using the Password Window in the SAS windowing environment. To invoke the Password Window from the ToolBox, use the global command SETPASSWORD followed by the filename. This opens the password window for the specified data file.

Assigning a Password outside of SAS

A SAS password does not control access to a SAS file beyond the SAS system. You should use the operating system-supplied utilities and file-system security controls in order to control access to SAS files outside of SAS.

Removing or Changing Passwords

To remove or change a password, use the MODIFY statement in the DATASETS procedure. For more information, see “DATASETS Procedure” in Base SAS Procedures Guide.

Using Password-Protected SAS Files in DATA and PROC Steps

To access password-protected files, use the same data set options that you use to assign protection.

```sas
/* Assign a read and alter password to the stored program file*/
/*STORED.SOURCE */
data mylib.schedule / pgm=stored.source
  (read=green alter=red);
  <... more DATA step statements ...>
run;

/*Access password-protected file*/
proc sort data=mylib.score(write=yellow alter=red);
  by number;
run;

/* Print read-protected data set MYLIB.AUTOS */
proc print data=mylib.autos(read=green);
run;
```
Passwords are hierarchical in terms of gaining access. For example, specifying the ALTER password gives you Read and Write access. The following example creates the data set States, with three different passwords, and then reads the data set to produce a plot:

```sas
data mylib.states(read=green write=yellow alter=red);
  input density crime name $;
  datalines;
  151.4 6451.3 Colorado
  … more data lines …
;
proc plot data=mylib.states(alter=red);
  plot crime*density;
run;
```

How SAS Handles Incorrect Passwords

If you are using the SAS windowing environment and you try to access a password-protected member without specifying the correct password, you receive a dialog box that prompts you for the appropriate password. The text that you enter in this window is not displayed. You can use the PWREQ= data set option to control whether a dialog box appears after a user enters a missing or incorrect password. PWREQ= is most useful in SCL applications.

If you are using batch or noninteractive mode, you receive an error message in the SAS log if you try to access a password-protected member without specifying the correct password.

If you are using interactive line mode, you are also prompted for the password if you do not specify the correct password. When you enter the password and press the Enter key, processing continues. If you cannot give the correct password, you receive an error message in the SAS log.

Assigning Complete Protection with the PW= Data Set Option

The PW= data set option assigns the same password for each level of protection. This data set option is convenient for thoroughly protecting a member with just one
password. If you use the PW= data set option, those who have access need to remember only one password for total access.

- To access a member whose password is assigned using the PW= data set option, use the PW= data set option. You can also use the data set option that equates to the specific level of access that you need:

```sas
/* create a data set using PW=, then use READ= to print the data set */
data mylib.states(pw=orange);
  input density crime name $;
  datalines;
  151.4 6451.3 Colorado
  ;
proc print data=mylib.states(read=orange);
run;
```

- PW= can be an alias for other password options:

```sas
/* Use PW= as an alias for ALTER=. */
data mylib.college(alter=red);
  input name $ 1-10 location $ 12-25;
  datalines;
  Vanderbilt Nashville
  Rice       Houston
  Duke       Durham
  Tulane     New Orleans
  ;
proc datasets library=mylib;
  delete college(pw=red);
run;
```

Encoded Passwords

Encoding a password enables you to write SAS programs without having to specify a password in plain text. The PWENCODE procedure uses encoding to disguise passwords. With encoding, one character set is translated to another character set through some form of table lookup. An encoded password is intended to prevent casual, non-malicious viewing of passwords. You should not depend on encoded passwords for all your data security needs; a determined and knowledgeable attacker can decode the encoded passwords.

When an encoded password is used, the syntax parser decodes the password and accesses the file. The encoded password is never written in plain text to the SAS log. SAS does not accept passwords longer than eight characters. If an encoded password is decoded and is longer than eight characters, SAS reads it as an incorrect password and sends an error message to the SAS log. For more information, see “PWENCODE Procedure” in Base SAS Procedures Guide.
Using Passwords with Views

Levels of Protection

The levels of protection for SAS views and stored programs are similar to the levels of protection for other types of SAS files. However, with SAS views, passwords affect not only the underlying data, but also the view’s definition (or source statements).

You can specify three levels of protection for SAS views: Read, Write, and Alter. The following section describes how these data set options affect the underlying data as well as the view’s descriptor information. Unless otherwise noted, the term “view” refers to any type of SAS view and the term “underlying data” refers to the data that is accessed by the SAS view:

Read
- protects against reading of the SAS view's underlying data
- prevents the display of source statements in the SAS log when using DESCRIBE
- allows replacement of the SAS view

Write
- protects the underlying data associated with a SAS view by insisting that a Write password is given
- prevents the display of source statements in the SAS log when using DESCRIBE
- allows replacement of the SAS view

Alter
- prevents the display of source statements in the SAS log when using DESCRIBE
- protects against replacement of the SAS view

For example, to DESCRIBE a view that has both Read and Write protection, you must specify its Write password. Similarly, to DESCRIBE a view that has both Read and Alter protection, you must specify its Alter password (since Alter is the more restrictive of the two).

The following program shows how to use the DESCRIBE statement to view the descriptor information for a Read-protected and Alter-protected view:

```sas
/*create a view with read and alter protection*/
data exam / view=exam(read=read alter=alter);
   set grades;
run;
/*describe the view by specifying the most restrictive password */
data view=exam(alter=alter);
   describe;
run;
```
Example Code 29.1 Password-protected View

```
NOTE: DATA step view WORK.EXAM is defined as:

data exam / view=exam(read=XXX alter=XXXXX);
  set grades;
run;

NOTE: DATA statement used (Total process time):
real time 0.01 seconds
cpu time 0.01 seconds
```

For more information, see “DESCRIBE Statement” in SAS DATA Step Statements: Reference and “DATA Statement” in SAS DATA Step Statements: Reference.

In most DATA and PROC steps, the way you use password-protected views is consistent with how you use other types of password-protected SAS files. For example, the following PROC PRINT prints a Read-protected view:

```
proc print data=mylib.grade(read=green);
run;
```

Note: You might experience unexpected results when you place protection on a SAS view if some type of protection is already placed on the underlying data set.

PROC SQL Views

Typically, when you create a PROC SQL view from a password-protected SAS data set, you specify the password in the FROM clause in the CREATE VIEW statement using a data set option. In this way, you can access the underlying data without re-specifying the password when you use the view later. For example, the following statements create a PROC SQL view from a Read-protected SAS data set, and drop a sensitive variable:

```
proc sql;
  create view mylib.emp as
    select * from mylib.employee(pw=orange drop=salary);
quit;
```

Note: You can create a PROC SQL view from password-protected SAS data sets without specifying their passwords. Use the view that you are prompted for the passwords of the SAS data sets named in the FROM clause. If you are running SAS in batch or noninteractive mode, you receive an error message.

SAS/ACCESS Views

SAS/ACCESS software enables you to edit View descriptors and, in some interfaces, the underlying data. To prevent someone from editing or reading (browsing) the View descriptor, assign Alter protection to the view. To prevent
When you create a DATA step view using a password-protected SAS data set, specify the password in the View definition. In this way, when you use the view, you can access the underlying data without respecifying the password.

The following statements create a DATA step view using a password-protected SAS data set, and drop a sensitive variable:

```sas
data mylib.emp / view=mylib.emp;
  set mylib.employee(pw=orange drop=salary);
run;
```

Note that you can use the SAS view without a password, but access to the underlying data requires a password. This is one way to protect a particular column of data. In the above example, `proc print data=mylib.emp;` executes, but `proc print data=mylib.employee;` fails without the password.

### SAS Data File Encryption

**About Encryption on SAS Data Files**

SAS passwords and metadata-bound data sets restrict access to SAS data sets within SAS. But neither can prevent SAS data sets from being viewed at the operating environment system level or from being read by an external program. Encryption provides security of your SAS data outside of SAS by writing to disk the encrypted data that represents the SAS data. The data is decrypted by the SAS system as it is read from the disk, but is not decrypted when read at the operating system level or by external programs.

Encryption does not affect file access. However, SAS recognizes all host security mechanisms that control file access and can extend host security mechanisms by binding the data sets to metadata. You can use encryption and those security mechanisms together.

There are three types of algorithms that SAS uses for encrypting data files:

- **SAS Proprietary Encryption** on page 625 is implemented with the `ENCRYPT=Yes` data set option.
- **AES (Advanced Encryption Standard) encryption** on page 626 is implemented with the `ENCRYPT=AES` or `ENCRYPT=AES2` data set option.

Beginning in SAS 9.4M1, a metadata-bound library administrator can require that all data files in the bound library be encrypted with one of the three algorithms. For more information, see “Requiring Encryption for Metadata-Bound Data Sets” in *Base SAS Procedures Guide* and *SAS Guide to Metadata-Bound Libraries*.
Table 29.1  Encryption Features

<table>
<thead>
<tr>
<th>Features</th>
<th>ENCRYPT=YES</th>
<th>ENCRYPT=AES</th>
<th>ENCRYPT=AES2</th>
</tr>
</thead>
<tbody>
<tr>
<td>License required</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Encryption level</td>
<td>Medium</td>
<td>High</td>
<td>Highest</td>
</tr>
<tr>
<td>Algorithm supported</td>
<td>SAS Proprietary (within Base SAS software)</td>
<td>AES</td>
<td>AES2</td>
</tr>
<tr>
<td>Installation required</td>
<td>No (part of Base SAS software)</td>
<td>No SAS/SECURE (delivered with Base SAS software)</td>
<td>No SAS/SECURE (delivered with Base SAS software)</td>
</tr>
<tr>
<td>Operating environments supported</td>
<td>UNIX</td>
<td>UNIX</td>
<td>UNIX</td>
</tr>
<tr>
<td></td>
<td>Windows</td>
<td>Windows</td>
<td>Windows</td>
</tr>
<tr>
<td></td>
<td>z/OS</td>
<td>z/OS</td>
<td>z/OS</td>
</tr>
<tr>
<td>SAS version support</td>
<td>8 and later</td>
<td>9.4 and later</td>
<td>9.4m5 and later</td>
</tr>
</tbody>
</table>

See Also

"AUTHLIB Procedure" in Base SAS Procedures Guide

SAS Proprietary Encryption

SAS Proprietary Encryption is licensed with Base SAS software and is available in all deployments. There are two types of SAS Proprietary Encryption.

- A 32-bit rolling-key encryption technique that is used for SAS data set encryption with passwords.

  This encryption technique for SAS data sets uses parts of the passwords that are stored in the SAS data set as part of the 32-bit rolling key encoding of the data. This encryption provides a medium level of security. Users must supply the appropriate passwords to authorize their access to the data, but with the speed of today’s computers, it could be subjected to a brute force attack on the 2,563,160,682,591 possible combinations of valid password values. Many of which must produce the same 32-bit key. SAS/SECURE and data set support of AES, which is also shipped with Base SAS software, provides a higher level of security.

- A 32-bit fixed-key encryption routine used for communications, such as passwords for login objects, passwords in configuration files, login passwords, internal account passwords, and so on.

SAS Proprietary Encryption for SAS data sets is implemented with the ENCRYPT= data set option. You can use the ENCRYPT= data set option only when you are creating a SAS data file. You must also assign a password when encrypting a data
file with SAS Proprietary Encryption. At a minimum, you must specify the READ= data set option or the PW= data set option at the same time you specify ENCRYPT=YES. Because passwords are used in this encryption technique, you cannot change any password on an encrypted data set without re-creating the data set.

The following rules apply to data file encryption:

- To copy an encrypted SAS data file, the output engine must support encryption. Otherwise, the data file is not copied.
- Encrypted files work only in Release 6.11 or in later releases of SAS.
- You cannot encrypt SAS data views, because they contain no data.
- If the data file is encrypted, all associated indexes are also encrypted.
- Encryption requires approximately the same amount of CPU resources as compression.
- You cannot use PROC CPORT on encrypted SAS data files.

The following example creates an SAS data set with SAS Proprietary Encryption:

```sas
data salary(encrypt=yes read=green);
  input name $ yrsal bonuspct;
  datalines;
  Muriel   34567  3.2
  Bjorn    74644  2.5
  Freda    38755  4.1
  Benny    29855  3.5
  Agnetha  70998  4.1
;

To print this data set, specify the Read password:

```sas
proc print data=salary(read=green);
run;
quit;
```

**TIP** Each password option must be coded on a separate line to ensure that they are properly blotted in the SAS log.

### AES Encryption

In SAS 9.4 release, AES encryption of data sets is available. You specify ENCRYPT=AES when creating a data set. AES produces a strong encryption by using a key value that can be up to 64 characters long. Beginning in SAS 9.4M5 release, a stronger AES key generation algorithm is available. You use ENCRYPT=AES2 data set option. Instead of passwords that are stored in the data set (SAS Proprietary encryption), AES and AES2 uses a key value that is not stored in the data set. The key value is created using the ENCRYPTKEY= data set option when the data set is created. You cannot change the ENCRYPTKEY= key value on
an AES encrypted data set without re-creating the data set or using PROC AUTHLIB MODIFY to change the recorded key of a metadata-bound library. For more information, see “AUTHLIB Procedure” in Base SAS Procedures Guide.

The following rules apply to AES and AES2 encryption of data sets:

- You use SAS/SECURE software, which is licensed with Base SAS software and is available in all deployments.
- You must use the ENCRYPTKEY= data set option when creating or accessing an AES encrypted data set unless the metadata-bound library administrator has securely recorded the encryption key in metadata to which the data set is bound. For more information, see “AUTHLIB Procedure” in Base SAS Procedures Guide and SAS Guide to Metadata-Bound Libraries.
- To copy an AES-encrypted data file, the output engine must support AES encryption. Otherwise, the data file is not copied.
- Releases before SAS 9.4 cannot use an AES-encrypted data file.
- Releases before SAS 9.4M5 cannot use an AES encrypted file that uses AES2 key generation algorithm.
- SAS Viya cannot access data sets created with ENCRYPT=AES2.
- You cannot encrypt SAS views, because they contain no data.
- If two or more data files are referentially related and any of them are AES encrypted, then all must be AES encrypted. The encryption key for all of the files must be the same unless the files are bound to metadata with the key securely recorded in the metadata. For more information about metadata-bound libraries, see “Metadata-Bound Library” in Base SAS Procedures Guide.
- If the data file has AES encryption, all associated indexes have AES encryption.
- You cannot use PROC CPORT on AES encrypted data files.

The ENCRYPTKEY= data set option does not protect the AES encrypted file from deletion or replacement. AES encrypted data sets can be deleted by using either of the following scenarios without having to specify an encrypt key value:

- the KILL option in PROC DATASETS
- the DROP statement in PROC SQL

The encrypt key only prevents access to the contents of the file. To protect the file from unauthorized deletion or replacement with the SAS system, the file must also contain an ALTER= password or be bound to metadata.

The following example creates an encrypted data set using AES encryption:

data salary(encrypt=aes encryptkey=green);
  input name $ yrsal bonuspct;
  datalines;
Muriel   34567  3.2
Bjorn    74644  2.5
Freda    38755  4.1
Benny    29855  3.5
Agnetha  70998  4.1
;

to print this data set, specify the ENCRYPTKEY= key value:

proc print data=salary(encryptkey=green);
run;
quit;
TIP Each password and encryption key option must be coded on a separate line to ensure that they are properly blotted in the SAS log.

If you omit the ENCRYPTKEY= key value when accessing an AES secured data set, a dialog box appears and prompts you to add the ENCRYPTKEY= key value. If the data set is metadata-bound and the key has been stored in the metadata for the library, the dialog box does not appear.

Figure 29.1 Dialog Box for ENCRYPTKEY=

See Also

“AUTHLIB Procedure” in Base SAS Procedures Guide

AES Encryption and Referential Integrity Constraints

Data files with referential integrity constraints can use AES encryption. All primary key and foreign key data files must use the same encryption key that opens all referencing foreign key and primary key data files.

Passwords and Encryption with Generation Data Sets, Audit Trails, Indexes, and Copies

SAS extends password protection, SAS Proprietary encryption, and AES encryption to other files associated with the original protected file. This includes generation data sets, indexes, audit trails, and copies. You can access protected or encrypted generation data sets, indexes, audit trails, and copies of the original file. The same rules, syntax, and behavior for invoking the original password protected or encrypted files apply. SAS views cannot have generation data sets, indexes, or audit trails. For more information about encryption, see “SAS Proprietary Encryption” on page 625 and “AES Encryption” on page 626.

Blotting Passwords and Encryption Key Values
Check the SAS Log

You need to check the SAS log to ensure that any password value or encryption key value is blotted out. This applies to the READ=, WRITE=, ALTER=, PW=, and ENCRYPTKEY= options.

In most cases, placing the password=value pair on a separate line blots the value:

```
data &ds(
  read=secret
  encrypt=aes
  encryptkey=evenmoreso
);
  x=1;
run;
```

Examples of Passwords and Encryption Keys That Are Not Blotted

The following examples are password values and encryption-key values that are not blotted in the SAS log:

- Do not use a macro variable for the libref or data set in a DATA statement:

  ```
  %let ds=dataset;
  data &ds(read=secret);
  x=1;
  run;
  ```

  The following is written to the SAS log:

  ```
  111  %let ds=dataset;
  112  data &ds(alter=secret);
  113     x=1;
  114  run;
  ```

  NOTE: The data set WORK.DATASET has 1 observations and 1 variables.
  NOTE: DATA statement used (Total process time):

  - real time 0.00 seconds
  - cpu time 0.00 seconds

- Using an incorrect password for a data set in certain procedures causes passwords in the log:

  ```
  proc append base=here(PW=XXXXXXX) data=more(READ=secret2);
  run;
  ```

- Typing errors cause the following passwords to show in the SAS log:
If the code causes an ERROR message, the password is not blotted. For example, in the following code the libref is misspelled causing SAS to issue the message: "ERROR: Libref MYLUB is not assigned." and the password is not blotted.

```sas
libname mylib 'c:";
  data mylub.abc(
    read=secret
  );
  x=1;
  run;
```

The following output is written to the SAS log:

```sas
636 libname mylib 'c:";
NOTE: Libref MYLIB was successfully assigned as follows:
Engine:        V9
Physical Name: c:\
637     data mylub.abc(
ERROR: Libref MYLUB is not assigned.
640     read=secret
641     );
641 x=1;
641 run;
```

NOTE: The SAS System stopped processing this step because of errors.

---

### Using Macros

When a password is assigned within a macro, the password is not blotted in the SAS log when the macro executes. To prevent the password from being revealed in the SAS log, you can redirect the SAS log to a file. For more information, see "PRINTTO Procedure" in *Base SAS Procedures Guide*.

---

### Length of Passwords

In some cases, the length of the displayed password is fixed at eight blotted characters. In other cases, the number of blotted characters is the length of the password. Output from the OPTIONS procedure, VERBOSE option, and OPLIST option have a fixed length of eight.

When a password value is being reported, its length is fixed at eight. But when a password value is simply being echoed from an input statement, it retains its input length. This example shows the length of the passwords:

```sas
options pdfpassword=(open=a owner=b );
proc options option=pdfpassword;
```
run;

The following is written to the SAS log:

```sas
634  options pdfpassword=XXXXXXXX XXXXXXX X;
635  proc options option=pdfpassword;run;

SAS (r) Proprietary Software Release 9.4  TS1M0
PDFPASSWORD=XXXXXXXX
  Specifies the password to use to open a PDF document and the
  password used by a PDF document owner.
NOTE: PROCEDURE OPTIONS used (Total process time):
  real time          0.04 seconds
  cpu time           0.00 seconds
```

---

## Metadata-Bound Libraries

A metadata-bound library is a physical library that is tied to a corresponding metadata secured table object. Each physical table within a metadata-bound library has information in its header that points to a specific metadata object. The pointer creates a security binding between the physical table and the metadata object. The binding ensures that SAS universally enforces metadata-layer access requirements for the physical table—regardless of how a user requests access from SAS. For more information, see *SAS Guide to Metadata-Bound Libraries*.

The AUTHLIB procedure is used to create, access, and modify metadata-bound libraries. This procedure is intended for use by SAS administrators. Users who lack sufficient privileges in either the metadata layer or the host layer cannot use this procedure. For more information, see “AUTHLIB Procedure” in *Base SAS Procedures Guide*.
Repairing SAS Files

Repairing Damage to SAS Data Sets and Catalogs

Overview

SAS can repair structural damage to SAS data sets and SAS catalogs resulting from certain system errors. It can also rebuild simple indexes and integrity constraints that were inadvertently deleted or damaged by using operating system commands. These repairs can be made with the DLDMGACTION= system option, the DLDMGACTION= data set option, or the PROC DATASETS REPAIR statement. For more information about the damage that SAS can repair and how, see "Understanding Damage That Can Be Repaired with SAS" on page 634.

SAS cannot repair the following:

- truncated SAS data sets
- data sets whose audit files are damaged or have been deleted
- data sets and catalogs that are on a damaged storage device
- SAS views
- stored compiled DATA step programs
complex indexes
indexes copied by using a DATA step
indexes that were deleted when using the FORCE option in the SORT procedure. The FORCE option overwrites the original file.

For these situations, take the action recommended in the table that follows.

**Table 30.1  Recommended Action by File or Damage Type**

<table>
<thead>
<tr>
<th>SAS File Type or Damage Type</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA step views and PROC SQL views</td>
<td>Re-create the file.</td>
</tr>
<tr>
<td>Stored compiled DATA step programs</td>
<td></td>
</tr>
<tr>
<td>Truncated SAS data sets, or SAS data sets whose audit file is damaged or deleted</td>
<td>Recover from a backup device.¹</td>
</tr>
<tr>
<td>SAS data sets and catalogs that are on a damaged storage device</td>
<td></td>
</tr>
<tr>
<td>Catalogs that cannot be repaired by the DLDMGACTION= system option or data set option, or by using the PROC DATASETS REPAIR statement</td>
<td></td>
</tr>
<tr>
<td>Complex indexes</td>
<td>Repair the data set without indexes, and then re-create the index using the PROC DATASETS &quot;INDEX CREATE Statement&quot; in Base SAS Procedures Guide. Or, recover the data set from a backup device.</td>
</tr>
<tr>
<td>Indexes copied by using a DATA step</td>
<td></td>
</tr>
<tr>
<td>Indexes that were deleted when using the FORCE option in the SORT procedure.</td>
<td></td>
</tr>
</tbody>
</table>

¹ When an audit file is damaged or accidentally deleted, recover the data set, its index, and the audit file.

**Understanding Damage That Can Be Repaired with SAS**

SAS can recover and repair SAS data sets (including indexes and integrity constraints) and SAS catalogs when one of the following events occurs:

- A system failure occurs while the data set or catalog is being updated.
- The disk where the data set (including the index file) or catalog is stored becomes full before the file is completely written to it.
- An input/output error occurs while writing to the data set, index file, or catalog.
- An index is accidentally deleted with operating system commands.
- A data set is damaged by using operating system commands to copy or rename the data set, but not its associated index.
SAS repairs the structure of a data set by copying and re-creating the data set. A data set's indexes and integrity constraints are reconstructed from metadata in the data set header. SAS attempts to restore the data in the data set, but in cases where the disk is full before the data is written to it, the data set might not contain the last several updates that occurred before the system failed. When this occurs, it is recommended that you recover the data set from a backup device.

SAS repairs damaged catalogs as follows. SAS checks the catalog to see which entries are damaged. If there is an error reading an entry, the entry is copied. If an error occurs during the copy process, then the entry is automatically deleted. For severe damage, the entire catalog is copied to a new catalog.

**Understanding the Repair Tools**

The DLDMGACTION= system option enables you to program a proactive repair response when damage to a data set or catalog is detected. Set the DLDMGACTION= system option to perform the action that you want to apply to the majority of data set and catalog damage cases in your SAS session. For example, for an interactive session, you might set the DLDMGACTION= system option to PROMPT to show the menu of repair options on the spot. Or you might set the system option to FAIL and pursue all repairs with the DLDMGACTION= data set option and the PROC DATASETS REPAIR statement.

The DLDMGACTION=NOINDEX and REPAIR options should be used only when you have seen an actual problem and are trying to rectify it. DLDMGACTION=REPAIR repairs the data set and all of its indexes to the extent that it can. When DLDMGACTION=REPAIR is set as a system option, damaged data sets and catalogs are automatically repaired the next time that the data set or catalog is accessed. This can prevent you from examining a data set for lost observations. DLDMGACTION=REPAIR also does not distinguish between regular data sets and generation data sets. Repairing a generation data set can be tricky. An automated repair is not appropriate. For information to repair generation data sets, see “Generation Data Sets” in SAS V9 LIBNAME Engine: Reference.

Differences between the DLDMGACTION=data set option and the PROC DATASETS REPAIR statement are as follows:

- The PROC DATASETS REPAIR statement can repair one or multiple data sets or catalogs at once. A MEMTYPE= option enables you to limit repairs to only files of type DATA or CATALOG. The DLDMGACTION= data set option operates on a single data set or catalog.

- The DLDMGACTION= data set option enables you to repair a data set with or without its indexes and integrity constraints by specifying a NOINDEX option. The PROC DATASETS REPAIR statement always repairs all indexes. The NOINDEX option automatically repairs the data set without the indexes and integrity constraints, deletes the index file, updates the data set to reflect the disabled indexes and integrity constraints, and limits the data set to be opened in INPUT mode. A warning is written to the SAS log instructing you to execute the “REBUILD Statement” in Base SAS Procedures Guide to correct or delete the disabled indexes and integrity constraints.

Some SAS applications have a shipped default DLDMGACTION= system option setting; others do not. It is a good idea to check the DLDMGACTION= system option setting to ensure that it is set to a value with which you are comfortable.
A damage log is maintained for every SAS data set that indicates how many times the data set has been repaired and the date of the last repair. You can view the repair log with PROC CONTENTS. See "Example: Determine How Many Times a Data Set Was Repaired" on page 642.

There is no damage log for catalogs.

Recommended Tasks for Repairing SAS Data Sets and Catalogs with SAS

Table 30.2 Tasks and Tools for Repairing Files with SAS

<table>
<thead>
<tr>
<th>Task</th>
<th>Code</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine the DLDMGACTION= system option setting for your SAS session</td>
<td>proc options option=dl dmgaction value; run;</td>
<td>Prints the setting of the DLDMGACTION= system option to the SAS log.</td>
<td>&quot;Example: Determine the Default Repair Action for Your SAS Session&quot; on page 637.</td>
</tr>
<tr>
<td>Change how SAS responds when it detects damaged files (optional)</td>
<td>options dl dmgaction=value;</td>
<td>Enables you to change or configure a repair response for the SAS session.</td>
<td>&quot;Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement&quot; on page 641</td>
</tr>
<tr>
<td>Actively repair damaged files yourself</td>
<td>(dl dmgaction=value)</td>
<td>Enables you to specify a repair action for a specific SAS data set or catalog.</td>
<td>&quot;Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option&quot; on page 640.</td>
</tr>
<tr>
<td></td>
<td>proc datasets; repair filename; run;</td>
<td>Attempts to repair a damaged SAS data set or catalog.</td>
<td>&quot;Example: Repair a SAS Data Set with the REPAIR Statement&quot; on page 639</td>
</tr>
<tr>
<td></td>
<td>proc datasets; rebuild filename; run;</td>
<td>Specifies whether to restore or delete indexes and integrity constraints that were disabled by the DLDMGACTION=NOINDEX option.</td>
<td>&quot;Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement&quot; on page 641</td>
</tr>
</tbody>
</table>
### Examples

**Example: Determine the Default Repair Action for Your SAS Session**

**Example Code**

The `DLDMGACTION=` system option controls the default repair action for your SAS session. You can determine the setting of the `DLDMGACTION=` system option by using `PROC OPTIONS`.

```sas
proc options option=dldmgaction value;
run;
```

SAS returns information that is similar to the following to the log:

**Example Code 30.1  Log Output from the OPTIONS Procedure**

<table>
<thead>
<tr>
<th>Option Value Information For SAS Option DLDMGACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value: REPAIR</td>
</tr>
<tr>
<td>Scope: Default</td>
</tr>
<tr>
<td>How option value set: Shipped Default</td>
</tr>
</tbody>
</table>
Key Ideas

- Most SAS sessions return the value DLDMGACTION=REPAIR, unless an administrator has configured a different value in a configuration file. See “DLDMGACTION= System Option” in SAS System Options: Reference for information about other DLDMGACTION= values.

- You can change the default repair response by changing the setting of the DLDMGACTION= system option. Or you can override the system option setting for specific data sets and catalogs by using the DLDMGACTION= data set option.

- There are many items that DLDMGACTION=REPAIR cannot repair. For information about what can and cannot be repaired, see “Understanding Damage That Can Be Repaired with SAS” on page 634. The DLDMGACTION= system option is not a replacement for having a current backup.

See Also

- “OPTIONS Procedure”
- “DLDMGACTION= System Option” in SAS System Options: Reference
- “DLDMGACTION= Data Set Option” in SAS Data Set Options: Reference
- “Example: Repair a SAS Data Set with the REPAIR Statement” on page 639.
- “Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option” on page 640.

Example: Change the DLDMGACTION= System Option

Example Code

The following code sets the DLDMGACTION= system option to PROMPT and verifies the setting with PROC OPTIONS.

```sas
options dldmgaction=prompt;

proc options option=dldmgaction value;
  run;
```
Key Ideas

- The OPTIONS statement changes the repair setting for all interactions in the current SAS session. The system option can also be specified at SAS invocation, in the SAS System Options window and in the SASV9_OPTIONS environment variable (UNIX only in SAS 9.4; Linux only in SAS Viya).
- The PROMPT setting instructs SAS to display a dialog box that lets the user make the repair decision for any given SAS data set or catalog.
- To override the default setting for a specific SAS data set or catalog, use the DLDMGACTION= data set option.

See Also

- "OPTIONS Statement" in SAS Global Statements: Reference
- "DLDMGACTION= System Option" in SAS System Options: Reference

Example: Repair a SAS Data Set with the REPAIR Statement

Example Code

The following code specifies to repair a damaged SAS data set named mylib.myfile that is encrypted. The PROC DATASETS REPAIR statement is used to repair the data set.

```
libname mylib "C:\dldmgactiontest";

proc datasets lib=mylib;
    repair myfile /encryptkey=secret;
run;
quit;
```

After the process completes, the following message is written to the log:
Example Code 30.2  Log Message from the PROC DATASETS REPAIR Statement

```
NOTE: Repairing MYLIB.MYFILE (memtype=DATA).
NOTE: File MYLIB.MYFILE.DATA is damaged.
NOTE: Data set MYLIB.MYFILE contained no structural errors, however some
changes during last update may not have been written to disk.
NOTE: Indexes recreated:
  1 Simple indexes
```

Key Ideas

- The functionality of the PROC DATASETS REPAIR statement is similar to that of
  DLDMGACTION=REPAIR. The statement attempts to restore damaged SAS data
  sets and catalogs to a usable condition. It repairs a data set's simple indexes and
  integrity constraints.
- The REPAIR statement can be used to repair SAS data sets that are used by SAS
  programming languages that do not support the DLDMGACTION= system option.
  The SAS FedSQL and SAS DS2 languages are examples of programming
  languages that do not support SAS system options or data set options.
- Like DLDMGACTION=REPAIR, the REPAIR statement is not a replacement for
  having a current backup. If you have extensive damage to your data set, the
  REPAIR statement will not correct it. You might need to recover the data set from a
  system backup.
- To repair a damaged catalog, you must use a version of SAS that runs in the same
  host operating environment as the one in which the catalog was created. There is
  no CEDA access for catalogs.

See Also

- "REPAIR Statement" in Base SAS Procedures Guide.

Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option

Example Code

The following request sets the DLDMGACTION= data set option for a request on
damaged data set mylib.myfile. For this example, the default action for the SAS
session is DLDMGACTION=REPAIR. The data set option specifies the NOINDEX
option for a PROC PRINT request on the data set using the DLDMGACTION= data
set option.

```
proc print data=mylib.myfile[dldmgaction=noindex];
run;
```

SAS returns information that is similar to the following to the log:
Example Code 30.3  Log Message Returned for a Data Set Repaired with the DLDMGACTION=NOINDEX Option

```
100  proc print data=mylib.myfile(dldmgaction=noindex);
   NOTE: File MYLIB.MYFILE.DATA is damaged.
   NOTE: Data set MYLIB.MYFILE contained no structural errors, however some changes during last update may not have been written to disk.
   WARNING: SAS data file MYLIB.MYFILE.DATA was damaged and has been partially repaired. To complete the repair, execute the DATASETS procedure REBUILD statement.
101  run;
```

Key Ideas

- The DLDMGACTION= data set option must be specified on an existing data set. The option is not valid in statements that create a data set, except for the SET statement.
- The NOINDEX option specifies to repair a damaged data set without the indexes and integrity constraints. The repair also deletes the index file and updates the data set to reflect the disabled indexes and integrity constraints. If the data set is not damaged, the data set option has no effect.
- Because this data set is damaged, a warning is written to the SAS log instructing you to execute the PROC DATASETS REBUILD statement to correct or delete the disabled indexes and integrity constraints. The data set can be opened only in input mode until you make the change.
- When used in the SET statement, the DLDMGACTION= data set option simply omits indexes from the new data set.

See Also

- “DLDMGACTION= Data Set Option” in SAS Data Set Options: Reference
- “Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement” on page 641

Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement

Example Code

The following code rebuilds data set indexes and integrity constraints that were disabled by the DLDMGACTION=NOINDEX data set option. The request rebuilds indexes and integrity constraints that were removed in “Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option” on page 640.

```
proc datasets library=mylib;
   rebuild myfile;
```
run;
quit;

SAS returns information that is similar to the following to the log:

**Example Code 30.4  Log Messages Written by the PROC DATASETS REBUILD Statement**

```
NOTE: Rebuilding MYLIB.MYFILE (memtype=DATA).
NOTE: Indexes recreated:
  2 Simple indexes
```

**Key Ideas**

- The PROC DATASETS REBUILD statement enables you specify whether to restore or delete the indexes and integrity constraints that were disabled with the DLDMACTION=NOINDEX option.
- The REBUILD statement restores disabled indexes and integrity constraints by default (shown here). To delete the indexes and integrity constraints from the data set, specify the NOINDEX option.

**See Also**

- “PROC DATASETS REBUILD Statement”
- “Example: Determine How Many Times a Data Set Was Repaired” on page 642.

**Example: Determine How Many Times a Data Set Was Repaired**

**Example Code**

To determine how many times a data set has been repaired, use the CONTENTS procedure. This example shows how many times data set mylib.myfile has been repaired.

```
proc contents data=mylib.myfile;
run;
```

The number of repairs is reported in the Engine/Host Dependent Information section of the CONTENTS procedure output.
The CONTENTS procedure has two fields that report information about repairs:

- Number of Data Set Repairs
- Last Repair

See Also

“CONTENTS Procedure”
Compressing SAS Data Sets

Compression in SAS

Compressing a SAS data set is a process that reduces the number of bytes required to represent each observation. In a compressed data set, each observation is a varying-length record, while in an uncompressed data set, each observation is a fixed-length record.

Advantages of compression include the following:

- reduced storage requirements for the file
- less I/O operations necessary to read from or write to the data during processing

Disadvantages of compression include the following:

- increased CPU resource requirements to read a compressed file because of the overhead of uncompressing each observation and, if updating, compressing again when written to disk
- situations when the resulting file size can increase rather than decrease

If conserving space is a primary concern, then testing is recommended. For information about compression, see the documents that are listed in the following table.

### Table 31.1 Compression in SAS Engines

<table>
<thead>
<tr>
<th>Engine</th>
<th>Documentation</th>
<th>Usage Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>V9 engine</td>
<td>“Compression” in SAS V9 LIBNAME Engine: Reference</td>
<td>Supports COMPRESS=NO</td>
</tr>
<tr>
<td>SPD Engine</td>
<td>“Updates to a Compressed SPD Engine Data Set” in SAS Scalable Performance Data Engine: Reference</td>
<td>Supports COMPRESS=NO</td>
</tr>
<tr>
<td>Engine</td>
<td>Documentation</td>
<td>Usage Notes</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------------------------------------------------------------------------</td>
<td>---------------------------</td>
</tr>
</tbody>
</table>
Moving SAS Files

Moving Files between Operating Environments

The procedures for moving SAS files from one operating environment to another vary according to your operating environment, the member type and version of the SAS files that you want to move, and the methods that you have available for moving the files.

For details about this subject, see Moving and Accessing SAS Files.
Definitions for Cross-Environment Data Access (CEDA)

This documentation explains the restrictions, benefits, and behavior of CEDA processing. Here are a few useful concepts:

Cross-Environment Data Access (CEDA) enables a SAS file that was created in a directory-based operating environment (for example, UNIX or Windows) to be processed in an incompatible environment or under an incompatible session encoding. With CEDA, the processing is automatic and transparent. You do not need to create a transport file, use SAS procedures that convert the file, or change your SAS program. CEDA supports files that were created with SAS 7 and later releases. CEDA is a Base SAS feature.

data representation is the form in which data is stored in a particular operating environment. Different operating environments use different standards or conventions for storing data. (See "Compatible Data Representations" on page 653.)
Floating-point numbers can be represented in IEEE floating-point format or IBM floating-point format.

Data alignment can be on a 1-byte, 4-byte, or 8-byte boundary, depending on data type requirements for the operating environment.

Data type lengths can be 8 bits or more for a character data type, 16 bit, 32 bit, or 64 bit for an integer data type, 32 bit for a single-precision floating-point data type, and 64 bit for a double-precision floating-point data type.

The ordering of bytes can be big Endian or little Endian.

**encoding**

is a set of characters (letters, logograms, digits, punctuation, symbols, control characters, and so on) that have been mapped to numeric values (called code points) that can be used by computers. The code points are assigned to the characters in the character set by applying an encoding method. Some examples of encodings are WLatin1 and Danish EBCDIC. (See “Encoding Combinations That Do Not Need CEDA Processing for Transcoding” in SAS National Language Support (NLS): Reference Guide.)

**incompatible**

describes a file that has a different data representation or encoding than the current SAS session. CEDA enables access to many types of incompatible files.

### Advantages of CEDA

CEDA offers these advantages:

- You can transparently process a supported SAS file with no knowledge of the file's data representation or encoding.
- No transport files are created. CEDA requires a single translation to the current session's data representation, rather than multiple translations from the source representation to the transport file to the target representation.
- CEDA eliminates the need to perform multiple steps in order to process the file.
- CEDA does not require a sign-on as is needed in SAS/CONNECT or a dedicated server as is needed in SAS/SHARE.

### SAS File Processing with CEDA

#### Conditions That Invoke CEDA

CEDA is used in these situations:

- when the encoding of character values for the SAS file is incompatible with the currently executing SAS session encoding.
when the data representation of the SAS file is incompatible with the data representation of the currently executing SAS session. For example, an incompatibility can occur if you move a file from an operating environment like Windows to an operating environment like UNIX, or if you have upgraded to 64-bit UNIX from 32-bit UNIX. See “Compatible Data Representations” on page 653.

CEDA supports SAS 7 and later SAS files that are created in directory-based operating environments like UNIX and Windows. CEDA provides the following SAS file processing for these SAS engines:

**BASE**
shipped default Base SAS engine and alias for the V9 engine in SAS 9, V8 in SAS 8, and V7 in SAS 7. Referred to as the V9 engine in the CEDA topics.

**SASESOCK**
TCP/IP port engine for SAS/CONNECT software.

**SPDE**
SAS Scalable Performance Data Engine, with some exceptions. For more information, see “Accessing SPD Engine Files on Another Host” in SAS Scalable Performance Data Engine: Reference. (Support was added in SAS 9.4M5.)

**TAPE**
sequential engine and alias for V9TAPE in SAS 9, V8TAPE in SAS 8, V7TAPE in SAS 7.

<table>
<thead>
<tr>
<th>Table 33.1 SAS File Processing Provided by CEDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAS File Type</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>SAS data set</td>
</tr>
<tr>
<td>PROC SQL view (DATA step view is not supported)</td>
</tr>
<tr>
<td>SAS/ACCESS view for Oracle or SAP</td>
</tr>
<tr>
<td>MDDB file²</td>
</tr>
</tbody>
</table>

¹ For output processing that replaces an existing SAS data set, there are behavioral differences. For more information, see “How Output Processing Affects Encoding and Data Representation” on page 654.

² CEDA supports SAS 8 and later MDDB files.

**Restrictions for CEDA**

CEDA has the following restrictions:

- SAS catalogs are not supported. Catalog entries could include formats, stored compiled macros, SAS/AF applications, SAS/GRAPH output, SAS code, SCL code, data, and other entry types that are specific to various SAS procedures.
Update processing is not supported.

Integrity constraints cannot be read or updated.

An audit trail file cannot be updated, but it can be read.

Indexes are not supported. Therefore, WHERE optimization with an index is not supported.

Extended attributes cannot be updated, but they can be read.

Other files that are not supported include DATA step views, SAS/ACCESS views that are not for SAS/ACCESS for Oracle or SAP, stored compiled DATA step programs, item stores, DMDB files, FDB files, or any SAS file that was created prior to SAS 7.

On z/OS, members of UNIX file system libraries can be created using any SAS data representation. However, when bound libraries are created, they are assigned the data representation of the SAS session that creates the library. SAS does not allow the creation of bound library members with a data representation that differs (except for the encoding) from the data representation of the library. For example, if you create a bound library with 31-bit SAS on z/OS, the library has a data representation of MVS_32 for the duration of its existence, and you cannot use the OUTREP option of the LIBNAME statement to create a member in the library with a data representation other than MVS_32. For more information about library implementation types for V9 and sequential engines on z/OS, see SAS Companion for z/OS.

SAS translates between data representations or transcodes between encodings as the data is read. When you use multiple procedures, SAS must translate or transcode the data multiple times, which can affect system performance.

If a data set is damaged, CEDA cannot process the file in order to repair it. CEDA does not support update processing, which is required in order to repair a damaged data set. To repair the file, you must move it back to the environment where it was created or to a compatible environment that does not invoke CEDA processing. For information about how to repair a damaged data set, see the REPAIR statement in the DATASETS procedure in Base SAS Procedures Guide.

Transcoding could result in character data loss when encodings are incompatible. See “Transcoding Considerations” in SAS National Language Support (NLS): Reference Guide.

Loss of precision can occur in numeric variables when you move data between operating environments. If a numeric variable is defined with a short length, you can try increasing the length of the variable. Full-size numeric variables are less likely to encounter a loss of precision with CEDA. For more information, see “Numeric Precision” on page 98.

Numeric variables have a minimum length of either 2 or 3 bytes, depending on the operating environment. In an operating environment that supports a minimum of 3 bytes (such as Windows or UNIX), CEDA cannot process a numeric variable that was created with a length of 2 bytes (for example, in z/OS). If you encounter this restriction, then use the XPORT engine or the CPORT and CIMPORT procedures instead of CEDA.

Note: If you encounter these restrictions because your files were created under a previous version of SAS, consider using the MIGRATE procedure, which is documented in the Base SAS Procedures Guide. PROC MIGRATE retains many features, such as integrity constraints, indexes, and audit trails.
Compatible Data Representations

In the following table, operating environments are grouped by compatibility. CEDA is not used if you process the file under a data representation of the same group. For example, a SOLARIS_X86_64 data set is compatible in a LINUX_POWER_64 session. (The current release of SAS does not run on some of these environments, but they are included here for completeness.)

<table>
<thead>
<tr>
<th>Data Representation Value and Environment Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALPHA_TRU64 (Tru64 UNIX)</td>
<td>Although all of the environments in this group are compatible, catalogs are an exception. Catalogs are compatible between Tru64 UNIX and Linux for Itanium. Catalogs are compatible between Linux for x64, Solaris for x64, and Linux on the Power Architecture. Linux on the Power Architecture is added in SAS Viya 3.5 and is not supported in SAS 9.</td>
</tr>
<tr>
<td>LINUX_IA64 (Linux for Itanium-based systems)</td>
<td></td>
</tr>
<tr>
<td>LINUX_X86_64 (Linux for x64)</td>
<td></td>
</tr>
<tr>
<td>SOLARIS_X86_64 (Solaris for x64)</td>
<td></td>
</tr>
<tr>
<td>LINUX_POWER_64 (Linux on the Power Architecture)</td>
<td></td>
</tr>
<tr>
<td>ALPHA_VMS_32 (OpenVMS Alpha)</td>
<td>Although the 32-bit and 64-bit OpenVMS environments have different data representations for some compiler types, SAS data sets that are created by the V9 engine do not store the data types that are different. Therefore, if the encoding is compatible, CEDA is not used between these environments. However, note that SAS 9 does not support SAS 8 catalogs from OpenVMS. You can migrate the catalogs with the MIGRATE procedure. For more information, see the Base SAS Procedures Guide.</td>
</tr>
<tr>
<td>ALPHA_VMS_64 (OpenVMS Alpha)</td>
<td></td>
</tr>
<tr>
<td>VMS_IA64 (OpenVMS on HP Integrity)</td>
<td></td>
</tr>
<tr>
<td>HP_IA64 (HP-UX for the Itanium Processor Family Architecture)</td>
<td>All of the environments in this group are compatible.</td>
</tr>
<tr>
<td>HP_UX_64 (HP-UX for PA-RISC, 64-bit)</td>
<td></td>
</tr>
<tr>
<td>RS_6000_AIX_64 (AIX)</td>
<td></td>
</tr>
<tr>
<td>SOLARIS_64 (Solaris for SPARC)</td>
<td></td>
</tr>
<tr>
<td>HP_UX_32 (HP-UX for PA-RISC)</td>
<td>All of the environments in this group are compatible.</td>
</tr>
<tr>
<td>MIPS_ABI (MIPS ABI)</td>
<td></td>
</tr>
<tr>
<td>RS_6000_AIX_32 (AIX)</td>
<td></td>
</tr>
<tr>
<td>SOLARIS_32 (Solaris for SPARC)</td>
<td></td>
</tr>
<tr>
<td>LINUX_32 (Linux for Intel architecture)</td>
<td>All of the environments in this group are compatible.</td>
</tr>
<tr>
<td>INTEL_ABI (ABI for Intel architecture)</td>
<td></td>
</tr>
</tbody>
</table>
### Data Representation Value and Environment

<table>
<thead>
<tr>
<th>Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVS_32 (31-bit SAS on z/OS)</td>
<td>No other environments are compatible.</td>
</tr>
<tr>
<td>MVS_64_BFP (64-bit SAS on z/OS)</td>
<td>No other environments are compatible.</td>
</tr>
<tr>
<td>OS2 (OS/2 for Intel)</td>
<td>No other environments are compatible.</td>
</tr>
<tr>
<td>VAX_VMS (OpenVMS VAX)</td>
<td>No other environments are compatible.</td>
</tr>
<tr>
<td>WINDOWS_32 (32-bit SAS on Microsoft Windows)</td>
<td>SAS data sets are compatible in these Windows environments. Other file types such as catalogs are not compatible between 32-bit and 64-bit SAS for Windows.</td>
</tr>
<tr>
<td>WINDOWS_64 (64-bit SAS on Microsoft Windows, for both Itanium-based systems and x64)</td>
<td></td>
</tr>
</tbody>
</table>

### Compatible Encodings


However, even when encodings are compatible, CEDA processing could be invoked by an incompatible data representation.

In addition, some Microsoft Word characters such as smart quotation marks could cause truncation errors. The characters require more than one byte in UTF-8 encoding.

If the data contains 7-bit ASCII characters (U.S. English) only, then the data is compatible in any other ASCII session, including UTF-8. Other ASCII encodings use the high-order bit for different national characters.

### How Output Processing Affects Encoding and Data Representation

For output processing that replaces an existing SAS data set, the engines behave differently regarding the following attributes:

- **encoding**
  - The V9 engine uses the encoding of the file from the source library. That is, the encoding is cloned.
  - The TAPE engine uses the current SAS session encoding, except with PROC COPY.
  - For both the V9 and TAPE engines, by default PROC COPY uses the encoding of the file from the source library. If, instead, you want to use the encoding of the current SAS session, specify the NOCLONE option. If you want to use a different encoding, specify the NOCLONE option and the
ENCODING= option. When you use PROC COPY with SAS/SHARE or SAS/CONNECT, the default behavior is to use the encoding of the client session (not the server session).

- The SPD Engine uses the current SAS session encoding. The CLONE option of PROC COPY is not supported.

Data representation

- The V9 and TAPE engines use the data representation of the current SAS session, except with PROC COPY.

- For both the V9 and TAPE engines, by default PROC COPY uses the data representation of the file from the source library. If, instead, you want to use the data representation of the current SAS session, specify the NOCLONE option. If you want to use a different data representation, specify the NOCLONE option and the OUTREP= option. When you use PROC COPY with SAS/SHARE or SAS/CONNECT, the default behavior is to use the data representation of the client session (not the server session).

- The SPD Engine uses the data representation of the current SAS session. The CLONE option of PROC COPY is not supported.

### Alternatives to Using CEDA

Because of the restrictions, it might not be feasible to use CEDA. You can use the following methods in order to move files across operating environments:

**MIGRATE procedure**

PROC MIGRATE is usually the best way to migrate members in a SAS library to the current SAS release. PROC MIGRATE is a one-step copy procedure that retains the data attributes that most users want in a data migration. A SAS/CONNECT or SAS/SHARE server is required in some cases. See “Examples: Migrate SAS Libraries” in SAS V9 LIBNAME Engine: Reference.

**CPORT and CIMPORT procedures**

In the source environment, PROC CPORT writes data sets or catalogs to transport format. In the target environment, PROC CIMPORT translates the transport file into the target environment’s format. If truncation occurs, you must expand variable lengths. You can either use the CVP engine with PROC CPORT or use the EXTENDVAR= option with PROC CIMPORT.

**Data transfer services in SAS/CONNECT software**

Data transfer services is a bulk data transfer mechanism that transfers a copy of the data and performs the necessary conversion of the data from one environment’s representation to another’s, as well as any necessary conversion between SAS releases. You must establish a connection between the two SAS sessions by using the SIGNON command and then executing either PROC UPLOAD or PROC DOWNLOAD to move the data.

**Remote library services in both SAS/CONNECT software and SAS/SHARE software**

Remote library services gives you transparent access to remote data through the use of the LIBNAME statement.

**XPORT engine with the DATA step or PROC COPY**

In the source environment, the LIBNAME statement with the XPORT engine and either the DATA step or PROC COPY creates a transport file from a SAS data
set. In the target environment, the same method translates the transport file into the target environment’s format. Note that the XPORT engine does not support SAS 7 and later features, such as file and variable names that are greater than 8 bytes.

The XPORT engine converts data representations but does not perform transcoding. Use the XPORT engine to transport data sets between single-byte encodings only. The XPORT engine expects that the bytes in a character variable are single-byte, and emits them to the transport file as is if on an ASCII system, and it converts them to ASCII if on an EBCDIC system. A character variable that contains UTF-8 data would need to be transcoded to a single-byte encoding (by using the KCVT function) before writing to transport.

XML engine with the DATA step or PROC COPY

In the source environment, the LIBNAME statement with the XML engine and either the DATA step or PROC COPY creates an XML document from a SAS data set. In the target environment, the same method translates the XML document into the target environment’s format.

Examples: CEDA

Example: Understand Log Messages about CEDA

Example Code

The following PRINT procedure generates a CEDA note because the SAS session has a different data representation than that of the data set. The mytest data set was created on Linux in “Example: Specify a Data Representation to Avoid CEDA” on page 659.

```sas
libname myfiles v9 'c:\examples';
proc print data=myfiles.mytest;
run;
```

Here is the CEDA note in the SAS log. The note does not indicate an error.

**Example Code 33.1  Log Output Showing CEDA Informational Note**

| NOTE: Data file MYFILES.MYTEST.DATA is in a format that is native to another host, or the file encoding does not match the session encoding. Cross Environment Data Access will be used, which might require additional CPU resources and might reduce performance. |

The following SQL procedure attempts to update the mytest data set.

```sas
proc sql;
    insert into myfiles.mytest
    values ('other data string');
quite;
```

As shown by the following error message, CEDA does not support update processing. The update fails.
Example Code 33.2  Log Output Showing CEDA Update Error

ERROR: File MYFILES.MYTEST cannot be updated because its encoding does not match the session encoding or the file is in a format native to another host, such as SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64.

Key Ideas

- CEDA processing is transparent and automatic, but most users want to know when CEDA processing occurs. CEDA has several restrictions. For example, update processing is not supported.

- Compatible encodings do not require CEDA processing for transcoding.

  When SAS writes a CEDA note to the log, this note is informational. The note does not indicate an error.

  However, transcoding could result in character data loss when encodings are incompatible. For example, a code point in one encoding could represent a different character in another encoding. Therefore, always check your output when you process a data set under a different encoding.

See Also

- “Restrictions for CEDA” on page 651
- “Compatibility and Migration” in SAS V9 LIBNAME Engine: Reference

Example: Understand Log Messages about Truncation

Example Code

This example creates the mytrunctest data set in a double-byte character set (DBCS) session of SAS. The session encoding is SHIFT-JIS. The length of the a variable is 22 bytes.

```sas
libname myfiles v9 'c:\examples';
data myfiles.mytrunctest;
a='サンプルテキスト文字列';
run;
proc print data=myfiles.mytrunctest;
run;
```

A Unicode SAS session is started. The session encoding is UTF-8. In this session, the PROC PRINT output is truncated.

```sas
libname myfiles v9 'c:\examples';
proc print data=myfiles.mytrunctest;
```
Here is the truncation warning in the SAS log.

**Example Code 33.3  Log Output Showing Truncation Warning**

```
WARNING: Some character data was lost during transcoding in the dataset MYFILES.MYTRUNCTEST. Either the data contains characters that are not representable in the new encoding or truncation occurred during transcoding.
```

The truncation occurs because the `a` variable requires more bytes in UTF-8 encoding than in SHIFT-JIS encoding. To prevent truncation, use the CVP engine to expand the variable length. See “Example: Avoid Truncation When Copying a SAS Library” in SAS V9 LIBNAME Engine: Reference and “Example: Avoid Truncation When Migrating a SAS Library by Using a Two-Step Process” in SAS V9 LIBNAME Engine: Reference.

**Output 33.1  PROC PRINT Output Showing Truncated Data**

<table>
<thead>
<tr>
<th>Obs</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>サンプルテキスト</td>
</tr>
</tbody>
</table>

**Key Ideas**

- CEDA processing is transparent and automatic. However, transcoding could result in character data loss when encodings are incompatible.
- When SAS writes an error or warning to the log about transcoding, you are advised to carefully check the output. The most common reason for a transcoding error is truncation.

When you process a file in an encoding that uses more bytes to represent the characters, truncation could occur if the column length does not accommodate the larger character size. For example, a character might be represented in wlatin1 encoding as one byte but in UTF-8 as two bytes. Truncation can also cause "garbage" characters or incorrect characters in output.

Some Microsoft Word characters such as smart quotation marks can also cause truncation. Those characters require more than one byte in UTF-8 encoding.

- To prevent truncation, use the CVP engine to expand the variable length.

**See Also**

- “Restrictions for CEDA” on page 651
Example: Specify a Data Representation to Avoid CEDA

Example Code

In this example, the user is running 64-bit SAS on Microsoft Windows, which has a data representation of `WINDOWS_64`. The following DATA step uses the `OUTREP=` data set option to create a data set that has a data representation of `LINUX_X86_64`.

```sas
libname myfiles v9 'c:\examples';
data myfiles.mytest (outrep=linux_x86_64);
a='sample data string';
run;
proc contents data=myfiles.mytest;
run;
```

The SAS log displays the following message when the data set is created. The message indicates that CEDA is invoked to create the data set. However, CEDA is not invoked when the `mytest` data set is accessed in a SAS session on 64-bit Linux. (CEDA is invoked if the encoding is not compatible.)

```
NOTE: Data file MYFILES.MYTEST.DATA is in a format that is native to another host, or the file encoding does not match the session encoding. Cross Environment Data Access will be used, which might require additional CPU resources and might reduce performance.
```

The CEDA message is written in the log again when the CONTENTS procedure runs. Below is the output from PROC CONTENTS, showing `LINUX_X86_64` as the data representation.

Also, notice that SAS assigns the encoding `latin1 Western (ISO)`. Without an `OUTREP=` specification, the user’s session would assign `wlatin1 Western (Windows)`.

**Output 33.2  Portion of PROC CONTENTS Output**

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>MYFILES.MYTEST</th>
<th>Observations</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>1</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>08/27/2019 10:22:01</td>
<td>Observation Length</td>
<td>18</td>
</tr>
<tr>
<td>Last Modified</td>
<td>08/27/2019 10:22:01</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRUS4, LINUX_IA64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>latin1 Western (ISO)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Key Ideas

- SAS automatically invokes CEDA processing when a file’s data representation or encoding is different from the data representation or encoding of the current session.
- CEDA processing can reduce performance. Therefore, when you create a data set to be accessed in a different operating environment, you might want to specify that data representation when you create the data set.
- Use the OUTREP= data set option or LIBNAME statement option to specify the data representation of the target environment. CEDA is invoked in the current session when the data set is created.
- When you use the OUTREP= option to specify a data representation, the current SAS session encoding is ignored and a default encoding is assigned instead. This default encoding is based on the locale of the current session and the operating environment that is represented by the OUTREP= option, not the encoding of the current session.
- To change the data representation of an existing file, use the COPY procedure (or the COPY statement in the DATASETS procedure). Specify the NOCLONE option for COPY and specify OUTREP= in the LIBNAME statement.

See Also

- “OUTREP= Data Set Option” in SAS Data Set Options: Reference
- “OUTREP= LIBNAME Statement Option” in SAS V9 LIBNAME Engine: Reference
- “Compatible Data Representations” on page 653
- “Compatibility and Migration” in SAS V9 LIBNAME Engine: Reference

Example: Specify an Encoding to Avoid CEDA

Example Code

In this example, the user is running SAS on Linux for x64. The session encoding is latin1 Western (ISO). The following DATA step uses the ENCODING= data set option to create a data set that has a UTF-8 encoding.

```
libname mylnx v9 '/mydata';
data mylnx.mytest (encoding=utf8);
a='sample data string';
run;
proc contents data=mylnx.mytest;
run;
```

The SAS log displays the following message when the data set is created. The message indicates that CEDA is invoked to create the data set. However, CEDA is
not invoked when the mytest data set is accessed in a UTF-8 session encoding. (CEDA is invoked if the operating environment is not compatible.)

NOTE: Data file MYLNX.MYTEST.DATA is in a format that is native to another host, or the file encoding does not match the session encoding. Cross Environment Data Access will be used, which might require additional CPU resources and might reduce performance.

The CEDA message is written in the log again when the CONTENTS procedure runs. Below is the output from PROC CONTENTS, showing utf-8 Unicode (UTF-8) as the encoding.

**Output 33.3  Portion of PROC CONTENTS Output**

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>MYLNX.MYTEST</th>
<th>Observations</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>1</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>08/27/2019 15:31:24</td>
<td>Observation Length</td>
<td>13</td>
</tr>
<tr>
<td>Last Modified</td>
<td>08/27/2019 15:31:24</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted NO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>utf-8 Unicode (UTF-8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key Ideas**

- SAS automatically invokes CEDA processing when a file’s data representation or encoding is different from the data representation or encoding of the current session.
- CEDA processing can reduce performance. Therefore, when you create a data set to be accessed in a different session encoding, you might want to specify that encoding when you create the data set.
- Use the ENCODING= data set option or the OUTENCODING= LIBNAME statement option to specify the encoding of the target environment. CEDA is invoked in the current session when the data set is created.

**See Also**

- “INENCODING=, OUTENCODING= LIBNAME Statement Options” in *SAS V9 LIBNAME Engine: Reference*
Example: Specify a Data Representation and UTF-8 Encoding

Example Code

This example shows the correct way to specify a nondefault encoding such as UTF-8 when you use the OUTREP= data set option or LIBNAME statement option. When you specify the OUTREP= option, SAS ignores your session encoding and assigns a default encoding. If you do not want the default encoding, you must specify an encoding option.

In this example, the user is running SAS for Windows, and they want to create a data set for a target session that is on Linux for x64. The user’s encoding and the target encoding are both set to UTF-8 in their SAS configuration files. As noted above, however, the user’s UTF-8 session encoding is ignored when they specify the OUTREP= option. Therefore, the ENCODING=UTF8 option is needed below.

The following DATA step specifies both data representation and encoding for the target environment.

```sas
libname myfiles 'C:\examples';
data myfiles.test (outrep=linux_x86_64 encoding=utf8);
x=1;
run;
proc contents data=myfiles.test;
run;
```

The SAS log displays the CEDA message when the data set is created and when the CONTENTS procedure runs:

```
NOTE: Data file MYFILES.TEST.DATA is in a format that is native
to another host, or the file encoding does not match the session
encoding. Cross Environment Data Access will be used, which might
require additional CPU resources and might reduce performance.
```

The PROC CONTENTS output shows that the data representation and encoding are correctly assigned. Now the user can transport the data set to the target environment, where it will not invoke CEDA.
Output 33.4 Portion of PROC CONTENTS Output

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>MYFILES.TEST</th>
<th>Observations</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>1</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>09/03/2019 14:51:14</td>
<td>Observation Length</td>
<td>8</td>
</tr>
<tr>
<td>Last Modified</td>
<td>09/03/2019 14:51:14</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>utf-8 Unicode (UTF-8)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key Ideas

- **SAS has a default session encoding** that is based on the following two values:
  - the operating environment of the current session
  - the locale of the current session

- Many sites use the **ENCODING system option** to specify a nondefault session encoding. The **ENCODING system option** is valid in a configuration file or at SAS invocation.

- As a best practice, when you specify the **OUTREP= option** to create a data set in a different data representation, also specify an encoding option. Use the **ENCODING= data set option** or the **OUTENCODING= LIBNAME statement option**.

  If you do not specify an encoding option, the current SAS session encoding is ignored and a default encoding is assigned instead. This default encoding is based on the following two values:
  - the operating environment that is represented by the **OUTREP= option**
  - the locale of the current session

- The default behavior also occurs with the **OVERRIDE=(OUTREP=data-rep-value)** syntax in the **COPY procedure** or **COPY statement** of the **DATASETS procedure**. If you want a nondefault encoding value, then specify **OVERRIDE=(OUTREP= data-rep-value ENCODING=encoding-value])**.

See Also

Example: Determine the Encoding and Data Representation of a SAS Session or a Data Set

Example Code

The following statements return the session encoding and locale:

```sas
%put %sysfunc(getoption(encoding));
%put %sysfunc(getoption(locale));
```

Here is an example of the information that is written in the SAS log. The session encoding is wlatin1, and the locale is en_us.

```
1    %put %sysfunc(getoption(encoding));
WLATIN1
2    %put %sysfunc(getoption(locale));
EN_US
```

The OPTIONS procedure is another way to check the session encoding and locale:

```sas
proc options option=(encoding locale) define value;
run;
```

PROC OPTIONS returns detailed information about the option settings. Many lines in the example output below are omitted to highlight the relevant lines.

```
Option Value Information For SAS Option ENCODING
    Value: WLATIN1

Option Value Information For SAS Option LOCALE
    Value: EN_US
```

To determine the data representation for your session, create a temporary data set and submit the CONTENTS procedure or the CONTENTS statement of the DATASETS procedure.

```sas
data sessiontest;
  x=1;
run;
proc contents data=sessiontest;
run;
```

The `sessiontest` data set is created in the temporary Work library. Below is a portion of the PROC CONTENTS output. The OUTREP= option was not specified
When the data set was created, so `sessiontest` has the data representation of the session. The output also shows the session encoding.

**Output 33.5** Portion of PROC CONTENTS Output to Learn the Session Encoding and Data Representation

<table>
<thead>
<tr>
<th>Data Set Name</th>
<th>WORK:SESSIONTEST</th>
<th>Observations</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member Type</td>
<td>DATA</td>
<td>Variables</td>
<td>1</td>
</tr>
<tr>
<td>Engine</td>
<td>V9</td>
<td>Indexes</td>
<td>0</td>
</tr>
<tr>
<td>Created</td>
<td>09/03/2019 14:17:06</td>
<td>Observation Length</td>
<td>8</td>
</tr>
<tr>
<td>Last Modified</td>
<td>09/03/2019 14:17:06</td>
<td>Deleted Observations</td>
<td>0</td>
</tr>
<tr>
<td>Protection</td>
<td>Compressed</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Data Set Type</td>
<td>Sorted</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Representation</td>
<td>WINDOWS_54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encoding</td>
<td>latin1 Western</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Key Ideas**

- When you share data among different environments, you might need to check the encoding and data representation of the SAS session. Knowing the locale can also be useful.
- Use PROC OPTIONS or the GETOPTION function to query the session encoding.
- Use PROC CONTENTS (or the CONTENTS statement of PROC DATASETS) to see the data representation and encoding of a data set.
- You cannot use PROC OPTIONS or the GETOPTION function to query the session's data representation value, because the OUTREP= option is not available as a system option. Instead, create a temporary data set, without using OUTREP= or ENCODING= options, so that the temporary data set has the session's data representation and encoding. Submit PROC CONTENTS (or the CONTENTS statement of the DATASETS procedure). For an example that uses PROC SQL, see Sample 55054.
- In the output of PROC CONTENTS (or the CONTENTS statement of PROC DATASETS), for some operating environments, several data representations are listed. In this case, the operating environments are compatible and do not invoke CEDA processing. However, under some compatible environments, catalogs are not compatible. See “Compatible Data Representations” on page 653.
- The information that is returned by PROC CONTENTS (or the CONTENTS statement of PROC DATASETS) varies according to the operating environment and the engine.
- When you share data sets across environments, another attribute that could be helpful is Host Created, which is reported in the Engine/Host Dependent Information portion of PROC CONTENTS output.
See Also

- “GETOPTION Function” in SAS System Options: Reference
- “OPTIONS Procedure” in SAS System Options: Reference
- “CONTENTS Procedure” in Base SAS Procedures Guide
Definition of a SAS Catalog

SAS catalogs are special SAS files that store many different types of information in smaller units called catalog entries. Each entry has an entry type that identifies its purpose to SAS. A single SAS catalog can contain several types of catalog entries. Some catalog entries contain system information such as key definitions. Other catalog entries contain application information such as window definitions, help windows, formats, informat, macros, or graphics output. You can list the contents of a catalog using various SAS features, such as SAS Explorer and PROC CATALOG.

For more information, see “Catalogs” in SAS V9 LIBNAME Engine: Reference.
PART 7

SAS System Features

Chapter 35
The SAS Registry .................................................. 671

Chapter 36
The SAS Windowing Environment ............................. 689

Chapter 37
Cloud Analytic Services ........................................... 735

Chapter 38
Industry Protocols Used in SAS ............................... 739
Introduction to the SAS Registry

What Is the SAS Registry?

The SAS registry is the central storage area for configuration data for SAS. For example, the registry stores the following:

- the libraries and file shortcuts that SAS assigns at startup
- the menu definitions for Explorer pop-up menus
- the printers that are defined for use
- configuration data for various SAS products

This configuration data is stored in a hierarchical form. The form works in a manner similar to how directory-based file structures work under the operating environments in UNIX and Windows, and under the z/OS UNIX System Services (USS).
Note: Host printers are not referenced in the SAS registry.

Who Should Use the SAS Registry?

The SAS registry is designed for use by system administrators and experienced SAS users. This section provides an overview of registry tools, and describes how to import and export portions of the registry.

**CAUTION**
If you make a mistake when you edit the registry, your system might become unstable or unusable.

Wherever possible, use the administrative tools, such as the New Library window, the PRTDEF procedure, Universal Print windows, and the Explorer Options window, to make configuration changes, rather than editing the registry directly. Using the administrative tools ensures that values are stored properly in the registry when you change the configuration.

**CAUTION**
If you use the Registry Editor to change values, you are not warned if any entry is incorrect. Incorrect entries can cause errors, and can even prevent you from starting a SAS session.

Where the SAS Registry Is Stored

Registry Files in the Sasuser and the Sashelp Libraries

Although the SAS registry is logically one data store, physically it consists of two different files located in both the Sasuser and Sashelp libraries. The physical filename for the registry is regstry.sas7bitm. By default, these registry files are hidden in the SAS Explorer views of the Sashelp and Sasuser libraries.

- The Sashelp library registry file contains the site defaults. The system administrator usually configures the printers that a site uses, the global file shortcuts or libraries that are assigned at start-up, and any other configuration defaults for your site.
- The Sasuser library registry file contains the user defaults. When you change your configuration information through a specialized window such as the Print Setup window or the Explorer Options window, the settings are stored in the Sasuser library.

How to Restore the Site Defaults

If you want to restore the original site defaults to your SAS session, delete the regstry.sas7bitm file from your Sasuser library and restart your SAS session.
How Do I Display the SAS Registry?

You can use one of the following three methods to view the SAS registry:

- Issue the `REGEDIT` command. This opens the SAS Registry Editor.
- Select **Solutions** ⇒ **Accessories** ⇒ **Registry Editor**.
- Submit the following line of code:

  ```sas
  proc registry list;
  run;
  ```

  This method prints the registry to the SAS log, and it produces a large list that contains all registry entries, including subkeys. Because of the large size, it might take a few minutes to display the registry using this method.

For more information about how to view the SAS registry, see the `REGISTRY` PROCEDURE in “REGISTRY Procedure” in Base SAS Procedures Guide. Base SAS Procedures Guide.

Definitions for the SAS Registry

The SAS registry uses keys and subkeys as the basis for its structure, instead of using directories and subdirectories like the file systems in DOS or UNIX. These terms and several others described here are frequently used when discussing the SAS Registry:

key

An entry in the registry file that refers to a particular aspect of SAS. Each entry in the registry file consists of a key name, followed on the next line by one or more values. Key names are entered on a single line between square brackets ([ and ]).

The key can be a place holder without values or subkeys associated with it, or it can have many subkeys with associated values. Subkeys are delimited with a backslash (\). The length of a single key name or a sequence of key names cannot exceed 255 characters (including the square brackets and the backslash). Key names can contain any character except the backslash and are not case sensitive.

The SAS Registry contains only one top-level key, called SAS_REGISTRY. All the keys under SAS_REGISTRY are subkeys.

subkey

A key inside another key. Subkeys are delimited with a backslash (\). Subkey names are not case-sensitive. The following key contains one root key and two subkeys: `SAS_REGISTRY\HKEY_USER_ROOT\CORE`

SAS_REGISTRY

is the root key.

HKEY_USER_ROOT

is a subkey of SAS_REGISTRY. In the SAS registry, there is one other subkey at this level it is HKEY_SYSTEM_ROOT.
CORE
is a subkey of HKEY_USER_ROOT, containing many default attributes for printers, windowing, and so on.

link
a value whose contents reference a key. Links are designed for internal SAS use only. These values always begin with the word "link:"

value
the names and content associated with a key or subkey. There are two components to a value, the value name and the value content, also known as a value datum.

Figure 35.1  Section of the Registry Editor Showing Value Names and Value Data for the Subkey 'HTML'

<table>
<thead>
<tr>
<th>Contents of 'HTML'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>ab</td>
</tr>
<tr>
<td>ccco</td>
</tr>
</tbody>
</table>

.SASXREG file
a text file with the file extension .SASXREG that contains the text representation of the actual binary SAS Registry file.

Managing the SAS Registry

Primary Concerns about Managing the SAS Registry

**CAUTION**
If you make a mistake when you edit the registry, your system might become unstable or unusable. Whenever possible, use the administrative tools, such as the New Library window, the PRTDEF procedure, Universal Print windows, and the Explorer Options window, to make configuration changes, rather than editing the registry. This is to ensure that values are stored properly in the registry when changing the configuration.

**CAUTION**
If you use the Registry Editor to change values, you are not warned if any entry is incorrect. Incorrect entries can cause errors, and can even prevent you from starting a SAS session.
Backing Up the Sasuser Registry

Why Back Up the Sasuser Registry?
The Sasuser part of the registry contains personal settings. It is a good idea to back up the Sasuser part of the registry if you have made substantial customizations to your SAS session. Substantial customizations include the following:

- installing new printers
- modifying printer settings from the default printer settings that your system administrator provides for you
- changing localization settings
- altering translation tables with TRANTAB

When SAS Resets to the Default Settings
When SAS starts up, it automatically scans the registry file. SAS restores the registry to its original settings under two conditions:

- If SAS detects that the registry is corrupted, then SAS rebuilds the file.
- If you delete the registry file called regstry.sas7bitm, which is located in the Sasuser library, then SAS restores the Sasuser registry to its default settings.

**CAUTION**
Do not delete the registry file that is located in Sashelp; this prevents SAS from starting.

Ways to Back Up the Registry
There are two methods for backing up the registry and each achieves different results:

Method 1: Save a copy of the Sasuser registry file called regstry.sas7bitm.
The result is an exact copy of the registry at the moment that you copied it. If you need to use that copy of the registry to restore a broken copy of the registry, then any changes to the registry after the copy date are lost. However, it is probably better to have this backup file than to revert to the original default registry.

Method 2: Use the Registry Editor or PROC REGISTRY to back up the parts of the Sasuser registry that have changed.
The result is a concatenated copy of the registry, which can be restored from the backup file. When you create the backup file using the EXPORT= statement in PROC REGISTRY, or by using the Export Registry File utility in the Registry Editor, SAS saves any portions of the registry that have been changed. When SAS restores this backup file to the registry, the backup file is concatenated with the current registry in the following way:

---

1. The Sashelp part of the registry contains settings that are common to all users at your site. Sashelp is Write protected, and can be updated only by a system administrator.
Any completely new keys, subkeys, or values that were added to the Sasuser registry after the backup date are retained in the new registry.

Any existing keys, subkeys, or values that were changed after SAS was initially installed, then changed again after the backup, are overwritten and revert to the backup file values after the restore.

Any existing keys or subkeys (or values that retain the original default values) will have the default values after the restore.

Using the Explorer to Back Up the SAS Registry

To use the Explorer to back up the SAS Registry:

1. Start SAS Explorer with the EXPLORER command, or select View ⇒ Explorer.
2. Select Tools ⇒ Options ⇒ Explorer.
   The Explorer Options window appears.
3. Select the Members tab.
4. Select ITEMSTOR in the Type list.
5. Click Unhide.
   If there is no icon associated with ITEMSTOR in the Type list, then you are prompted to select an icon.
6. Open the Sasuser library in the Explorer window.
8. Select Copy from the pop-up menu and copy the Regstry file. SAS assigns the name Regstry_copy to the file.

Operating Environment Information: You can also use a copy command from your operating environment to make a copy of your registry file for backup purposes. When viewed from outside SAS Explorer, the filename is registry.sas7bitm. Under z/OS, you cannot use the environment copy command to copy your registry file unless your Sasuser library is assigned to an HFS directory.

Using the Registry Editor to Back Up the SAS Registry

Using the Registry Editor to back up the SAS registry is generally the preferred backup method, because it retains any new keys or values in case you must restore the registry from the backup.

To use the Registry Editor to back up the SAS Registry:

1. Open the Registry Editor with the REGEDIT command.
2. Select the top-level key in the left pane of the registry window.
3. From the Registry Editor, select File ⇒ Export Registry File.
   A Save As window appears.
4. Enter a name for your registry backup file in the filename field. (SAS applies the proper file extension name for your operating system.)
5 Click **Save**.

This saves the registry backup file in *Sasuser*. You can control the location of your registry backup file by specifying a different location in the **Save As** window.

---

### Recovering from Registry Failure

This section gives instructions for restoring the registry with a backup file, and shows you how to repair a corrupt registry file.

To install the registry backup file that was created using SAS Explorer or an operating system copy command:

1. Change the name of your corrupt registry file to something else.
2. Rename your backup file to `registry.sas7bitm`, which is the name of your registry file.
3. Copy your renamed registry file to the *Sasuser* location where your previous registry file was located.
4. Restart your SAS session.

To restore a registry backup file created with the Registry Editor:

1. Open the Registry Editor with the **REGEDIT** command.
2. Select **File ➔ Import Registry File**.
3. Select the registry file that you previously exported.
4. Click **Open**.
5. Restart SAS.

To restore a registry backup file created with PROC REGISTRY:

1. Open the Program editor and submit the following program to import the registry file that you created previously.

   ```
   proc registry import=<registry file specification>;
   run;
   ```

   This imports the registry file to the *Sasuser* library.

2. If the file is not already properly named, then use Explorer to rename the registry file to `registry.sas7bitm`:

3. Restart SAS.

To attempt to repair a damaged registry:

1. Rename the damaged registry file to something other than “registry” (for example, `temp`).
2. Start your SAS session.
3. Define a library pointing to the location of the `temp` registry.

   ```
   libname here '.'
   ```
Run the REGISTRY procedure and redefine the Sasuser registry:

```sas
proc registry setsasuser="here.temp";
run;
```

Start the Registry Editor with the REGEDIT command. Select Solutions ⇒ Accessories ⇒ Registry Editor ⇒ View All.

Edit any damaged fields under the HKEY_USER_ROOT key.

Close your SAS session and rename the modified registry back to the original name.

Open a new SAS session to see whether the changes fixed the problem.

---

Using the SAS Registry to Control Color

Overview of Colors and the SAS Registry

The SAS registry contains the RGB values for color names that are common to most web browsers. These colors can be used for ODS and GRAPH output. The RGB value is a triplet (Red, Green, Blue), and each component has a range of 00 to FF (0 to 255).

The registry values for color are located in the COLORNAMES\HTML subkey.

Adding Colors Using the Registry Editor

You can create your own new color values by adding them to the registry in the COLORNAMES\HTML subkey:

1. Open the SAS Registry Editor using the REGEDIT command.
2. Select the COLORNAMES\HTML subkey.
4. Enter the color name in the Value Name field and the RGB value in the Value Data field.
5. Click OK.

Adding Colors Programmatically

You can create your own new color values by adding them to the registry in the COLORNAMES\HTML subkey, using SAS code.

The easiest way is to first write the color values to a file in the layout that the REGISTRY procedure expects. Then you import the file by using the REGISTRY procedure. In this example, Spanish color names are added to the registry.

```sas
filename mycolors temp;
data _null_;    
file "mycolors";
put "[colornames\html]";
```
Using the Registry Editor

When to Use the Registry Editor

The best way to view the contents of the registry is using the Registry Editor. The Registry Editor is a graphical alternative to PROC REGISTRY, an experienced SAS user might use the Registry Editor to do the following:

- View the contents of the registry. The registry shows keys and values stored in keys.
- Add, modify, and delete keys and values stored in the registry.
- Import registry files into the registry, starting at any key.
- Export the contents of the registry to a file, starting at any key.
- Uninstall a registry file.
- Compare a registry file to the SAS registry.

Many of the windows in the SAS windowing environment update the registry for you when you make changes to such items as your printer setting or your color preferences. Because these windows update the registry using the correct syntax and semantics, it is often best to use these alternatives when making adjustments to SAS.

Starting the Registry Editor

To run the Registry Editor, issue the REGEDIT command on a SAS command line. You can also open the registry window by selecting **Solutions ➜ Accessories ➜ Registry Editor**.

Finding Specific Data in the Registry

In the Registry Editor window, double-click a folder icon that contains a registry key. This displays the contents of that key.
Another way to find things is to use the Find utility.

1. From the Registry Editor, select **Edit ➤ Find**.

2. Enter all or part of the text string that you want to find, and click **Options** to specify whether you want to find a **key name**, a **value name**, or **data**.

3. Click **Find**.

*Figure 35.2 The Registry Editor Find Utility*

![Image of Registry Editor Find Utility](image)

---

**Changing a Value in the SAS Registry**

**CAUTION**
Before modifying registry values, always back up the registry.sas7bitm file from Sasuser.

1. In the left pane of the Registry Editor window, click the key that you want to change. The values contained in the key appear in the right pane.

2. Double-click the value.
   - The Registry Editor displays several types of windows, depending on the type of value that you are changing.
Adding a New Value or Key to the SAS Registry

1. In the SAS Registry Editor, right-click the key that you want to add the value to.

2. From the pop-up menu, select the **New** menu item with the type that you want to create.

3. Enter the values for the new key or value in the window that is displayed.
Deleting an Item from the SAS Registry

From the SAS Registry Editor:

1. Right-click the item that you want to delete.
2. Select **Delete** from the pop-up menu.
3. Confirm the deletion.

Renaming an Item in the SAS Registry

From the SAS Registry Editor:

1. Right-click the item that you want to rename.
2. Select **Rename** from the pop-up menu and enter the new name.
3. Click **OK**.

Displaying the Sasuser and Sashelp Registry Items Separately

After you open the Registry Editor, you can change your view from the default. The default view shows the registry contents without regard to the storage location. The other registry view displays both Sasuser and Sashelp items in separate trees in the Registry Editor's left pane.

1. Select **TOOLS** ⇄ **Options** ⇄ **Registry Editor** This opens the **Select Registry View** group box.
2. Select **View All** to display the Sasuser and Sashelp items separately in the Registry Editor's left pane.
   - The Sashelp portion of the registry is listed under the HKEY_SYSTEM_ROOT folder in the left pane.
   - The Sasuser portion of the registry is listed under the HKEY_USER_ROOT folder in the left pane.
Importing a Registry File

You usually import a registry file or SASXREG file when you are restoring a backup registry file. A registry file can contain a complete registry or just part of a registry.

To import a registry file using the SAS Registry Editor:

1. Select File → Import Registry File.
2. In the Open window, select the SASXREG file to import.

*Note:* In order to first create the backup registry file, you can use the REGISTRY Procedure or the Export Registry File menu choice in the Registry Editor.

Exporting a Registry File

You usually export a registry file or SASXREG file, when you are preparing a backup registry file. You can export a complete registry or just part of a registry.

To export a registry file using the SAS Registry Editor:

1. In the left pane of the Registry Editor, select the key that you want to export to a SASXREG file.
   To export the entire registry, select the top key.
2. Select File → Export Registry File.
3. In the Save As window, give the export file a name.
4. Click Save.
Configuring Your Registry

Configuring Universal Printing

Universal Printers should be configured by using either the PRTDEF procedure or the Print Setup window. The REGISTRY procedure can be used to back up a printer definition and to restore a printer definition from a SASXREG file. Any other direct modification of the registry values should be done only under the guidance of SAS Technical Support.

Configuring SAS Explorer

Use the Explorer Options window to configure your Explorer settings. You can also use the Registry Editor to view the current Explorer settings in the SAS registry. The Explorer Options Window is available from the TOOLS ⇒ Options ⇒ Explorer menu when you click on the SAS Explorer window. All the Explorer configuration data is stored in the registry under CORE\Explorer. The following table outlines the location of the most commonly used Explorer configuration data.

Table 35.1  Registry Locations for Commonly Used Explorer Configuration Data

<table>
<thead>
<tr>
<th>Registry Key</th>
<th>What portion of the Explorer it configures</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE\EXPLORER\CONFIGURATION</td>
<td>the portions of the Explorer get initialized at startup.</td>
</tr>
<tr>
<td>CORE\EXPLORER\MENUS</td>
<td>the pop-up menus that are displayed in the Explorer.</td>
</tr>
<tr>
<td>CORE\EXPLORER\KEYEVENTS</td>
<td>the valid key events for the 3270 interface. This key is used only on the mainframe platforms.</td>
</tr>
<tr>
<td>CORE\EXPLORER\ICONS</td>
<td>Which icons to display in the Explorer. If the icon value is −1, this causes the icon to be hidden in the Explorer.</td>
</tr>
<tr>
<td>CORE\EXPLORER\NEW</td>
<td>This subkey controls what types of objects are available from the File ⇒ New menu in the Explorer.</td>
</tr>
</tbody>
</table>
Configuring Libraries and File Shortcuts with the SAS Registry

When you use the New Library window or the File Shortcut Assignment window to create a library reference (libref) or a file reference (fileref), you can click the Enable at startup check box to save the libref or fileref for future use.

When you do this, they are stored in the SAS registry, where it is possible to modify or delete them, as follows:

Deleting an “Enable at startup” library reference
You can use the Registry Editor to delete an “Enable at startup” library reference by deleting the corresponding key under CORE\OPTIONS\LIBNAMES\"your libref". However, it is best to delete your library reference by using the SAS Explorer. This removes this key from the registry when you delete the library reference.

Deleting an “Enable at startup” file shortcut
You can use the Registry Editor to delete an “Enable at startup” file shortcut by deleting the corresponding key under CORE\OPTIONS\FILEREFS\your fileref. However, it is best to delete your library reference by using the SAS Explorer. This removes this key automatically when you delete the file shortcut.

Creating an “Enable at startup” File Shortcut as a site default
A site administrator might want to create a file shortcut that is available to all users at a site. To do this, you first create a version of the file shortcut definition in the Sasuser registry. Then you modify it so that it can be used in the Sashelp registry.

Note: You need special permission to write to the Sashelp part of the SAS registry.

1. Enter the DMFILEASSIGN command.
   This opens the File Shortcut Assignment window.

2. Create the file shortcut that you want to use.

3. Check Enable at startup.

4. Click OK.

5. Verify that the file shortcut was created successfully and enter the REGEDIT command.

6. Find and select the key CORE\OPTIONS\FILEREFS\your fileref.

7. Select File ⇒ Export Registry File and export the file.

8. Edit the exported file and replace all instances of HKEY_USER_ROOT with HKEY_SYSTEM_ROOT.

9. To apply your changes to the site’s Sashelp, use PROC REGISTRY.
   The following code imports the file:
Creating an “Enable at startup” library as a site default

A site administrator might want to create a library that is available to all users at a site. To do this, the Sasuser version of the library definition needs to be migrated to Sashelp.

Note: You need special permission to write to the Sashelp part of the SAS registry.

1. Enter the `dmllibassign` command.
   - This opens the New Library window.

2. Create the library reference that you want to use.

3. Select Enable at startup.

4. Select Enable at startup.

5. Click OK.

6. Issue the `REGEDIT` command after verifying that the library was created successfully.

7. Find and select the registry key `CORE\OPTIONS\LIBNAMES\your libref`.

8. Select File ➔ Export Registry File.
   - The Save As window appears.

9. Select a location to store your registry file.

10. Enter a filename for your registry file in the Filename field.

11. Click Save to export the file.

12. Right-click the file and select Edit in NOTEPAD to edit the file.

13. Edit the exported file and replace all instances of "HKEY_USER_ROOT" with "HKEY_SYSTEM_ROOT".

14. To apply your changes to the site's Sashelp use PROC REGISTRY. The following code imports the file:

   ```sas
   proc registry import=yourfile.sasxreg usesashelp;
   run;
   ```

Fixing Library Reference (Libref) Problems with the SAS Registry

Library references (librefs) are stored in the SAS Registry. You might encounter a situation where a libref fails after it had previously worked. In some situations, editing the registry is the fastest way to fix the problem. This section describes what is involved in repairing a missing or failed libref.
If any permanent libref that is stored in the SAS Registry fails at startup, then the following note appears in the SAS Log:

**NOTE:** One or more library startup assignments were not restored.

The following errors are common causes of library assignment problems:

- Required field values for libref assignment in the SAS Registry are missing.
- Required field values for libref assignment in the SAS Registry are invalid. For example, library names are limited to eight characters, and engine values must match actual engine names.
- Encrypted password data for a libref has changed in the SAS Registry.

**Note:** You can also use the New Library window to add librefs. You can open this window by typing DMLIBASSIGN in the toolbar, or selecting **File ➔ New** from the Explorer window.

---

**CAUTION**

You can correct many libref assignment errors in the SAS Registry Editor. If you are unfamiliar with librefs or the SAS Registry Editor, then ask for technical support. Errors can be made easily in the SAS Registry Editor, and they can prevent your libraries from being assigned at startup.

To correct a libref assignment error using the SAS Registry Editor:

1. Select **Solutions ➔ Accessories ➔ Registry Editor** or issue the **REGEDIT** command to open the Registry Editor.

2. Select one of the following paths, depending on your operating environment, and then make modifications to keys and key values as needed:
   - `CORE\OPTIONS\LIBNAMES`
   - `CORE\OPTIONS\LIBNAMES\CONCATENATED`

**Note:** These corrections are possible only for permanent librefs. That is, those that are created at startup by using the New Library or File Shortcut Assignment window.

For example, if you determine that a key for a permanent, concatenated library has been renamed to something other than a positive whole number, then you can rename that key again so that it is in compliance. Select the key, and then select **Rename** from the pop-up menu to begin the process.
# The SAS Windowing Environment

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>What Is the SAS Windowing Environment?</td>
<td>690</td>
</tr>
<tr>
<td>Main Windows in the SAS Windowing Environment</td>
<td>690</td>
</tr>
<tr>
<td>Overview of SAS Windows</td>
<td>690</td>
</tr>
<tr>
<td>SAS Explorer Window</td>
<td>691</td>
</tr>
<tr>
<td>Enhanced Editor Window</td>
<td>692</td>
</tr>
<tr>
<td>Log Window</td>
<td>693</td>
</tr>
<tr>
<td>Results Window</td>
<td>694</td>
</tr>
<tr>
<td>Output Window</td>
<td>695</td>
</tr>
<tr>
<td>Navigating in the SAS Windowing Environment</td>
<td>698</td>
</tr>
<tr>
<td>Overview of SAS Navigation</td>
<td>698</td>
</tr>
<tr>
<td>Menus in SAS</td>
<td>699</td>
</tr>
<tr>
<td>Toolbars in SAS</td>
<td>702</td>
</tr>
<tr>
<td>The Command Line</td>
<td>702</td>
</tr>
<tr>
<td>Getting Help in SAS</td>
<td>703</td>
</tr>
<tr>
<td>Type Help in the Command Line</td>
<td>703</td>
</tr>
<tr>
<td>Open the Help Menu from the Toolbar</td>
<td>704</td>
</tr>
<tr>
<td>Click Help in Individual SAS Windows</td>
<td>705</td>
</tr>
<tr>
<td>List of SAS Windows and Window Commands</td>
<td>705</td>
</tr>
<tr>
<td>Introduction to Managing Your Data in the SAS Windowing Environment</td>
<td>709</td>
</tr>
<tr>
<td>Managing Data with SAS Explorer</td>
<td>710</td>
</tr>
<tr>
<td>Introduction to Managing Data with SAS Explorer</td>
<td>710</td>
</tr>
<tr>
<td>Viewing Libraries and Data Sets</td>
<td>710</td>
</tr>
<tr>
<td>Assign File Shortcuts</td>
<td>712</td>
</tr>
<tr>
<td>Rename a SAS Data Set</td>
<td>712</td>
</tr>
<tr>
<td>Copy or Duplicate a SAS Data Set</td>
<td>713</td>
</tr>
<tr>
<td>Sorting Data Sets in a Library</td>
<td>713</td>
</tr>
<tr>
<td>View the Properties of a SAS Data Set</td>
<td>714</td>
</tr>
<tr>
<td>Working with VIEWTABLE</td>
<td>715</td>
</tr>
<tr>
<td>Overview of VIEWTABLE</td>
<td>715</td>
</tr>
<tr>
<td>Opening a SAS Data Set in a VIEWTABLE Window</td>
<td>715</td>
</tr>
<tr>
<td>Displaying Table Headers as Names or Labels</td>
<td>716</td>
</tr>
<tr>
<td>Customizing SAS Explorer for Opening the VIEWTABLE Window</td>
<td>717</td>
</tr>
<tr>
<td>Order of Precedence for How Column Headings Are Displayed</td>
<td>719</td>
</tr>
<tr>
<td>Mapping the VIEWTABLE Command to a Function Key</td>
<td>719</td>
</tr>
<tr>
<td>Temporarily Change Column Headings</td>
<td>720</td>
</tr>
<tr>
<td>Move Columns in a Table</td>
<td>721</td>
</tr>
<tr>
<td>Sort by Values of a Column</td>
<td>722</td>
</tr>
<tr>
<td>Edit Cell Values</td>
<td>724</td>
</tr>
<tr>
<td>Subsetting Data By Using the WHERE Expression</td>
<td>725</td>
</tr>
</tbody>
</table>
What Is the SAS Windowing Environment?

SAS provides a graphical user interface that makes SAS easier to use. Collectively, all the windows in SAS are called the SAS windowing environment.

The SAS windowing environment contains the windows that you use to create SAS programs. However, you also find other windows that enable you to manipulate data or change your SAS settings without writing a single line of code.

You might find the SAS windowing environment a convenient alternative to writing a SAS program when you want to work with a SAS data set, or control some aspect of your SAS session.

Main Windows in the SAS Windowing Environment

Overview of SAS Windows

SAS windows have several features that operate in a similar manner across all operating environments: menus, toolbars, and online Help. You can customize many features of the SAS windowing environment, including toolbars, icons, menus, and so on.

The five main windows in the SAS windowing environment are the Explorer, Results, Enhanced Editor, Log, and Output windows.

Note: The arrangement of your SAS windows depends on your operating environment. For example, in the Microsoft Windows operating environment, the Enhanced Editor window appears instead of the Program Editor.

When you first invoke SAS, the Enhanced Editor, Log, Output, and Explorer windows are displayed. When you execute a SAS program, the default output
(HTML) is displayed in the Results window. If you use a PUT statement in your program, then output is written to the SAS Log by default.

**Note:** The Microsoft Windows operating environment was used to create the examples in this section. Menus and toolbars in other operating environments have a similar appearance and behavior.

**Windows Specifics:** If you are using Microsoft Windows, the active window determines which items are available on the main menu bar.

The following display shows one example of the arrangement of SAS windows. The Explorer window shows active libraries.

---

**Figure 36.1  Windows in the SAS Windowing Environment**

---

**SAS Explorer Window**

**Uses of the SAS Explorer Window**

The Explorer window enables you to manage your files in the windowing environment. You can use the SAS Explorer to perform the following tasks:

- View lists of your SAS files.
- Create new SAS files.
View, add, or delete libraries.
Create shortcuts to external files.
Open any SAS file and view its contents.
Move, copy, and delete files.
Open related windows, such as the New Library window.

Open the SAS Explorer Window
You can open SAS Explorer in the following ways:
Command:
Enter EXPLORER in the command line and press Enter.
Menu:
Select View ⇒ Explorer.

Display SAS Explorer with and without a Tree View
You can display the Explorer window with or without a tree view of its contents. Displaying the Explorer with a tree view enables you to view the hierarchy of the files. To display the tree view, select Show Tree from the View menu. To turn tree view off, deselect Show Tree in the menu.

Note: You can resize the Explorer window by dragging an edge or a corner of the window. You can resize the left and right panes of the Explorer window by clicking the split bar between the two panes and dragging it to the right or left.

Enhanced Editor Window

Uses of the Enhanced Editor Window
The Enhanced Editor window enables you to enter, edit, submit, and save SAS programs.

Open the Enhanced Editor Window
To open the Enhanced Editor window select View ⇒ Enhanced Editor from the main menu.

Note: To open your SAS programs in the SAS windowing environment, you can drag and drop them onto the Enhanced Editor window.

View a Program in the Enhanced Editor Window
The following example shows a SAS program in the Enhanced Editor window:
Log Window

Uses of the Log Window

The Log window enables you to view messages about your SAS session and your SAS programs. If the program that you submit has unexpected results, then the log helps you identify the error. You can also use a PUT statement to write program output to the Log.

Note: To keep the lines of your log from wrapping when your window is maximized, use the LINESIZE= system option.

Open the Log Window

You can open the Log window in the following ways:

Command:
Enter LOG in the command line and press Enter.

Menu:
Select View 🔄 Log.
View Log Output

The following is an example of Log output.

Figure 36.3  Example of Output in the Log Window

```
data EmpList;
  input Name $ 1-8 EmpNumber $ 10-13 Division $ 15-25;
  datalines;
NOTE: The data set WORK.EMPLIST has 4 observations and 3 variables.
NOTE: DATA statement used (Total process time):
   real time 1.15 seconds
   cpu time 0.29 seconds

; run;
proc sort data=EmpList;
   by Name;
run;
NOTE: There were 4 observations read from the data set WORK.EMPLIST.
NOTE: The data set WORK.EMPLIST has 4 observations and 3 variables.
NOTE: PROCEDURE SORT used (Total process time):
   real time 0.45 seconds
   cpu time 0.09 seconds

proc print data=EmpList;
NOTE: Writing HTML Body file: esah1.htm
   title 'Employee Listing';
run;
NOTE: There were 4 observations read from the data set WORK.EMPLIST.
NOTE: PROCEDURE PRINT used (Total process time):
   real time 0.43 seconds
   cpu time 0.08 seconds
```

Results Window

Uses of the Results Window

The Results window enables you to view HTML output from a SAS program. HTML is the default output type, and HTMLBlue is the default output style. The Results window uses a tree structure to list various types of output that might be available after you run SAS. You can view, save, or print individual files. The Results window is empty until you execute a SAS program and produce output. When you submit a SAS program, the output is displayed in the Results Viewer and the file is listed in the Results window.

Open the Results Window

You can open the Results window in the following ways:

Command:

Enter ODSRESULTS in the command line and press Enter.
Menu:
Select View → Results.

View Output in the Results Window
The left pane of the following display shows the Results window, and the right pane shows the Results Viewer where the default HTML output is displayed. The Results window lists the files that were created when the SAS program executed.

Figure 36.4  Results Window and Results Viewer

Output Window
Uses of the Output Window
The Output window enables you to view LISTING output from your SAS programs. By default, the Output window is positioned behind the other windows. When you create LISTING output, the Output window automatically moves to the front of your display.

Note: To keep the lines of your output from wrapping when your window is maximized, use the LINESIZE= system option.
Open the Output Window

You can open the Output window in the following ways:

Command:
- Enter OUTPUT or OUT in the command line and press Enter.
- Enter LISTING or LST in the command line and press Enter.

Menu:
- Select View ➔ Output.

Create and View LISTING Output

Because LISTING output is not the default output type, you must use ODS statements to open the LISTING destination. Along with LISTING output, HTML output is also generated.

The following example shows a program that produces LISTING output. There is an ODS statement before the DATA statement and after the RUN statement:

Figure 36.5 Example of a Program That Produces Listing Output

```
options page=1 nodate;
ods listing;
  data EmpList;
    input Name $ 1-8 EmpNumber $ 10-13 Division $ 15-25;
    datalines;
    Jacobson 4355 Agriculture
    Lincoln 3422 Chemistry
    Demara 4599 Physics
    Wojtyla 5593 Ethics
  ;
  proc sort data=EmpList;
    by Name;
  run;
  proc print data=EmpList;
    title 'Employee Listing';
  run;
  ods listing close;
```

SAS creates the following LISTING output:
Using the Preferences Dialog Box to Select Output Types

You can use the Preferences dialog box to select output types and set system preferences. Each tab in the Preferences dialog box holds a related group of items. To access the Preferences dialog box, select **Tools** ⇒ **Options** ⇒ **Preferences**.

The following is an example of the Preferences dialog box, with the **Results** tab selected:
Several default values are selected in the Results tab. Under HTML, Create HTML is the default output type, and HTMLBlue is the default output style. Use ODS Graphics is also selected by default. When the Use ODS Graphics box is checked, you are able to automatically generate graphs when running procedures that support ODS graphics. Checking or unchecking this box enables you to turn on or turn off ODS graphics when you invoke SAS.

To produce LISTING output, check the Create listing box under Listing. If you deselect Create HTML and leave the Create listing box checked, your program produces listing output only.

Navigating in the SAS Windowing Environment

Overview of SAS Navigation

SAS windows have several features that work in a similar manner across all operating environments: menus, toolbars, and online Help. You can customize many of these features by selecting Tools ⇒ Customize from the menu. For specific information about these features, see the documentation for your operating environment.
Menus in SAS

Menus contain lists of options that you can select.

The following example shows the menu options that are available when you select Help from the menu bar:

**Figure 36.8  The Help Menu**

Menu choices change as you change the windows that you are using. For example, if you select Explorer from the View menu, and then select View again, the menu lists the View options that are available when the Explorer window is active.

The following display shows the View menu when the Explorer window is active:
If you select **Program Editor** from the **View** menu, and then select **View** again, the menu lists the **View** options that are available when the Program Editor window is active.

The following display shows the **View** menu when the Program Editor window is active:
You can also access menus when you right-click an item. For example, when you select `View ⇒ Explorer` and then right-click `Libraries` in the Explorer window, the following menu appears:
The menu remains visible until you make a selection from the menu or until you click an area outside of the menu area.

Toolbars in SAS

A toolbar displays a block of window buttons or icons. When you click items in the toolbar, a function or an action is started. For example, clicking a picture of a printer in a toolbar starts a print process. The toolbar displays icons for many of the actions that you perform most often in a particular window.

z/OS Specifics: SAS in the z/OS operating environment does not have a toolbar. See SAS Companion for z/OS for more information.

The toolbar that you see depends on which window is active. For example, when the Program Editor window is active, the following toolbar is displayed:

*Figure 36.12  Example of the SAS Toolbar When the Enhanced Editor Window Is Active*

![Toolbar Example](image)

When you position your cursor at one of the items in the toolbar, a text window appears that identifies the purpose of the icon.

The Command Line

The command line is located to the left of the toolbar. In the command line, you can enter commands, such as those that open SAS windows and those that retrieve help information.

The following is an example of a command that opens the SASCOLOR window:

*Figure 36.13  Example of the Command Line*

![Command Line Example](image)
Getting Help in SAS

Type Help in the Command Line

When you enter Help in the command line, help for the active window is displayed. When you enter Help <item> (for example, Help footnote), you can access help for the item that you entered. The following window is displayed when you enter Help footnote in the command line of a SAS session:
Open the Help Menu from the Toolbar

When you open the Help menu, you can select from the following choices:

Using This Window
opens a Help system window that describes the current active window.

SAS Help and Documentation
opens the SAS Help and Documentation system. Help is available for Base SAS and other SAS products that are installed on your system. You can find information by clicking an item in the table of contents or by searching for the item and then clicking the results.

Getting Started with SAS Software
opens the Getting Started with SAS tutorial. This is a good way to learn the basics of how to use SAS.

Learning SAS Programming
enables you to use SAS online training if you have an online training license. The software provides 50–60 hours of instruction for beginning as well as experienced SAS programmers.

SAS on the web
provides links to the SAS website where you can do the following:
Click Help in Individual SAS Windows

When you open a SAS window, you can press the HELP key (usually F1) from your keyboard to display information about that window.

List of SAS Windows and Window Commands

The basic SAS windows consist of the Explorer, Results, Program Editor, Enhanced Editor (Windows operating environment), Log, and Output windows. However, there are more than 30 other windows to help you with such tasks as printing and fine-tuning your SAS session.

The following table lists all portable SAS windows, window descriptions, and the commands that open the windows.

Table 36.1  List of SAS Windows, Descriptions, and Window Commands

<table>
<thead>
<tr>
<th>Window Name</th>
<th>Description</th>
<th>Window Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documents</td>
<td>Displays your ODS documents in a hierarchical tree structure.</td>
<td>ODSDOCUMENTS</td>
</tr>
<tr>
<td>Edit Scheme</td>
<td>Enables you to change the default colors in edit windows.</td>
<td>SYNCONFIG</td>
</tr>
<tr>
<td>Explorer</td>
<td>Provides a central access point to data such as catalogs, libraries, data sets, and host files.</td>
<td>ACCESS, BUILD, CATALOG, DIR, EXPLORER, FILENAME, LIBNAME, V6CAT, V6DIR, V6FILENAME, V6LIBNAME</td>
</tr>
<tr>
<td>Window Name</td>
<td>Description</td>
<td>Window Commands</td>
</tr>
<tr>
<td>------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Explorer Options</td>
<td>Enables you to add or delete file types, change or add pop-up menu items, select folders that appear in the Explorer, and display member details.</td>
<td>DMEXPOPTS</td>
</tr>
<tr>
<td>File Shortcut Assignment</td>
<td>Assigns a file shortcut to a file using a graphical user interface.</td>
<td>DMFILEASSIGN</td>
</tr>
<tr>
<td>Find</td>
<td>Enables you to search for an expression in a SAS library.</td>
<td>EXPFIND</td>
</tr>
<tr>
<td>Select Font</td>
<td>Enables you to select a font, font style, and font size.</td>
<td>DLGFONT</td>
</tr>
<tr>
<td>(operating-environment specific)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOOTNOTES</td>
<td>Enables you to enter, browse, and modify footnotes for output.</td>
<td>FOOTNOTES</td>
</tr>
<tr>
<td>FSBROWSE</td>
<td>Enables you to select a data set for browsing.</td>
<td>FSBROWSE</td>
</tr>
<tr>
<td>FSEDIT</td>
<td>Enables you to select a data set to be processed by the FSEDIT procedure.</td>
<td>FSEDIT</td>
</tr>
<tr>
<td>FSFORM</td>
<td>Enables you to customize a form for sending output to the printer.</td>
<td>FSFORM formname</td>
</tr>
<tr>
<td>FSLETTER</td>
<td>Enables you to edit or create catalog entries.</td>
<td>FSLETTER</td>
</tr>
<tr>
<td>FSLIST</td>
<td>Enables you to browse external files in a SAS session.</td>
<td>FSLIST</td>
</tr>
<tr>
<td>FSVIEW</td>
<td>Enables you to browse, edit, or create a SAS data set, displaying the data set as a table with rows and columns.</td>
<td>FSVIEW</td>
</tr>
<tr>
<td>HELP</td>
<td>Displays help information about SAS.</td>
<td>HELP</td>
</tr>
<tr>
<td>Window Name</td>
<td>Description</td>
<td>Window Commands</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>KEYS</td>
<td>Enables you to browse, alter, and save function key settings.</td>
<td>KEYS</td>
</tr>
<tr>
<td>Log</td>
<td>Displays messages and SAS statements for the current SAS session.</td>
<td>LOG</td>
</tr>
<tr>
<td>Metabase</td>
<td>Accesses the SAS/EIS Metabase Facility to register data, to copy data registrations, and to create, delete, or edit repository files.</td>
<td>METABASE</td>
</tr>
<tr>
<td>Metadata Browser</td>
<td>Opens the Metadata Server Configuration dialog box.</td>
<td>METABROWSE</td>
</tr>
<tr>
<td>Metafind</td>
<td>Enables you to search for metadata objects in repositories by using Uniform Resource Identifiers (URIs).</td>
<td>METAEXPAND</td>
</tr>
<tr>
<td>Metadata Server Connections</td>
<td>Enables you to import, export, add, remove, reorder, and test metadata server connections.</td>
<td>METACON</td>
</tr>
<tr>
<td>New Library</td>
<td>Enables you to create a new SAS library and assign a libref.</td>
<td>DMLIBASSIGN</td>
</tr>
<tr>
<td>NOTEPAD</td>
<td>Enables you to create and store notepads of text.</td>
<td>NOTEPAD, NOTE, FILEPAD filename</td>
</tr>
<tr>
<td>Options (SAS system options)</td>
<td>Enables you to view and change some SAS system options.</td>
<td>OPTIONS</td>
</tr>
<tr>
<td>Output</td>
<td>Displays procedure output in listing format.</td>
<td>OUTPUT, OUT, LISTING, LIST, LST</td>
</tr>
<tr>
<td>Page Setup</td>
<td>Enables you to specify page setup options that apply to Universal Printing jobs.</td>
<td>DMPAGESETUP</td>
</tr>
<tr>
<td>Password</td>
<td>Enables you to edit, assign, or clear passwords for a particular data set.</td>
<td>SETPASSWORD</td>
</tr>
</tbody>
</table>

(not available on z/OS)
<table>
<thead>
<tr>
<th>Window Name</th>
<th>Description</th>
<th>Window Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preferences</td>
<td>Enables you to set or edit SAS system preferences.</td>
<td>DLGPREF</td>
</tr>
<tr>
<td>(operating-environment specific)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Print</td>
<td>Enables you to print the content of an active SAS window through Universal Printing.</td>
<td>DMPRINT</td>
</tr>
<tr>
<td>Print Setup</td>
<td>Enables you to change your default printer, create or edit a printer definition, or delete a printer definition for Universal Printing.</td>
<td>DMPRTSETUP</td>
</tr>
<tr>
<td>Program Editor</td>
<td>Enables you to enter, edit, and submit SAS statements and save source files.</td>
<td>PROGRAM, PGM</td>
</tr>
<tr>
<td>Properties</td>
<td>Shows details that are associated with the current data set.</td>
<td>VAR libref.SAS-data-set, V6VAR libref.SAS-data-set</td>
</tr>
<tr>
<td>SAS Registry Editor</td>
<td>Enables you to edit the SAS registry and to customize aspects of the SAS windowing environment.</td>
<td>REGEDIT</td>
</tr>
<tr>
<td>Results</td>
<td>Lists the procedure output that is produced by SAS.</td>
<td>ODSRESULTS</td>
</tr>
<tr>
<td>SAS/ACCESS</td>
<td></td>
<td>ACCESS</td>
</tr>
<tr>
<td>SAS/AF</td>
<td>Displays windowing applications that are created by SAS/AF software.</td>
<td>AF, AFA</td>
</tr>
<tr>
<td>SAS/ASSIST</td>
<td>Displays the primary menu of SAS/ASSIST software, which simplifies the use of SAS.</td>
<td>ASSIST</td>
</tr>
<tr>
<td>SASCOLOR</td>
<td>Enables you to change default colors for the different window elements in your SAS windows.</td>
<td>SASCOLOR</td>
</tr>
</tbody>
</table>
### Introduction to Managing Your Data in the SAS Windowing Environment

The SAS windowing environment contains windows that enable you to perform common data manipulation and make changes without writing code.

If you are not familiar with SAS or with writing code in the SAS language, then you might find the windowing environment helpful. With the windowing environment, you can open a data set, point to rows and columns in your data. Then, you can click menu items to reorganize and perform analyses on the information.

For more information about the SAS windowing environment, select **SAS Help and Documentation** from the Help menu after you invoke a SAS session.

---

<table>
<thead>
<tr>
<th>Window Name</th>
<th>Description</th>
<th>Window Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL QUERY</td>
<td>Enables you to build, run, and save queries without being familiar with Structured Query Language (SQL).</td>
<td>QUERY</td>
</tr>
<tr>
<td>SAS System Options</td>
<td>Enables you to change SAS system option settings.</td>
<td>OPTIONS</td>
</tr>
<tr>
<td>Templates</td>
<td>Enables you to browse and edit template source code.</td>
<td>ODSTEMPLATES</td>
</tr>
<tr>
<td>TITLES</td>
<td>Enables you to enter, browse, and modify titles for output.</td>
<td>TITLES</td>
</tr>
<tr>
<td>VIEWTABLE</td>
<td>Enables you to browse, edit, or create tables (data sets).</td>
<td>VIEWTABLE, VT</td>
</tr>
</tbody>
</table>

---

*Note: Some additional SAS windows that are specific to your operating environment might also be available. For more information, see the SAS documentation for your operating environment.*
Managing Data with SAS Explorer

Introduction to Managing Data with SAS Explorer

You can use SAS Explorer to view and manage data sets. Data sets are stored in libraries, which are storage locations for SAS files and catalogs. By default, SAS defines several libraries for you:

Sashelp
is a library created by SAS that stores the text for Help windows, default function-key definitions, window definitions, and menus.

Maps
is a library created by SAS that presents graphical representations of geographical or other areas.

Sasuser
is a permanent SAS library that is created at the beginning of your first SAS session. This library contains a Profile catalog that stores the customized features or settings that you specify for SAS. (You can store other SAS files in this library.)

Work
is a library that is created by SAS at the beginning of each SAS session or SAS job. Unless you have specified a User library, any newly created SAS file with a one-level name is placed in the Work library by default. The newly created file is deleted at the end of the current session or job.

Viewing Libraries and Data Sets

Libraries and data sets are represented in SAS by large icons, small icons, or as a list. With the Explorer window active, you can change this representation by selecting an option from the View menu:

- To view large icons, select **Large Icons** from the **View** menu.
- To view small icons, select **Small Icons** from the **View** menu.
- To view data sets in a list, select **List** from the **View** menu.

The following example uses large icons to show the contents of Sashelp:
If you select the Sashelp library and then select **View → Details** from the menu bar, the contents of the Sashelp library is displayed, along with the size and type of the data sets:

If you double-click a table in this list, the data set opens. The VIEWTABLE window, which is a SAS table viewer and editor, appears and is populated with the data from the table.
Assign File Shortcuts

A file shortcut is also known as a file reference or fileref. Filerefs save you programming time by enabling you to assign a nickname to a commonly used file. You can use the FILENAME statement to create a fileref, or you can use the File Shortcut Assignment window from SAS Explorer.

To assign a fileref to a file, follow these steps:

1. Select Tools ➔ New File Shortcut from the menu.
2. In the File Shortcut Assignment window that appears, enter the name of the fileref that you want to use in the Name field.
3. Enter the full pathname for the file in the File field.

The following display shows the File Shortcut Assignment window:

By default, filerefs that you create are temporary and can be used in the current SAS session only. Selecting Enable at Start-up from the File Shortcut Assignment window, however, assigns the fileref to the file whenever you start a new SAS session.

Rename a SAS Data Set

You can rename any data set in a SAS library as long as it is not Write protected. To rename a data set, follow these steps:
1 Open SAS Explorer and select a library.
The contents of the library appear in the right pane.

2 Right-click the data set that you want to rename.

3 Select **Rename** from the menu, and enter the new name of the data set.

4 Click **OK**.

---

**Copy or Duplicate a SAS Data Set**

You can copy a SAS data set to another library or catalog, or you can duplicate the data set in the same directory as the original data set. To copy or duplicate a data set, follow these steps:

1 Open SAS Explorer and select a library.
The contents of the library appear in the right pane.

2 Right-click the data set you want to copy or duplicate.

3 From the menu that is displayed, choose **Copy** to copy a data set to another library or catalog, or choose **Duplicate** to copy the data set to the same library or catalog.

4 If you choose **Copy**, do the following:
   a Click the library in the left pane of SAS Explorer to select the library or catalog into which the data set will be copied.
   b In the right pane, right-click the mouse and select **Paste** from the menu that appears.
      A copy of the data set now resides in the new directory.

5 If you choose **Duplicate**, then the Duplicate window appears. In the Duplicate window, SAS appends `_copy` to the data set name (for example, `data-set-name_copy`).
   Do one of the following:
   - Keep the name and click **OK**.
   - Create another name for your duplicated data set and click **OK**.

---

**Sorting Data Sets in a Library**

Data sets in SAS Explorer are sorted automatically by name. You can change the sort order of the data sets by size or type by clicking the **Size** or **Type** column. To return data sets to their original order, select the **Refresh** option from the **View** menu.
View the Properties of a SAS Data Set

You can view the properties of a data set by using the Properties window. To view properties, follow these steps:

1. Open SAS Explorer and select a library.
   The contents of the library appears in the right pane.

2. Right-click the data set that you want to view.

3. Select Properties from the menu.
   The following window appears for the Air data set:

4. In the Description field of the General tab, you can enter a description of the data set. To save the description, click OK.

5. Select other tabs to display additional information about the data set.
Working with VIEWTABLE

Overview of VIEWTABLE

To manipulate data interactively, you can use the SAS table editor, VIEWTABLE. In the VIEWTABLE window, you can create a new table, and view or edit an existing table.

Opening a SAS Data Set in a VIEWTABLE Window

Using the SAS Explorer Window

You can open an existing SAS data set in a VIEWTABLE window by double-clicking on the SAS data set icon in the SAS Explorer window, or by VIEWTABLE Command in the SAS Display Manager command line.

Here are the steps for using the SAS Explorer window to open a SAS data set in a VIEWTABLE window:

1. Open SAS Explorer and double-click on the icon for the library that contains the target data set.
2. Select the desired data set and double-click on its icon.
3. The VIEWTABLE window should appear, populated with data from the data set.
Use the scroll bar on the VIEWTABLE window to view all of the data.

Using the VIEWTABLE Command

You can also open a data set in a VIEWTABLE window by using the VIEWTABLE command in the SAS Display Manager command line.

1 Specify the VIEWTABLE command in the SAS Display Manager command line using the following syntax:

```
VIEWTABLE data-set-name <-options>
```

2 Here is an example:

```sas
table cars
```

Displaying Table Headers as Names or Labels

When you open a data set that contains labels in a VIEWTABLE window, SAS automatically displays the table headers as variable labels rather than the variable names. You can change the way SAS displays table headers by using the VIEWTABLE pop-up menu or by using the VIEWTABLE command.

- Using the VIEWTABLE pop-up menu to change the way table headers are displayed:
1. Open a data set in VIEWTABLE (to access the VIEWTABLE pop-up menu, you must have an active VIEWTABLE window open).

2. Make sure that the VIEWTABLE window is active.

3. Select View → Column Names or View → Column Labels from the drop-down View menu.

4. Once this selection is made, the opened table, and all tables that are subsequently opened, will display table headers based on this setting in the VIEWTABLE pop-up menu. When you exit VIEWTABLE, or exit SAS, the preference for column labels or column names is saved. When you open VIEWTABLE or invoke SAS again, the preference that you chose is automatically selected.

This feature is available in SAS 9.4M1 and later releases.

Using the VIEWTABLE command to change the way table headers are displayed when a table is opened:

1. Specify the COLHEADING= option on the VIEWTABLE command in the SAS command line using the following syntax.

   ```sas
   VIEWTABLE data-set-name -<COLHEADING>=NAMES | LABELS>
   ```

2. Here is an example:

   ```sas
   viewtable cars colheading=names
   ```

Customizing SAS Explorer for Opening the VIEWTABLE Window

You can customize SAS Explorer to open a VIEWTABLE window so that column headings are displayed as either names or labels every time that the table is opened from the SAS Explorer window. To do this, add the COLHEADING= option to the Action Command in the SAS Explorer Options dialog box.

1. With the SAS Explorer window active, select Tools → Options → Explorer to open the Explorer Options window.
2. Select the **Members** tab.

3. Select **Table** in the list of registered types, and then click **Edit** to open the TABLE Options dialog box.

4. Select the &Open Action Command in the list of actions, and then click **Edit** to open the Edit Action dialog box.

5. In the Edit Action dialog box, add `-COLHEADING=<value>` to the end of the VIEWTABLE command:

   ```
   VIEWTABLE %8b. '%s'.DATA colheading=names
   ```

6. When you are finished making changes, click OK three times to exit all of the open dialog boxes. From this point on, when you use the SAS Explorer Window...
to open the VIEWTABLE window, SAS displays the table headers according to what you specified in this SAS Explorer dialog box.

**Note:** These steps only affect how tables are displayed when they are opened from the SAS Explorer Window (either by double-clicking on the icon or by right-clicking on the icon and selecting "Open"). They do not affect how tables are opened when you use the VIEWTABLE command to open a table.

---

### Order of Precedence for How Column Headings Are Displayed

If you open a table using the VIEWTABLE command and you do not specify COLHEADING= to control how column headings should be displayed, then SAS will display column headings based on how they were last set in the VIEWTABLE pop-up menu (View ⇒ Column Names or View ⇒ Column Labels).

If you open a table using the VIEWTABLE colheading=＜value＞ command, SAS will display the column headings according to the COLHEADING value, regardless of how column headings are set in the VIEWTABLE pop-up menu. The setting in the VIEWTABLE pop-up menu will reflect the COLHEADING= value. In other words, COLHEADING= overrides the setting specified in the VIEWTABLE pop-up menu.

For information about the LABEL statement in SAS, see "LABEL Statement" in *SAS DATA Step Statements: Reference*.

---

### Mapping the VIEWTABLE Command to a Function Key

You can map a Function Key in the Display Manager Keys window to execute the VIEWTABLE command. To do this, follow these steps:

1. Select **Tools ⇒ Options ⇒ Keys** from the SAS menu. The Keys window will appear.

2. In the Keys window, select the F-Key that you want to assign to the VIEWTABLE command and place the cursor in the Definition field of the selected F-Key.

3. Type the VIEWTABLE command with the desired option. Here is an example:
   ```
   VIEWTABLE &8b. '8s'.DATA colheading=name
   ```

4. Close the Keys window.
For more information about using VIEWTABLE, see Doing More with the SAS®
Display Manager: From Editor to ViewTable - Options and Tools You Should Know
(PDF).

Temporarily Change Column Headings

Within the VIEWTABLE window, you can temporarily change column headings. To
temporarily change column headings, follow these steps:

1. Right-click the heading for the column that you want to change, and then select
   Column Attributes from the menu.

2. In the Label field of the Column Attributes window, enter the new name of the
   column heading and then click Apply.

   In this example, the Name heading is replaced by the Name of Player label.
When you press **Apply**, the column heading in **VIEWTABLE** changes to the new name.

In this example, the label was changed to **Name of Player**.

3  Click **Close** to close the Column Attributes window.

**Move Columns in a Table**

Within the **VIEWTABLE** window, you can rearrange columns in your table. To move columns in your table, follow these steps:

1  Click a column heading for the column that you want to move.

2  Drag and drop the heading onto another column heading.

In this example, if you click the heading **Name**, and then drag and drop **Name** onto **Team at the End of 1986**, the **Name** column moves to the right of the **Team at the End of 1986** column.
Sort by Values of a Column

You can sort your table in ascending or descending order, based on the values in a column. You can sort data permanently or create a sorted copy of your table.

To sort your table, follow these steps:

1. Right-click the heading of the column on which you want to sort, and select Sort from the menu.
2. Select Ascending or Descending from the menu.
3. When the following warning message appears, click Yes to create a sorted copy of the table.

Note: If you selected Edit Mode after opening the table and clicking a data cell, this window does not appear. SAS updates the original table.
4. In the Sort window, enter the name of the new sorted table.
   In this example, the name of the sorted table is **BaseballStatisticsList**.

5. Click **OK**.
   The rows in the new table are sorted in ascending order by values of **Team at the End of 1986**.
Edit Cell Values

By default, VIEWTABLE opens existing tables in browse mode, which protects the table data. To edit the table, you need to switch to Edit mode. To switch to Edit mode and edit a table cell, follow these steps:

1. With the table open, select Edit ➦ Edit Mode from the Edit menu.
2. Click a cell in the table, and the value in the cell is highlighted. In this example, the third cell in the fifth row is highlighted.
3. Enter a new value in the cell and press Enter. In this example, the cell has been updated with a new value for Times at Bat in 1986.
4. Select File ➦ Close from the File menu.
When prompted to save pending changes to the table, click Yes to save your changes or No to disregard changes.

**Note:** If you make changes in one row and then edit cells in another row, the changes in the first row are automatically saved. When you select File \(\Rightarrow\) Close, you are prompted to save the pending changes to the second row.

---

**Subsetting Data By Using the WHERE Expression**

**Subset Rows of a Table**

In the VIEWTABLE window, you can subset the display to show only those rows that meet one or more conditions. To subset rows of a table, follow these steps:

1. In the Explorer window, open a library and double-click the table that you want to subset.

   In this example, the Cars data table is selected.

2. Right-click any table cell that is not a heading and select Where from the menu.

   The WHERE EXPRESSION window appears.
3 In the **Available Columns** list, select a column, and then select an operator from the **Operators** menu.

In this example, **Make** is selected from the **Available Columns** list, and **EQ** (equal to) is selected from the **Operators** menu. Note that the WHERE expression is being built in the **Where** box at the bottom of the window.

4 In the **Available Columns** list, select another value to complete the WHERE expression.

In this example, scroll to the bottom of the **Available Columns** window and select `<LOOKUP distinct values>`.

5 In the Lookup Distinct Values window that appears, select a value.

In this example, **Honda** is selected.
Note that the complete WHERE expression appears in the **Where** box at the bottom of the window.

6. Click **OK** to close the WHERE EXPRESSION window.

In this example, VIEWTABLE displays only rows where the value of **Make** is Honda.
Clear the WHERE Expression

You can clear the WHERE expression that you used to subset your data, and redisplay all of the data in the table. To do this, follow these steps:

1. Right-click anywhere in the table except in a column heading.
2. Select WHERE Clear from the menu.

The VIEWTABLE window removes any existing subsets of data that were created with the WHERE expression, and displays all of the rows of the table.

Exporting a Subset of Data

Overview of Exporting Data

The Export Wizard reads data from a SAS data set and writes it to an external file. You can export SAS data to a variety of formats. The formats that are available depend on your operating environment and the SAS products that you have installed.

Export Data

To export data, follow these steps:

1. With the Explorer window active, select File ➔ Export Data.
The Export Wizard - Select library and member window appears.

2 Select the SAS data set from which you want to export data.

In this example, Sashelp is selected as the library, and Cars is the member name.

3 Click Next and the Export Wizard - Select export type window appears.

4 Select the type of data source to which you want to export files.

In this example, Microsoft Excel Workbook is selected. Note that Standard data source is selected by default.

5 Click Next to display the Connect to MS Excel window.

6 In the Workbook field, enter the name of the workbook that will contain the exported file and then click OK.
In this example, **Myworkbook** is entered as the name of the workbook.

![Connect to MS Excel](image1)

7 When the Export Wizard - Select table window appears, enter a name for the table that you are exporting.

In this example, **Mytable** is the table name.

![Export Wizard - Select table](image2)

8 Click **Next**.

9 If you want SAS to create a file of PROC EXPORT statements for later use, then enter the name of the file that will contain the SAS statements.

In this example, PROC EXPORT statements are saved to the file. The **Replace file if it exists** box is checked.
Importing Data into a Table

Overview of Importing Data

Whether your data is stored in a standard file format or in your own special file format, you can use the Import Wizard to import data into a SAS table. The types of files that you can import depend on your operating environment.

Import a Standard File

To import a standard file, follow these steps:

1. With the Explorer window active, select File ➔ Import data.

   The Import Wizard - Select import type window appears.

2. Select the type of file that you are importing by selecting a data source from the Select a data source menu.

   Note that Standard data source is selected by default. In this example, Microsoft Excel Workbook is selected.

10 Click Finish to complete this task.
3 Click **Next** to continue.

4 In the Connect to MS Excel window, enter the pathname of the file that you want to export, and then click **OK**.

5 In the Import Wizard - Select table window, enter the name of the table that you want to import.
6 Click **Next** to continue.

7 In the Import Wizard - Select library and member window, enter a location in which to store the imported file. In this example, **Work** is selected as the library, and **Book1** is selected as the member name.

8 Click **Next** to continue.

9 If you want SAS to create a file of PROC IMPORT statements for later use, then enter the name of a file that will contain the SAS statements.
10 Click Finish to complete this task.

Import a Nonstandard File

If your data is not in standard format, you can use the External File Interface (EFI) facility to import data. This tool enables you to define your file format and offers you a range of format options. To use EFI, select User-defined file format in the Import Wizard and follow the directions for describing your data file.
What is SAS Cloud Analytic Services?

SAS Cloud Analytic Services (CAS) is a server that provides the cloud-based runtime environment for data management and analytics with SAS. CAS is part of the SAS Viya platform, an open, cloud-enabled platform that supports high-performance analytics. A SAS Viya license is required for access to SAS Cloud Analytic Services.

“What Does This Mean for the SAS 9 Programmer?”

What Does This Mean for the SAS 9 Programmer?

- When you license SAS Viya, you can write programs in SAS 9.4 in your SAS 9.4 environment and submit them to CAS for processing. This means faster processing and faster results.
- New and existing SAS 9.4 programs can be submitted to CAS from the SAS Windowing environment (SAS Display Manager) or from SAS Studio.
- SAS Viya is not a replacement for SAS 9.4. It is a platform designed to work with SAS 9.4 and other languages such as Java, Python, Lua, and R.
SAS Language Elements for CAS

DATA Step Processing

The DATA step and most of the language elements that run in the DATA step are supported for processing in CAS. The DATA step runs in multiple threads in CAS, which means that processing is faster.

For information about DATA step processing in CAS, see SAS Cloud Analytic Services: DATA Step Programming.

DATA Step Language Elements for CAS

Many SAS language elements, including the new CAS engine LIBNAME statement, have been enhanced to provide access to CAS via the SAS DATA step. The CAS engine serves as the bridge between your SAS 9.4 programs and the CAS server. For more information about DATA step processing in CAS, see SAS Cloud Analytic Services: DATA Step Programming.

Not all SAS language elements are supported for DATA step processing in CAS. Language elements that are not supported in CAS are marked in the documentation with a “Restriction” as shown in the following image:

Figure 37.1  Example Documentation Syntax Page That Shows a “Restriction” to Indicate That the Language Element Is Not Supported in CAS

<table>
<thead>
<tr>
<th>ADDR Function</th>
<th>ADDR Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR Function</td>
<td>Returns the memory address of a variable on a 32-bit platform.</td>
</tr>
<tr>
<td>ADDRLONG Function</td>
<td>Category: Special</td>
</tr>
<tr>
<td>AIRY Function</td>
<td>Restrictions: Use on 32-bit platforms only.</td>
</tr>
<tr>
<td>ALLCOMB Function</td>
<td>This function is not valid on the CAS server.</td>
</tr>
</tbody>
</table>

SAS language elements that are supported in CAS display “CAS” in the Categories field of the syntax page for the language element:
Each language element dictionary also contains a summary table of CAS-supported language elements:

Here is a list of category tables for each of the SAS language element types:

- DATA Step Statements By Category in SAS DATA Step Statements: Reference
- Global Statements by Category in SAS Global Statements: Reference
- Functions and CALL Routines By Category in SAS Functions and CALL Routines: Reference
- Formats By Category in SAS Formats and Informats: Reference
- Informats by Category in SAS Formats and Informats: Reference
- Data Set Options By Category in SAS Data Set Options: Reference

SAS Procedures for CAS

Like DATA step language elements, SAS procedures can interact with or run in CAS by using the CAS LIBNAME engine. For information about the Base SAS procedures that are supported by CAS, see SAS Viya Foundation Procedures in An Introduction to SAS Viya Programming.
CAS-specific Language Elements

CAS-specific SAS language elements are designed specifically for interfacing with the CAS server and can be used only in a CAS server environment.

For information about these language elements, see the following CAS documentation:

- CAS Conceptual Information: *SAS Cloud Analytic Services: Fundamentals*
- Introduction for SAS 9 programmers: *An Introduction to SAS Viya Programming*

---

**Note:** A SAS Viya Visual Analytics license is required for access to SAS Cloud Analytic Services.
## Sending E-Mail through SMTP

You can send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) e-mail interface. SMTP is available for all operating environments in which SAS runs. To send SMTP e-mail with SAS e-mail support, you must have an intranet or Internet connection that supports SMTP.

Some SMTP servers require just the user identification as the login ID while others require the full e-mail address. The SAS SMTP e-mail interface authenticates the user identification in the following order.

### System Options That Control SMTP E-mail

- FILENAME Statement
- FILE and PUT Statements

### Statements That Control SMTP E-mail

- FILENAME Statement
- FILE and PUT Statements

### Universally Unique Identifiers and the Object Spawner

- What Is a Universally Unique Identifier?
- What Is the Object Spawner?
- Defining the UUID Generator Daemon
- Installing the UUID Generator Daemon

### Using SAS Language Elements to Assign UUIDs

- Overview of Using SAS Language Elements to Assign UUIDs
- UUIDGEN Function
- UUIDCOUNT= System Option
- UUIDGENDHOST System Option

### Overview of IPv6

- IPv6 Address Format
- Examples of IPv6 Addresses
- Example of Full and Collapsed IPv6 Address
- Example of an IPv6 Address That Includes a Port Number
- Example of an IPv6 Address That Includes a URL

### Fully Qualified Domain Names (FQDN)

### Sending E-Mail through SMTP

You can send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) e-mail interface. SMTP is available for all operating environments in which SAS runs. To send SMTP e-mail with SAS e-mail support, you must have an intranet or Internet connection that supports SMTP.

Some SMTP servers require just the user identification as the login ID while others require the full e-mail address. The SAS SMTP e-mail interface authenticates the user identification in the following order.
1. If the user ID is specified by the USERID= option in the EMAILHOST= system option, the SAS SMTP e-mail interface attempts to authenticate by using this user ID.

2. If the user ID is not specified by the USERID= option, the SAS SMTP e-mail interface attempts to authenticate by using the user ID specified by the FROM= option of the FILENAME= statement.

3. If the user ID is not specified in the FROM= option in the FILENAME= statement, the SAS SMTP e-mail interface attempts to authenticate by using the user ID specified by the EMAILID= system option.

4. If the user ID is not specified by the EMAILID= system option, the SAS SMTP e-mail interface looks up the user ID from the operating system and attempts to authenticate that user ID.

For more information about sending e-mail from SAS, see the SAS documentation for your operating environment.

### System Options That Control SMTP E-Mail

Several SAS system options control SMTP e-mail. Depending on your operating environment and whether the SMTP e-mail interface is supported at your site, you might need to specify these options at start-up or in your SAS configuration file.

**Operating Environment Information:** To determine the default e-mail interface for your operating environment and to determine the correct syntax for setting system options, see the SAS documentation for your operating environment.

The EMAILSYS system option specifies which e-mail system to use for sending electronic mail from within SAS. For more information about the EMAILSYS system option, see the SAS documentation for your operating environment.

The following system options are specified only when the SMTP e-mail interface is supported at your site:

- **EMAILACKWAIT=**
  
  Specifies the number of seconds that SAS will wait to receive an acknowledgment from an SMTP server. For more information, see “EMAILACKWAIT= System Option” in SAS System Options: Reference.

- **EMAILAUTHPROTOCOL=**
  
  Specifies the authentication protocol for SMTP E-mail. For more information, see the “EMAILAUTHPROTOCOL= System Option” in SAS System Options: Reference.

- **EMAILFROM**
  
  Specifies whether the FROM e-mail option is required when sending e-mail by using either the FILE or FILENAME statements. For more information, see the “EMAILFROM System Option” in SAS System Options: Reference.

- **EMAILHOST**
  
  Specifies the SMTP server that supports e-mail access for your site. For more information, see the “EMAILHOST= System Option” in SAS System Options: Reference.
EMAILPORT
specifies the port to which the SMTP server is attached. For more information, see the “EMAILPORT System Option” in SAS System Options: Reference.

EMAILUTCOFFSET
specifies a UTC offset that is used in the Date: header field of the e-mail message. For more information, see the “EMAILUTCOFFSET= System Option” in SAS System Options: Reference.

The following system options are specified with other e-mail systems, as well as SMTP:

EMAILID=
specifies the identity of the individual sending e-mail from within SAS. For more information, see the “EMAILID= System Option” in SAS System Options: Reference.

EMAILPW=
specifies your e-mail login password. For more information, see the “EMAILPW= System Option” in SAS System Options: Reference.

Statements That Control SMTP E-mail

FILENAME Statement
In the FILENAME statement, the EMAIL (SMTP) access method enables you to send e-mail programmatically from SAS using the SMTP e-mail interface. For more information, see the “FILENAME Statement” in SAS Global Statements: Reference.

FILE and PUT Statements
You can specify e-mail options in the FILE statement. E-mail options that you specify in the FILE statement override any corresponding e-mail options that you specified in the FILENAME statement.

In the DATA step, after using the FILE statement to define your e-mail fileref as the output destination, use PUT statements to define the body of the message. The PUT statement directives override any other e-mail options in the FILE and FILENAME statements.
Universally Unique Identifiers and the Object Spawner

What Is a Universally Unique Identifier?

A universally unique identifier (UUID) is a 128-bit identifier that consists of date and time information, and the IEEE node address of a host. UUIDs are useful when objects such as rows or other components of a SAS application must be uniquely identified. For example, if SAS is running as a server and is distributing objects to several clients concurrently, you can associate a UUID with each object. This ensures that a particular client and SAS are referencing the same object.

What Is the Object Spawner?

The object spawner is a program that runs on the server and listens for requests. When a request is received, the object spawner accepts the connection and performs the action that is associated with the port or service on which the connection was made. The object spawner can be configured to be a UUID Generator Daemon (UUIDGEND), which creates UUIDs for the requesting SAS session.

The UUIDGEND utility is required for non-Windows hosts that are running versions of SAS prior to SAS 9.4M2.

The UUID Generator Daemon is not required for the following:
- SAS applications that execute on Windows
- SAS applications that execute in UNIX environments that are running SAS version 9.4M2 (or later)

Defining the UUID Generator Daemon

The definition of UUIDGEND is contained in a setup configuration file that you specify when you invoke the object spawner. This configuration file identifies the port that listens for UUID requests, and (in operating environments other than Windows) the configuration file also identifies the UUID node.

If you install UUIDGEND in an operating environment other than Windows, contact SAS Technical Support http://support.sas.com/techsup/contact/ to obtain a UUID node. The UUID node must be unique for each UUIDGEND installation in order for UUIDGEND to guarantee truly unique UUIDs.

Here is an example of a UUIDGEND setup configuration file for an operating environment other than Windows:
Universally Unique Identifiers and the Object Spawner

# Define our UUID Generator Daemon. Since this UUIDGEND is executing on a UNIX host, we contacted SAS Technical Support to get the specified sasUUIDNode.

```
# dn: sasSpawnercn=UUIDGEND,sascomponent=sasServer,cn=SAS,o=ABC Inc,c=US
objectClass: sasSpawner
sasSpawnercn: UUIDGEND
sasDomainName: unx.abc.com
sasMachineDNSName: medium.unx.abc.com
sasOperatorPassword: myPassword
sasOperatorPort: 6340
sasUUIDNode: 0123456789ab
sasUUIDPort: 6341
description: SAS Session UUID Generator Daemon on UNIX
```

Here is an example of a UUIDGEND setup configuration file for Windows:

```
# Define our UUID Generator Daemon. Since this UUIDGEND is executing in a Windows operating environment, we do not need to specify the sasUUIDNode.

# dn: sasSpawnercn=UUIDGEND,sascomponent=sasServer,cn=SAS,o=ABC Inc,c=US
objectClass: sasSpawner
sasSpawnercn: UUIDGEND
sasDomainName: wnt.abc.com
sasMachineDNSName: little.wnt.abc.com
sasOperatorPassword: myPassword
sasOperatorPort: 6340
sasUUIDPort: 6341
description: SAS Session UUID Generator Daemon on XP
```

## Installing the UUID Generator Daemon

When you have created the setup configuration file, you can install UUIDGEND by starting the object spawner program (objspawn) and specifying the setup configuration file with the following syntax:

```
objspawn -configFile filename
```

The `configFile` option can be abbreviated as `-cf`.

`filename` specifies a fully qualified path to the UUIDGEND setup configuration file. Enclose pathnames that contain embedded blanks in single or double quotation marks. On Windows, enclose pathnames that contain embedded blanks in double quotation marks. On z/OS, specify the configuration file as follows:

- `//dsn:myid.objspawn.log` for MVS files
- `//hfs:filename.ext` for OpenEdition files

On Windows, the `objspawn.exe` file is installed in the `SAS-installation-directory\SASFoundation\SAS-version\` directory. For example, in a typical Windows installation, the `objspawn.exe` file might be installed in the following directory:
On UNIX, the objspawn file is installed in the utilities/bin directory in your installed SAS directory.

In the VMS operating environment, the OBJSPAWN_STARTUP.COM file executes the OBJSPAWN.COM file as a detached process. The OBJSPAWN.COM file runs the object spawner. The OBJSPAWN.COM file also includes the following commands that your site might need to perform before the object spawner is started:

- command to set the display node
- command to run the appropriate version of the spawner
- command to define a process level logical name that points to a template DCL file (OBJSPAWN_TEMPLATE.COM)

The OBJSPAWN_TEMPLATE.COM file performs setup that is needed in order for the client process to execute. The object spawner first checks to see whether the logical name SAS$OBJSPAWN_TEMPLATE is defined. If it is, the commands in the template file are executed as part of the command sequence used when starting the client session. You do not have to define the logical name.

Using SAS Language Elements to Assign UUIDs

Overview of Using SAS Language Elements to Assign UUIDs

If your SAS application executes on a platform other than Windows and you have installed UUIDGEND, you can use the following to assign UUIDs:

- UUIDGEN function
- UUIDCOUNT= system option
- UUIDGENDHOST systems option

UUIDGEN Function

The UUIDGEN function returns a UUID for each cell. For more information, see “UUIDGEN Function” in SAS Functions and CALL Routines: Reference.
UUIDCOUNT= System Option

The UUIDCOUNT= system option specifies the number of UUIDs to acquire each time the UUID Generator Daemon is used. For more information, see “UUIDCOUNT= System Option” in SAS System Options: Reference.

UUIDGENDHOST System Option

The UUIDGENDHOST system option identifies the operating environment and the port of the UUID Generator Daemon. For more information, see “UUIDGENDHOST= System Option” in SAS System Options: Reference.

Overview of IPv6

SAS 9.2 introduced support for the next generation of Internet Protocol, IPv6, which is the successor to the current Internet Protocol, IPv4. Rather than replacing IPv4 with IPv6, SAS supports both protocols. There is a lengthy transition period during which the two protocols coexist.

A primary reason for the new protocol is that the limited supply of 32-bit IPv4 address spaces was being depleted. IPv6 uses a 128-bit address scheme. This scheme provides more IP addresses than did IPv4.

IPv6 includes these benefits over IPv4:
- larger address space (128 bits rather than 32 bits)
- simplified header format
- automatic configuration
- more efficient routing
- improved quality of service and security
- compliance with regulatory requirements
- widespread use in global markets

IPv6 Address Format

IPv6 and IPv4 use different address formats. The following table compares the features of the protocols.
### Table 38.1  Comparison of Features of the IPv6 and IPv4 Address Formats

<table>
<thead>
<tr>
<th>Feature</th>
<th>IPv6</th>
<th>IPv4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Space</td>
<td>128-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td>Representation</td>
<td>string</td>
<td>integer</td>
</tr>
<tr>
<td>Length (including Field Separators)</td>
<td>39</td>
<td>15</td>
</tr>
<tr>
<td>Field Separator</td>
<td>colon (:)</td>
<td>period (.)</td>
</tr>
<tr>
<td>Notation</td>
<td>hexadecimal</td>
<td>decimal</td>
</tr>
<tr>
<td>Example of IP Address</td>
<td>db8:0:0:1</td>
<td>10.23.2.3</td>
</tr>
</tbody>
</table>

### Examples of IPv6 Addresses

#### Example of Full and Collapsed IPv6 Address

Here is an example of a full IPv6 address:

FE80:0000:0000:0000:0202:B3FF:FE1E:8329

It shows a 128-bit address in eight 16-bit blocks in the format `global:subnet:interface`.

Here is an example of a collapsed IPv6 address:

FE80::0202:B3FF:FE1E:8329

The `::` (consecutive colons) notation can be used to represent four successive 16-bit blocks that contain zeros. When SAS software encounters a collapsed IP address, it reconstitutes the address to the required 128-bit address in eight 16-bit blocks.

#### Example of an IPv6 Address That Includes a Port Number

Here is an example of an IP address that contains a port number:

[2001:db8:0::1]:80

The brackets are necessary only if also specifying a port number. Brackets are used to separate the address from the port number. If no port number is used, the brackets can be omitted.
As an alternative, the block that contains the zero can be collapsed. Here is an example:

\[2001:db8::1]:80

---

**Example of an IPv6 Address That Includes a URL**

Here is an example of an IP address that contains a URL:

http://[2001:db8:0::1]:80

The `http://` prefix specifies a URL. The brackets are necessary only if also specifying a port number. Brackets are used to separate the address from the port number. If no port number is used, the brackets can be omitted.

---

**Fully Qualified Domain Names (FQDN)**

Because IP addresses can change easily, SAS applications that contain hardcoded IP addresses are prone to maintenance problems.

To avoid such problems, use of an FQDN is preferred over an IP address. The name-resolution system that is part of the TCP/IP protocol is responsible for locating the IP address that is associated with the FQDN.

The following example restores client activity in the paused repository:

```sas
PROC METAOPERATE
    SERVER="d6292.us.company.com"
    PORT=2222
    USERID="myuserid"
    PASSWORD="mypassword"
    PROTOCOL=BRIDGE
    ACTION=RESUME
    OPTIONS=""
    NOAUTOPAUSE;
```

If an IP address had been used and if the IP address that was associated with the computer node name had changed, the code would be inaccurate.

An FQDN can remain intact in the code while the underlying IP address can change without causing unpredictable results. The TCP/IP name-resolution system automatically resolves the FQDN to its associated IP address.

Here is an example of an FQDN that is specified in a SAS GUI application.
The full FQDN, d11076.na.apex.com, is specified in the Remote Host field of the Connect Server Properties window in SAS Management Console.

Some SAS products impose limits on the length for computer names.

The following code is an example of an FQDN that is assigned to a SAS menu variable:

```sas
%let sashost=hrmach1.dorg.com;
rsubmit sashost.sasport;
```

Because the FQDN is longer than eight characters, the FQDN must be assigned to a SAS macro variable, which is used in the RSUBMIT statement.
Appendixes

Appendix 1

Data Sets Used in Examples .............................................. 751

Appendix 2

Understanding How the DATA Step Works .......................... 765
## Appendix 1

### Data Sets Used in Examples

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Animal</td>
<td>752</td>
</tr>
<tr>
<td>AnimalDupes</td>
<td>752</td>
</tr>
<tr>
<td>AnimalMissing</td>
<td>752</td>
</tr>
<tr>
<td>CarSales</td>
<td>753</td>
</tr>
<tr>
<td>CarsSmall</td>
<td>753</td>
</tr>
<tr>
<td>Class</td>
<td>754</td>
</tr>
<tr>
<td>Classfit</td>
<td>754</td>
</tr>
<tr>
<td>Inventory</td>
<td>755</td>
</tr>
<tr>
<td>InventoryAdd</td>
<td>755</td>
</tr>
<tr>
<td>One</td>
<td>755</td>
</tr>
<tr>
<td>Many</td>
<td>756</td>
</tr>
<tr>
<td>Master</td>
<td>756</td>
</tr>
<tr>
<td>Minerals</td>
<td>756</td>
</tr>
<tr>
<td>Plant</td>
<td>757</td>
</tr>
<tr>
<td>PlantDupes</td>
<td>757</td>
</tr>
<tr>
<td>PlantG</td>
<td>757</td>
</tr>
<tr>
<td>PlantMissing</td>
<td>757</td>
</tr>
<tr>
<td>PlantMissing2</td>
<td>758</td>
</tr>
<tr>
<td>PlantNew</td>
<td>758</td>
</tr>
<tr>
<td>PlantNewDupes</td>
<td>758</td>
</tr>
<tr>
<td>Product_List</td>
<td>759</td>
</tr>
<tr>
<td>Quarter1, Quarter 2, Quarter3, Quarter4</td>
<td>759</td>
</tr>
<tr>
<td>Sales</td>
<td>760</td>
</tr>
<tr>
<td>Sales2019</td>
<td>760</td>
</tr>
<tr>
<td>Supplier</td>
<td>760</td>
</tr>
<tr>
<td>Table1</td>
<td>761</td>
</tr>
<tr>
<td>Table2</td>
<td>761</td>
</tr>
<tr>
<td>Year1</td>
<td>761</td>
</tr>
<tr>
<td>Year2</td>
<td>761</td>
</tr>
</tbody>
</table>

### Description of Column-Binary Data Storage

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>762</td>
</tr>
</tbody>
</table>
Data Sets Used in Examples

Animal

data animal;
input common $ animal $;
datalines;
a Ant
b Bird
c Cat
d Dog
e Eagle
f Frog
;

AnimalDupes

data animalDupes;
input common $ animal $;
datalines;
a Ant
a Ape
b Bird
c Cat
d Dog
e Eagle
;

AnimalMissing

data animalMissing;
input common $ animal $;
datalines;
a Ant
c Cat
d Dog
e Eagle
;
CarSales

data carSales;
input transID $1-9 make $11-19 model $21-38;
datalines;
CSH203073 Chevrolet Aveo 4dr
CCA460564 Chevrolet Aveo LS 4dr htc
DEB848135 Honda Civic DX 2dr
CCA762350 Honda Insight 2dr
CCA314633 Hyundai Accent 2dr hatch
CSH118553 Hyundai Accent GL 4dr
CCA295057 Hyundai Accent GT 2dr htc
CSH285627 Kia Rio 4dr auto
DEB898883 Kia Rio 4dr manual
CCA803483 Kia Rio Cinco
DEB661568 Mazda MX-5 Miata LS
CSH36345 Mazda MX-5 Miata
CCA562255 Scion xA 4dr hatch
CCA651575 Scion xB
DEB322009 Toyota Echo 2dr auto
CSH607254 Toyota Echo 2dr manual
CSH879119 Toyota Echo 4dr
CSH230940 Toyota MR2 Spyder
;

CarsSmall

data carsSmall;
input make $1-9 model $11-39 type $41-46 driveTrain $48-52 weight $54-57 length 59-61 makeModelDrive $63-120;
datalines;
Chevrolet Aveo 4dr Sedan Front 2370 167
Chevrolet Aveo 4dr - Front wheel drive
Chevrolet Aveo LS 4dr hatch Sedan Front 2348 153
Chevrolet Aveo LS 4dr hatch - Front wheel drive
Honda Civic DX 2dr Sedan Front 2432 175 Honda
Civic DX 2dr - Front wheel drive
Honda Insight 2dr (gas/electric) Hybrid Front 1850 155 Honda
Insight 2dr (gas/electric) - Front wheel drive
Hyundai Accent 2dr hatch Sedan Front 2255 167 Hyundai
Accent 2dr hatch - Front wheel drive
Hyundai Accent GL 4dr Sedan Front 2290 167 Hyundai
Accent GL 4dr - Front wheel drive
Hyundai Accent GT 2dr hatch Sedan Front 2339 167 Hyundai
Accent GT 2dr hatch - Front wheel drive
Kia Rio 4dr auto Sedan Front 2458 167 Kia Rio
4dr auto - Front wheel drive
Kia Rio 4dr manual                    Sedan  Front 2403 167 Kia Rio
4dr manual - Front wheel drive
Kia Rio  Cinco                      Wagon  Front 2447 167 Kia Rio
Cinco - Front wheel drive
Mazda     MX-5 Miata LS convertible 2dr Sports Rear  2387 156 Mazda
MX-5 Miata LS convertible 2dr - Rear wheel drive
Mazda     MX-5 Miata convertible 2dr Sports Rear  2387 156 Mazda
MX-5 Miata convertible 2dr - Rear wheel drive
Scion     xA 4dr hatch             Sedan  Front 2340 154 Scion xA
4dr hatch - Front wheel drive
Scion     xB                        Wagon  Front 2425 155 Scion xB
- Front wheel drive
Toyota    Echo 2dr auto            Sedan  Front 2085 163 Toyota
Echo 2dr auto - Front wheel drive
Toyota    Echo 2dr manual          Sedan  Front 2035 163 Toyota
Echo 2dr manual - Front wheel drive
Toyota    Echo 4dr                 Sedan  Front 2055 163 Toyota
Echo 4dr - Front wheel drive
Toyota    MR2 Spyder convertible 2dr Sports Rear 2195 153 Toyota
MR2 Spyder convertible 2dr - Rear wheel drive
;

Class

data class;
input name $ age height weight;
format weight comma8.;
format height comma8.;
datalines;
Alice   13 56.5   84.00
Barbara 13 65.3   98.00
Carol   14 62.8  102.50
Jane    12 59.8   84.50
Janet   15 62.5  112.50
Joyce   11 51.3   50.50
Judy    14 64.3   90.00
Louise  12 56.3   77.00
Mary    15 66.5  112.00
;

Classfit

data classfit;
input name $ age height weight predict;
format weight comma8.2;
format predict comma8.1;
label weight="Weight in Pounds";
datalines;
Janet   15 62.5  112.5 100.662
Mary    15 66.5  112.0 116.259
Philip  16 72.0  112.0 137.703
Ronald  15 67.0 133.0 118.208
William 15 66.5 150.0 116.259
;

Inventory

data Inventory;
input partNumber $ partName $;
datalines;
K89R seal
M4J7 sander
LK43 filter
MN21 brace
BC85 clamp
NCF3 valve
KJ66 cutter
UYN7 rod
JD03 switch
BV1E timer
;

InventoryAdd

data InventoryAdd;
input partNumber $ partName $ newStock newPrice;
format newPrice dollar12.2;
datalines;
K89R seal 6 247.50
AA11 hammer 55 32.26
BB22 wrench 21 17.35
KJ66 cutter 10 24.50
CC33 socket 7 22.19
BV1E timer 30 36.50
;

One

data one;
   input ID state $;
datalines;
1 AZ
2 MA
3 WA
4 WI
;
Many

    data many;
        input ID city $ state $;
    datalines;
    1 Phoenix Ariz
    2 Boston Mass
    2 Foxboro Mass
    3 Olympia Mass
    3 Seattle Wash
    3 Spokane Wash
    4 Madison Wis
    4 Milwaukee Wis
    4 Madison Wis
    4 Hurley Wis
    ;

Master

    data master;
        input common $ animal $ plant $;
    datalines;
    a Ant Apple
    b Bird Banana
    c Cat Coconut
    d Dog Dewberry
    e Eagle Eggplant
    f Frog Fig
    ;

Minerals

    data minerals;
        input common $ plant $ mineral $;
    datalines;
    a Apricot Amethyst
    b Barley Beryl
    c Cactus
    e .
    f Fennel
    g Grape Garnet
    ;
Plant

data plant;
input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
d Dewberry
e Eggplant
f Fig
;

PlantDupes

data plantDupes;
input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
c Celery
d Dewberry
e Eggplant
;

PlantG

data plantG;
input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
d Dewberry
e Eggplant
g Fig
;

PlantMissing

data plantMissing;
input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
;

PlantMissing2

data plantMissing2;
  input common $ plant $;
datalines;
a Apple
b Banana
c Coconut
e Eggplant
f Fig
;

PlantNew

data plantNew;
  input common $ plant $;
datalines;
a Apricot
b Barley
c Cactus
d Date
e Escarole
f Fennel
;

PlantNewDupes

data plantNewDupes;
  input common $ plant $;
datalines;
a Apricot
b Barley
c Cactus
d Date
d Dill
e Escarole
f Fennel
;
Product List

data product_list;
input Product_Id Product_Name $14-49 Supplier_ID;
datalines;
240200100101 Grandslam Staff Tour Mhl Golf Gloves 3808
210200100017 Sweatshirt Children's O-Neck 3298
240400200022 Aftm 95 Vf Long Bg-65 White 1280
230100100017 Men's Jacket Rem 50
210200300006 Fleece Cuff Pant Kid'S 1303
210200500002 Children's Mitten 772
210200700016 Strap Pants BBO 798
210201000050 Kid Children's T-Shirt 2963
21020010009 Kids Sweat Round Neck, Large Logo 3298
210201000067 Logo Coord. Children's Sweatshirt 2963
220100100019 Fit Racing Cap 1303
220100100025 Knit Hat 1303
220100300001 Fleece Jacket Compass 772
220200200036 Soft Astro Men's Running Shoes 1747
230100100015 Men's Jacket Caians 50
230100500004 Backpack Flag, 6,5x9 Cm. 316
210200500006 Rain Suit, Plain w/backpack Jacket 772
230100500006 Collapsible Water Can 316
224040020000 Bat 5-Ply 3808
220200200035 Soft Alta Plus Women's Indoor Shoes 1747
240400200066 Memhis 350, Yellow Medium, 6-pack 1280
240200100081 Extreme Distance 90 3-pack 3808
;

Quarter1, Quarter 2, Quarter3, Quarter4

data quarter1;
length mileage 4;
input account mileage;
datalines;
1 932
2 563
;
data quarter2;
length mileage 8;
input account mileage;
datalines;
1 1288
2 1087
;
data quarter3;
length mileage 6;
input account mileage;
datalines;
1 2781
2 1990
;
data quarter4;
length mileage 6;
input account mileage;
datalines;
1 3278
2 2209
;

Sales

data Sales;
input partNumber $ partName $ salesPerson $;
datalines;
NCF3 valve  JN
BV1E timer  JN
LK43 filter  KM
K89R seal      SJ
LK43 filter  JN
M4J7 sander  KM
BV1E timer  KM
;

Sales2019

data Sales2019;
input partNumber $ lastSoldDate mmddyy10.;
format lastSoldDate mmddyy10.;
datalines;
BC85 10/15/2019
BV1E 10/23/2019
KJ66 11/11/2019
LK43 09/12/2019
MN21 09/13/2019
NCF3 07/24/2019
UYN7 12/11/2019
;

Supplier

data Supplier;
input Supplier_ID Supplier_Name $6-32 Supplier_Address $34-52 Country $54-55;
datalines;
50 Scandinavian Clothing A/S       Kr. Augusts Gate 13 NO
316 Prime Sports Ltd               9 Carlisle Place GB

Table 1

data Table1;
   set sashelp.class(where=(age=14));
run;

Table 2

data Table2;
   set sashelp.classfit(where=(age=14));
   drop uppermean lower upper;
run;
proc sort data=Table2; by name; run;

Year 1

data Year1;
input date;
datalines;
2009
2010
2011
2012
;

Year 2

data Year2;
input date;
datalines;
2010
2011
2012
2013
2014
Description of Column-Binary Data Storage

The arrangement and numbering of rows in a column on physical punched cards originated with the Hollerith system of encoding characters and numbers. It was based on using a pair of values to represent either a character or a numeric digit. In the Hollerith system, each column on a card had a maximum of two punches, one punch in the zone portion, and one in the digit portion. These punches corresponded to a pair of values, and each pair of values corresponded to a specific alphabetic character or sign and numeric digit.

In the zone portion of the punched card (the first three rows), the zone component of the pair can have the values 12, 11, 0 (or 10), or not punched. In the digit portion of the card (the fourth through the twelfth rows), the digit component of the pair can have the values 1 through 9, or not punched.

The following figure shows the multi-punch combinations corresponding to letters of the alphabet.

Figure A1.1  Columns and Rows in a Punched Card

SAS stores each column of column-binary data (a “virtual” punched card) in two bytes. Since each column has only 12 positions and since 2 bytes contain 16 positions, the 4 extra positions within the bytes are located at the beginning of each byte. The following figure shows the correspondence between the rows of “virtual” punched card data and the positions within 2 bytes that SAS uses to store them. SAS stores a punched position as a binary 1 bit and an unpunched position as a binary 0 bit.
Figure A1.2  Column-Binary Representation on a “Virtual” Punched Card
Appendix 2

Understanding How the DATA Step Works

How the DATA Step Processes Data

Flow of Action .................................................. 765
The Compilation Phase ...................................... 766
The Execution Phase ........................................... 767

Processing a DATA Step: A Walk-Through

Sample DATA Step .............................................. 768
Creating the Input Buffer and the Program Buffer ......... 768
Reading a Record .............................................. 769
Writing an Observation to the SAS Data Set ................. 770
Reading the Next Record ..................................... 771
When the DATA Step Finishes Executing ....................... 772

More About DATA Step Execution

The Default Sequence of Execution ......................... 772
Changing the Sequence of Execution Using Statements .... 774
Changing the Sequence of Execution Using Functions ...... 774
Altering the Flow for a Given Observation ................... 775
Step Boundaries .............................................. 776
What Causes the DATA Step to Stop Executing ............... 777

How the DATA Step Processes Data

Flow of Action

When you submit a DATA step for execution, it is first compiled and then executed. The following figure shows the flow of action for a typical SAS DATA step.
The Compilation Phase

When you submit a DATA step for execution, SAS checks the syntax of the SAS statements and compiles them; that is, it automatically translates the statements into machine code. In this phase, SAS identifies the type and length of each new variable and determines whether a variable type conversion is necessary for each subsequent reference to a variable. During the compilation phase, SAS creates the following three items:
input buffer
is a logical area in memory into which SAS reads each record of raw data when SAS executes an INPUT statement. This buffer is created only when the DATA step reads raw data. When the DATA step reads a SAS data set, SAS reads the data directly into the program data vector.

program data vector (PDV)
is a logical area in memory where SAS builds a data set, one observation at a time. When a program executes, SAS reads data values from the input buffer or creates them by executing SAS language statements. The data values are assigned to the appropriate variables in the program data vector. From here, SAS writes the values to a SAS data set as a single observation.

Along with data set variables and computed variables, the PDV contains two automatic variables: _N_ and _ERROR_. The _N_ variable counts the number of times the DATA step begins to iterate. The _ERROR_ variable signals the occurrence of an error caused by the data during execution. The value of _ERROR_ is either 0 (indicating no errors exist), or 1 (indicating that one or more errors have occurred). SAS does not write these variables to the output data set.

descriptor information
is information that SAS creates and maintains about each SAS data set, including data set attributes and variable attributes. For example, it contains the name of the data set, its member type, the date and time that the data set was created, and the number, names, and data types (character or numeric) of the variables. The descriptor information also contains information about extended attributes (if defined on a data set). Extended attribute descriptor information includes the name of the attribute, the name of the variable, and the value of the attribute.

The Execution Phase

By default, a simple DATA step iterates once for each observation that is being created. The flow of action in the Execution Phase of a simple DATA step is described as follows:

1. The DATA step begins with a DATA statement. Each time the DATA statement executes, a new iteration of the DATA step begins, and the _N_ automatic variable is incremented by 1.

2. SAS sets the newly created program variables to missing in the program data vector (PDV).

3. SAS reads a data record from a raw data file into the input buffer, or it reads an observation from a SAS data set directly into the program data vector. You can use an INPUT, MERGE, SET, MODIFY, or UPDATE statement to read a record.

4. SAS executes any subsequent programming statements for the current record.

5. At the end of the statements, an output, return, and reset occur automatically. SAS writes an observation to the SAS data set, the system automatically returns to the top of the DATA step, and the values of variables created by INPUT and assignment statements are reset to missing in the program data vector. Note that variables that you read with a SET, MERGE, MODIFY, or UPDATE statement are not reset to missing here.
The DATA step terminates when SAS encounters the end-of-file in a SAS data set or a raw data file. 

Note: The figure shows the default processing of the DATA step. You can place data-reading statements (such as INPUT or SET), or data-writing statements (such as OUTPUT), in any order in your program.

Processing a DATA Step: A Walk-Through

Sample DATA Step

The following statements provide an example of a DATA step that reads raw data, calculates totals, and creates a data set:

```sas
data total_points (drop=TeamName);  1
  input TeamName $ ParticipantName $ Event1 Event2 Event3;  2
  TeamTotal + (Event1 + Event2 + Event3);  3
datalines;
Knights Sue    6  8  8
Kings Jane     9  7  8
Knights John   7  7  7
Knights Lisa   8  9  9
Knights Fran   7  6  6
Knights Walter 9  8 10
;
proc print data=total_points;
run;
```

1 The DROP= data set option prevents the variable TeamName from being written to the output SAS data set called Total_Points.

2 The INPUT statement describes the data by giving a name to each variable, identifying its data type (character or numeric), and identifying its relative location in the data record.

3 The SUM statement accumulates the scores for three events in the variable TeamTotal.

Creating the Input Buffer and the Program Buffer

When DATA step statements are compiled, SAS determines whether to create an input buffer. If the input file contains raw data (as in the example above), SAS creates an input buffer to hold the data before moving the data to the program data
vector (PDV). (If the input file is a SAS data set, however, SAS does not create an input buffer. SAS writes the input data directly to the PDV.)

The PDV contains all the variables in the input data set, the variables created in DATA step statements, and the two variables, _N_ and _ERROR_, that are automatically generated for every DATA step. The _N_ variable represents the number of times the DATA step has iterated. The _ERROR_ variable acts like a binary switch whose value is 0 if no errors exist in the DATA step, or 1 if one or more errors exist. The following figure shows the Input Buffer and the program data vector after DATA step compilation.

**Figure A2.2  Input Buffer and Program Data Vector**

**Input Buffer**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 |

**Program Data Vector**

<table>
<thead>
<tr>
<th>TeamName</th>
<th>ParticipantName</th>
<th>Event1</th>
<th>Event2</th>
<th>Event3</th>
<th>TeamTotal</th>
<th><em>N</em></th>
<th><em>ERROR</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Variables that are created by the INPUT and the Sum statements (TeamName, ParticipantName, Event1, Event2, Event3, and TeamTotal) are set to missing initially. Note that in this representation, numeric variables are initialized with a period and character variables are initialized with blanks. The automatic variable _N_ is set to 1; the automatic variable _ERROR_ is set to 0.

The variable TeamName is marked Drop in the PDV because of the DROP= data set option in the DATA statement. Dropped variables are not written to the SAS data set. The _N_ and _ERROR_ variables are dropped because automatic variables created by the DATA step are not written to a SAS data set. See Chapter 5, “Variables,” on page 69 for details about automatic variables.

---

**Reading a Record**

SAS reads the first data line into the input buffer. The input pointer, which SAS uses to keep its place as it reads data from the input buffer, is positioned at the beginning of the buffer, ready to read the data record. The following figure shows the position of the input pointer in the input buffer before SAS reads the data.

**Figure A2.3  Position of the Pointer in the Input Buffer Before SAS Reads Data**

**Input Buffer**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 |

↑

The INPUT statement then reads data values from the record in the input buffer and writes them to the PDV where they become variable values. The following figure
shows both the position of the pointer in the input buffer, and the values in the PDV after SAS reads the first record.

**Figure A2.4** Values from the First Record Are Read into the Program Data Vector

Input Buffer

```
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
Knights Sue 6 8 8
```

Program Data Vector

```
TeamName | ParticipantName | Event1 | Event2 | Event3 | TeamTotal | _N_ | _ERROR_
---------|-----------------|-------|-------|-------|-----------|-----|--------
Knights  | Sue             | 6     | 8     | 8     | 22        | 1   | 0      
```

After the INPUT statement reads a value for each variable, SAS executes the Sum statement. SAS computes a value for the variable TeamTotal and writes it to the PDV. The following figure shows the PDV with all of its values before SAS writes the observation to the data set.

**Figure A2.5** Program Data Vector with Computed Value of the Sum Statement

Program Data Vector

```
TeamName | ParticipantName | Event1 | Event2 | Event3 | TeamTotal | _N_ | _ERROR_
---------|-----------------|-------|-------|-------|-----------|-----|--------
Knights  | Sue             | 6     | 8     | 8     | 22        | 1   | 0      
```

Writing an Observation to the SAS Data Set

When SAS executes the last statement in the DATA step, all values in the PDV, except those marked to be dropped, are written as a single observation to the data set Total_Points. The following figure shows the first observation in the Total_Points data set.

**Figure A2.6** The First Observation in Data Set Total_Points

```
ParticipantName | Event1 | Event2 | Event3 | TeamTotal
Sue             | 6     | 8     | 8      | 22
```

SAS then returns to the DATA statement to begin the next iteration. SAS resets the values in the PDV in the following way:

- The values of variables created by the INPUT statement are set to missing.
- The value created by the Sum statement is automatically retained.
The value of the automatic variable _N_ is incremented by 1, and the value of _ERROR_ is reset to 0.

The following figure shows the current values in the PDV.

**Figure A2.7  Current Values in the Program Data Vector**

Program Data Vector

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>TeamTotal</th>
<th><em>N</em></th>
<th><em>ERROR</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop</td>
<td>Drop</td>
<td>Drop</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Reading the Next Record**

SAS reads the next record into the input buffer. The INPUT statement reads the data values from the input buffer and writes them to the PDV. The Sum statement adds the values of Event1, Event2, and Event3 to TeamTotal. The value of 2 for variable _N_ indicates that SAS is beginning the second iteration of the DATA step. The following figure shows the input buffer, the PDV for the second record, and the SAS data set with the first two observations.

**Figure A2.8  Input Buffer, Program Data Vector, and First Two Observations**

Program Data Vector

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>TeamTotal</th>
<th><em>N</em></th>
<th><em>ERROR</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinals</td>
<td>Jane</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>46</td>
<td>2</td>
</tr>
<tr>
<td>Drop</td>
<td>Drop</td>
<td>Drop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output SAS Data Set TOTAL_POINTS: 1st and 2nd observations

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Jane</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>46</td>
<td></td>
</tr>
</tbody>
</table>

As SAS continues to read records, the value in TeamTotal grows larger as more participant scores are added to the variable. _N_ is incremented at the beginning of each iteration of the DATA step. This process continues until SAS reaches the end of the input file.
When the DATA Step Finishes Executing

The DATA step stops executing after it processes the last input record. You can use PROC PRINT to print the output in the Total_Points data set:

```sas
data total_points (drop=TeamName);
  input TeamName $ ParticipantName $ Event1 Event2 Event3;
  TeamTotal + (Event1 + Event2 + Event3);
  datalines;
  Knights Sue 6 8 8
  Cardinals Jane 9 7 8
  Knights John 7 7 7
  Cardinals Lisa 8 9 9
  Cardinals Fran 7 6 6
  Knights Walter 9 8 10
;
proc print data=total_points;
  title 'Total Team Scores';
run;
```

**Output A2.1**  Output from the Walk-through DATA Step

<table>
<thead>
<tr>
<th>Total Team Scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

More About DATA Step Execution

The Default Sequence of Execution

The following table outlines the default sequence of execution for statements in a DATA step. The DATA statement begins the step and identifies usually one or more SAS data sets that the step will create. (You can use the keyword _NULL_ as the
Optional programming statements process your data. SAS then performs the default actions at the end of processing an observation.

### Table A2.1  Default Execution for Statements in a DATA Step

<table>
<thead>
<tr>
<th>Structure of a DATA Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA statement</td>
<td>begins the step counts iterations</td>
</tr>
<tr>
<td>Data-reading statements: 1</td>
<td>describes the arrangement of values in the input data record from a raw data source</td>
</tr>
<tr>
<td>INPUT</td>
<td>reads an observation from one or more SAS data sets</td>
</tr>
<tr>
<td>SET</td>
<td>joins observations from two or more SAS data sets into a single observation</td>
</tr>
<tr>
<td>MERGE</td>
<td>replaces, deletes, or appends observations in an existing SAS data set in place</td>
</tr>
<tr>
<td>MODIFY</td>
<td>updates a master file by applying transactions</td>
</tr>
<tr>
<td>Optional SAS programming statements, for example:</td>
<td>further processes the data for the current observation</td>
</tr>
<tr>
<td>FirstQuarter=Jan+Feb+Mar; if RetailPrice &lt; 500;</td>
<td>computes the value for FirstQuarter for the current observation subsets by value of variable RetailPrice for the current observation</td>
</tr>
<tr>
<td>Default actions at the end of processing an observation</td>
<td></td>
</tr>
<tr>
<td>At end of DATA step:</td>
<td>writes an observation to a SAS data set returns to the DATA statement resets values to missing in program data vector</td>
</tr>
<tr>
<td>Automatic write, automatic return</td>
<td></td>
</tr>
<tr>
<td>At top of DATA step:</td>
<td></td>
</tr>
<tr>
<td>Automatic reset</td>
<td></td>
</tr>
</tbody>
</table>

1 The table shows the default processing of the DATA step. You can alter the sequence of statements in the DATA step. You can code optional programming statements, such as creating or reinitializing a constant, before you code a data-reading statement.
Changing the Sequence of Execution Using Statements

You can change the default sequence of execution to control how your program executes. SAS language statements offer you a lot of flexibility to do this in a DATA step. The following list shows the most common ways to control the flow of execution in a DATA step program.

Table A2.2 Common Methods That Alter the Sequence of Execution

<table>
<thead>
<tr>
<th>Task</th>
<th>Possible Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read a record</td>
<td>merge, modify, join data sets</td>
</tr>
<tr>
<td></td>
<td>read multiple records to create a single observation</td>
</tr>
<tr>
<td></td>
<td>randomly select records for processing</td>
</tr>
<tr>
<td></td>
<td>read from multiple external files</td>
</tr>
<tr>
<td></td>
<td>read selected fields from a record by using statement or data set options</td>
</tr>
<tr>
<td>Process data</td>
<td>use conditional logic</td>
</tr>
<tr>
<td></td>
<td>retain variable values</td>
</tr>
<tr>
<td>Write an observation</td>
<td>write to a SAS data set or to an external file</td>
</tr>
<tr>
<td></td>
<td>control when output is written to a data set</td>
</tr>
<tr>
<td></td>
<td>write to multiple files</td>
</tr>
</tbody>
</table>

For more information, see the individual statements in *SAS DATA Step Statements: Reference*.

Changing the Sequence of Execution Using Functions

You can also use functions to read and process data. For information about how statements and functions process data differently, see “Using Functions to Manipulate Files” in *SAS Functions and CALL Routines: Reference*.
Altering the Flow for a Given Observation

You can use statements, statement options, and data set options to alter how SAS processes specific observations. The following table lists SAS language elements and their effects on processing.

**Table A2.3  Language Elements That Alter Programming Flow**

<table>
<thead>
<tr>
<th>SAS Language Element</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>subsetting IF statement</td>
<td>stops the current iteration when a condition is false, does not write the current observation to the data set, and returns control to the top of the DATA step.</td>
</tr>
<tr>
<td>IF-THEN/ELSE statement</td>
<td>executes a SAS statement for observations that meet the current condition and continues with the next statement.</td>
</tr>
<tr>
<td>DO loops</td>
<td>cause parts of the DATA step to be executed multiple times.</td>
</tr>
<tr>
<td>LINK and RETURN statements</td>
<td>alter the flow of control, execute statements following the label specified, and return control of the program to the next statement following the LINK statement.</td>
</tr>
<tr>
<td>HEADER= option in the FILE statement</td>
<td>alters the flow of control whenever a PUT statement causes a new page of output to begin; statements following the label specified in the HEADER= option are executed until a RETURN statement is encountered, at which time control returns to the point from which the HEADER= option was activated.</td>
</tr>
<tr>
<td>GO TO statement</td>
<td>alters the flow of execution by branching to the label that is specified in the GO TO statement. SAS executes subsequent statements then returns control to the beginning of the DATA step.</td>
</tr>
<tr>
<td>EOF= option in an INFILE statement</td>
<td>alters the flow of execution when the end of the input file is reached; statements following the label that is specified in the EOF= option are executed at that time.</td>
</tr>
<tr>
<td><em>N</em> automatic variable in an IF-THEN construct</td>
<td>causes parts of the DATA step to execute only for particular iterations.</td>
</tr>
<tr>
<td>SAS Language Element</td>
<td>Function</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------</td>
</tr>
<tr>
<td>SELECT statement</td>
<td>conditionally executes one of a group of SAS statements.</td>
</tr>
<tr>
<td>OUTPUT statement in an IF-THEN construct</td>
<td>outputs an observation before the end of the DATA step, based on a condition; prevents automatic output at the bottom of the DATA step.</td>
</tr>
<tr>
<td>DELETE statement in an IF-THEN construct</td>
<td>deletes an observation based on a condition and causes a return to the top of the DATA step.</td>
</tr>
<tr>
<td>ABORT statement in an IF-THEN construct</td>
<td>stops execution of the DATA step and instruct SAS to resume execution with the next DATA or PROC step. It can also stop executing a SAS program altogether, depending on the options specified in the ABORT statement and on the method of operation.</td>
</tr>
<tr>
<td>WHERE statement or WHERE= data set option</td>
<td>causes SAS to read certain observations based on one or more specified criteria.</td>
</tr>
</tbody>
</table>

**Step Boundaries**

Understanding step boundaries is an important concept in SAS programming because step boundaries determine when SAS statements take effect. SAS executes program statements only when SAS crosses a default or a step boundary. Consider the following DATA steps:

```sas
data _null_; 1
   set allscores(drop=score5-score7);
   title 'Student Test Scores'; 2

data employees; 3
   set employee_list;
run;
```

1. The DATA statement begins a DATA step and is a step boundary.
2. The TITLE statement is in effect for both DATA steps because it appears before the boundary of the first DATA step. (The TITLE statement is a global statement.)
3. The DATA statement is the default boundary for the first DATA step.

The TITLE statement in this example is in effect for the first DATA step as well as for the second because the TITLE statement appears before the boundary of the first DATA step. This example uses the default step boundary `data employees;`.

The following example shows an OPTIONS statement inserted after a RUN statement.
The DATA statement is a step boundary.

The RUN statement is the boundary for the first DATA step.

The OPTIONS statement affects the second DATA step only.

The OPTIONS statement specifies that the first observation that is read from the input data set should be the 5th, and the last observation that is read should be the 55th. Inserting a RUN statement immediately before the OPTIONS statement causes the first DATA step to reach its boundary (run;) before SAS encounters the OPTIONS statement. The OPTIONS statement settings, therefore, are put into effect for the second DATA step only.

Following the statements in a DATA step with a RUN statement is the simplest way to make the step begin to execute, but a RUN statement is not always necessary. SAS recognizes several step boundaries for a SAS step:

- another DATA statement
- a PROC statement
- a RUN statement

Note: For SAS programs executed in interactive mode, a RUN statement is required to signal the step boundary for the last step that you submit.

- the semicolon (with a DATALINES or CARDS statement) or four semicolons (with a DATALINES4 or CARDS4 statement) after data lines
- an ENDSAS statement
- in noninteractive or batch mode, the end of a program file containing SAS programming statements
- a QUIT statement (for some procedures)

When you submit a DATA step during interactive processing, it does not begin running until SAS encounters a step boundary. This fact enables you to submit statements as you write them while preventing a step from executing until you have entered all the statements.

What Causes the DATA Step to Stop Executing

DATA steps stop executing under different circumstances, depending on the type and number of sources of input.
### Table A2.4 Causes That Stop DATA Step Execution

<table>
<thead>
<tr>
<th>Data Read</th>
<th>Data Source</th>
<th>SAS Statements</th>
<th>DATA Step Stops</th>
</tr>
</thead>
<tbody>
<tr>
<td>no data</td>
<td></td>
<td></td>
<td>after only one iteration</td>
</tr>
<tr>
<td>any data</td>
<td></td>
<td></td>
<td>when it executes STOP or ABORT when the data is exhausted</td>
</tr>
<tr>
<td>raw data</td>
<td>instream data lines</td>
<td>INPUT statement</td>
<td>after the last data line is read</td>
</tr>
<tr>
<td></td>
<td>one external file</td>
<td>INPUT and INFILE statements</td>
<td>when end-of-file is reached</td>
</tr>
<tr>
<td></td>
<td>multiple external files</td>
<td>INPUT and INFILE statements</td>
<td>when end-of-file is first reached on any of the files</td>
</tr>
<tr>
<td>observations sequentially</td>
<td>one SAS data set</td>
<td>SET and MODIFY statements</td>
<td>after the last observation is read</td>
</tr>
<tr>
<td></td>
<td>multiple SAS data sets</td>
<td>one SET, MERGE, MODIFY, or UPDATE statement</td>
<td>when all input data sets are exhausted</td>
</tr>
<tr>
<td></td>
<td>multiple SAS data sets</td>
<td>multiple SET, MERGE, MODIFY, or UPDATE statements</td>
<td>when end-of-file is reached by any of the data-reading statements</td>
</tr>
</tbody>
</table>

A DATA step that reads observations from a SAS data set with a SET statement that uses the POINT= option has no way to detect the end of the input SAS data set. (This method is called direct or random access.) Such a DATA step usually requires a STOP statement.

A DATA step also stops when it executes a STOP or an ABORT statement. Some system options and data set options, such as OBS=, can cause a DATA step to stop earlier than it would otherwise.

If the VARINITCHK= system option is set to ERROR, a DATA step stops processing and writes an error to the SAS log if a variable is not initialized. For more information, see “VARINITCHK= System Option” in SAS System Options: Reference.