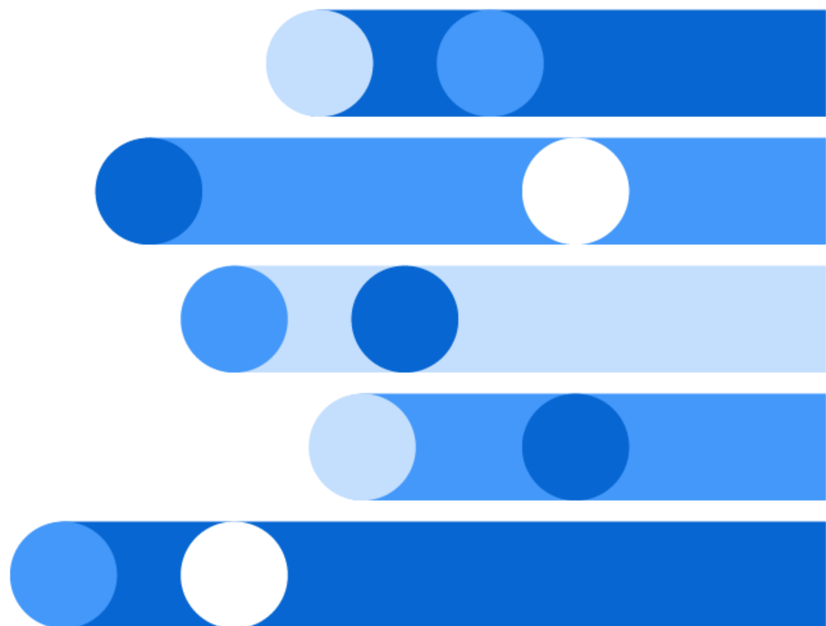
SAS® 9.4 Programmer's Guide: Essentials



SAS® Documentation
August 8, 2024

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® 9.4 Programmer's Guide: Essentials*. Cary, NC: SAS Institute Inc.

SAS® 9.4 Programmer's Guide: Essentials

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 2024

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 Language . . . **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 . . . 14
- Scope of This Book . . . 15
- Operating Environment Information . . . 15
- Using This Book . . . 16

Chapter 2 / SAS Processing . . . **17**
- Definition of SAS Processing . . . 17
- Types of Input to a SAS Program . . . 18
- The DATA Step . . . 20
- The PROC Step . . . 21
- SAS Processing Restrictions for Servers in a Locked-Down State . . . 22

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 . . . 33
- Generating Data from Programming Statements . . . 34
- DATA Step Processing Time . . . 35
- Stored Compiled DATA Step Programs . . . 37

PART 2 Syntax 39

Chapter 4 / Words and Names . . . **41**
- SAS Words . . . 42
- SAS Names . . . 44

SAS Name Literals . . . 49
Variable Names . . . 51
Member Names . . . 54
Data Set Names . . . 56
Examples: SAS Words and Names . . . 63

Chapter 5 / Variables . . . 75
Definition of SAS Variables . . . 77
Ways to Create Variables . . . 77
Creating a New Variable in a Formatted INPUT Statement . . . 79
Manage Variables . . . 80
Variable Attributes . . . 88
Data Types . . . 91
Variable Type Conversions . . . 94
Automatic Variables . . . 96
SAS Variable Lists . . . 99
Missing Variable Values . . . 102
Numeric Precision . . . 107
Examples: Create and Modify SAS Variables . . . 121
Examples: Control Output of Variables . . . 130
Examples: Reorder and Align Variables . . . 133
Examples: Convert Variable Types . . . 141
Examples: Use Automatic Variables . . . 145
Examples: Use Variable Lists . . . 148
Examples: Manage Missing Variable Values . . . 153
Examples: Manage Problems Related to Precision . . . 157
Examples: Encrypt Variable Values . . . 172

Chapter 6 / Data Types . . . 179
Data Types in SAS . . . 179
Data Types in SAS Viya . . . 179

Chapter 7 / SAS Expressions . . . 191
SAS Expressions . . . 191
SAS Constants in Expressions . . . 192
SAS Variables in Expressions . . . 193
SAS Functions in Expressions . . . 193
SAS Operators in Expressions . . . 193

Chapter 8 / SAS Constants . . . 195
SAS Constants in Expressions . . . 195
Examples: Expressions and Constants . . . 203

Chapter 9 / Operators . . . 215
SAS Operators . . . 215
Summary of Ways to Use Operators . . . 229
Examples: Operators . . . 231

Chapter 10 / Dates and Times . . . 241
Dates, Times, and Intervals . . . 241

Chapter 11 / Component Objects **243**
DATA Step Component Objects 243

PART 3 Accessing Data 245

Chapter 12 / SAS Libraries **247**
Definitions for SAS Libraries 248
Elements of a Library Assignment 250
Library Concatenation 255
SAS Default Libraries 256
Examples: Access Data by Using a Libref 259
Examples: Access Data without Using a Libref 272
Examples: View Information about a Library 278
Examples: Manage SAS Libraries 282

Chapter 13 / SAS Engines **289**
Definitions for SAS Engines 289
How Engines Work with Files 290
Engine Characteristics 292
Examples: Use a SAS Engine to Process SAS Data 298
Examples: Use a SAS Engine to Process External Data 308

Chapter 14 / SAS Data Sets **321**
Definitions for SAS Data Sets 322
Managing SAS Data Sets 323
Generation Data Sets 329
Examples: Create and Read SAS Data Sets 329
Examples: Control Variables and Observations in Data Sets 337
Examples: View Descriptor and Sort Information for Data Sets 346

Chapter 15 / Raw Data **353**
Definitions for Raw Data 353
Reading Raw Data 354
Definitions for External Files 369
Reading External Files 370
Examples: Read External Files Using PROC IMPORT 378

Chapter 16 / Database and PC Files **385**
Definition of SAS/ACCESS Software 385
Dynamic LIBNAME Engine 386
SQL Procedure Pass-Through Facility 387
ACCESS Procedure and Interface View Engine 389
DBLOAD Procedure 390
Interface DATA Step Engine 390

Chapter 17 / SAS Views **393**
Definitions for SAS Views 393

Chapter 18 / SAS Dictionary Tables **395**
Definition of a DICTIONARY Table 395

How to View DICTIONARY Tables . . . 396

PART 4 Manipulating Data 401

Chapter 19 / Grouping Data . . . 403
Definitions for BY-Group Processing . . . 404
Syntax . . . 405
Understanding BY Groups . . . 405
Invoking BY-Group Processing . . . 408
Determining Whether the Data Requires Preprocessing for BY-Group Processing . . . 409
Preprocessing Input Data for BY-Group Processing . . . 409
FIRST. and LAST. DATA Step Variables . . . 410
Processing BY-Groups in the DATA Step . . . 415

Chapter 20 / Loops and Conditionals . . . 421
Definitions for Loops and Conditionals . . . 422
Summary of Statements for Conditional Processing in SAS . . . 423
DO Loops . . . 425
WHERE Expressions . . . 428
IF Statements . . . 444
SELECT WHEN Statement . . . 446
Other Control Flow Statements . . . 447
Examples: DO Loops . . . 449
Examples: WHERE Processing . . . 457
Examples: IF Statements . . . 464
Example: SELECT WHEN Statement . . . 467
Example: Data Set Options . . . 469

Chapter 21 / Combining Data . . . 473
Overview of Combining Data . . . 474
Comparing Methods . . . 488
Examples: Prepare Data . . . 495
Examples: Concatenate Data . . . 510
Examples: Interleave Data . . . 518
Examples: Match-Merge Data . . . 525
Examples: Merge Data One-to-One . . . 542
Examples: Combine Data One-to-One . . . 550
Example: Merge Data Using a Hash Table . . . 552
Examples: Update Data . . . 556
Example: Modify Data . . . 563

Chapter 22 / Using Indexes . . . 571
Indexes in SAS . . . 571

Chapter 23 / Using Arrays . . . 573
Definitions for Array Processing . . . 574
Rules for Referencing Arrays . . . 575
A Conceptual View of Arrays . . . 576

Syntax for Defining and Referencing an Array 577
Processing Simple Arrays 578
Variations on Basic Array Processing 583
Multidimensional Arrays: Creating and Processing 584
Specifying Array Bounds 587
Examples 590

Chapter 24 / Debugging Errors 595
Definitions of Error Types in SAS 595
Error Processing in SAS 606
Error Checking When Using Indexes to Randomly Access or Update Data 615
Examples 625

Chapter 25 / Optimizing System Performance 635
Definitions for Optimizing System Performance 636
Collecting and Interpreting Performance Statistics 636
Techniques for Optimizing I/O 638
Techniques for Optimizing Memory Usage 646
Techniques for Optimizing CPU Performance 647
Calculating Data Set Size 649

Chapter 26 / Using Parallel Processing 651
Overview 651
What Is Threading Technology in SAS? 652
How Is Threading Controlled in SAS? 653
Threading in Base SAS 654
SAS/ACCESS Engines 657
SAS Scalable Performance Data Server 657
SAS Intelligence Platform 658
SAS High-Performance Analytics Portfolio of Products 659
SAS Grid Manager 660
SAS In-Database Technology 661
SAS In-Memory Analytics Technology 661
SAS High-Performance Analytics Product Integration 663
SAS Viya 665

PART 5 Creating Output 667

Chapter 27 / Output 669
Definitions for SAS Output 670
Default Output Destination 671
Change the Output Destination 672
Customize Output 674
The SAS Log 677
Examples: Manage Output Destinations 686
Examples: Rolling Over the SAS Log 688
Examples: Suppress Output to the SAS Log 692

Chapter 28 / ***Printing*** **697**
Universal Printing 697

PART 6 Managing Files 699

Chapter 29 / ***Protecting Files*** **701**
Definition of a Password 702
Assigning Passwords 703
Removing or Changing Passwords 705
Using Password-Protected SAS Files in DATA and PROC Steps 705
How SAS Handles Incorrect Passwords 706
Assigning Complete Protection with the PW= Data Set Option 707
Encoded Passwords 708
Using Passwords with Views 708
SAS Data File Encryption 711
Blotting Passwords and Encryption Key Values 716
Metadata-Bound Libraries 718

Chapter 30 / ***Repairing SAS Files*** **719**
Repairing Damage to SAS Data Sets and Catalogs 719
Examples 724

Chapter 31 / ***Compressing SAS Data Sets*** **733**
Compression in SAS 733

Chapter 32 / ***Moving SAS Files*** **735**
Moving Files between Operating Environments 735

Chapter 33 / ***Cross-Environment Data Access*** **737**
Definitions for Cross-Environment Data Access (CEDA) 737
Advantages of CEDA 738
SAS File Processing with CEDA 739
Alternatives to Using CEDA 744
Examples: CEDA 745

Chapter 34 / ***Managing SAS Catalogs*** **757**
Definition of a SAS Catalog 757

PART 7 SAS System Features 759

Chapter 35 / ***The SAS Registry*** **761**
Introduction to the SAS Registry 761
Managing the SAS Registry 765
Configuring Your Registry 776

Chapter 36 / ***The SAS Windowing Environment*** **781**
What Is the SAS Windowing Environment? 782

Main Windows in the SAS Windowing Environment . . . 782
Navigating in the SAS Windowing Environment . . . 792
Getting Help in SAS . . . 797
List of SAS Windows and Window Commands . . . 799
Introduction to Managing Your Data in the SAS Windowing Environment . . . 804
Managing Data with SAS Explorer . . . 804
Working with VIEWTABLE . . . 809
Subsetting Data By Using the WHERE Expression . . . 820
Exporting a Subset of Data . . . 824
Importing Data into a Table . . . 827

Chapter 37 / Cloud Analytic Services . . . **831**
What is SAS Cloud Analytic Services? . . . 831
What Does This Mean for the SAS 9 Programmer? . . . 831
SAS Language Elements for CAS . . . 832
CAS-specific Language Elements . . . 834

Chapter 38 / Industry Protocols Used in SAS . . . **835**
SAS Language Elements That Control SMTP E-Mail . . . 836
How the SMTP e-Mail Interface Authenticates Users . . . 838
Universally Unique Identifiers and the Object Spawner . . . 838
Using SAS Language Elements to Assign UUIDs . . . 841
Overview of IPv6 . . . 842
IPv6 Address Format . . . 843
Examples of IPv6 Addresses . . . 843
Fully Qualified Domain Names (FQDN) . . . 844

PART 8 Appendixes 847

Appendix 1 / Data Sets Used in Examples . . . **849**
Data Sets Used in Examples . . . 850
Description of Column-Binary Data Storage . . . 860

Appendix 2 / Understanding How the DATA Step Works . . . **863**
How the DATA Step Processes Data . . . 863
Processing a DATA Step: A Walk-Through . . . 866
More About DATA Step Execution . . . 871

Appendix 3 / Updating Data Using the MODIFY Statement and the KEY= Option . . . **879**
Updating Data Using the MODIFY Statement and the KEY= Option . . . 879

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 292.

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 Language 3

Chapter 2
SAS Processing 17

Chapter 3
DATA Step Processing 25

1

The SAS Language

About the SAS System 3
Definition of Base SAS Software 4
Components of Base SAS Software 4
Base SAS Language Definitions 5
SAS Files 5
External Files 6
Database Management System Files 6
SAS Language Elements 7
Global Statements 8
SAS Macro Facility 8
SAS Engines 8
Additional Languages 9
Anatomy of a SAS Program 10
Ways to Submit SAS Programs 12
Customizing Your SAS Session 14
Scope of This Book 15
Operating Environment Information 15
Using This Book 16

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 1.1 *Summary of SAS File Types*

File Type	Description
SAS data set	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.
SAS view	a virtual data set that extracts data values from other files in order to provide a customized and dynamic representation of the data. There are two types of views: DATA step view is a stored DATA step program.

File Type	Description
	PROC SQL view is a stored query expression that is created in PROC SQL.
SAS catalog	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.
SAS stored program	a type of SAS file that contains compiled code that you create and save for repeated use.
item store file	pieces of information that can be accessed independently. 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.

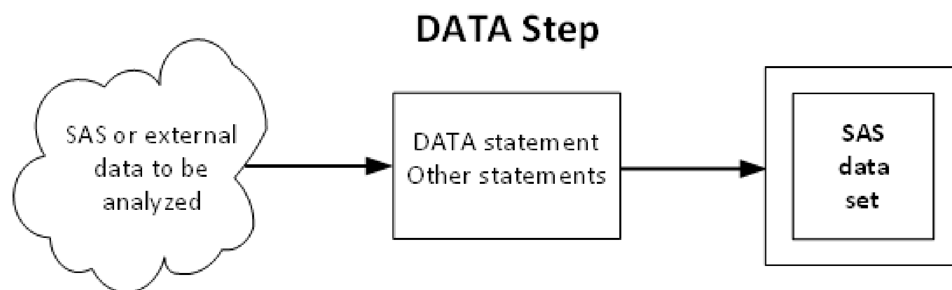
SAS Language Elements

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:

Figure 1.1 High-Level View of the SAS DATA Step



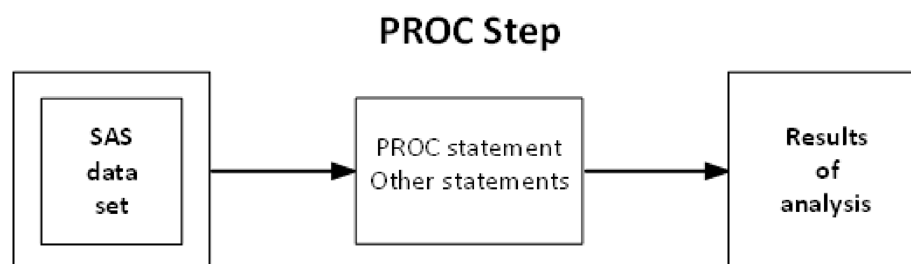
You typically use a DATA step to read data from an input source, process it, and create a SAS data set. In addition, you can perform the following tasks:

- 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:

Figure 1.2 High-Level View of the PROC Step



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 289.

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*

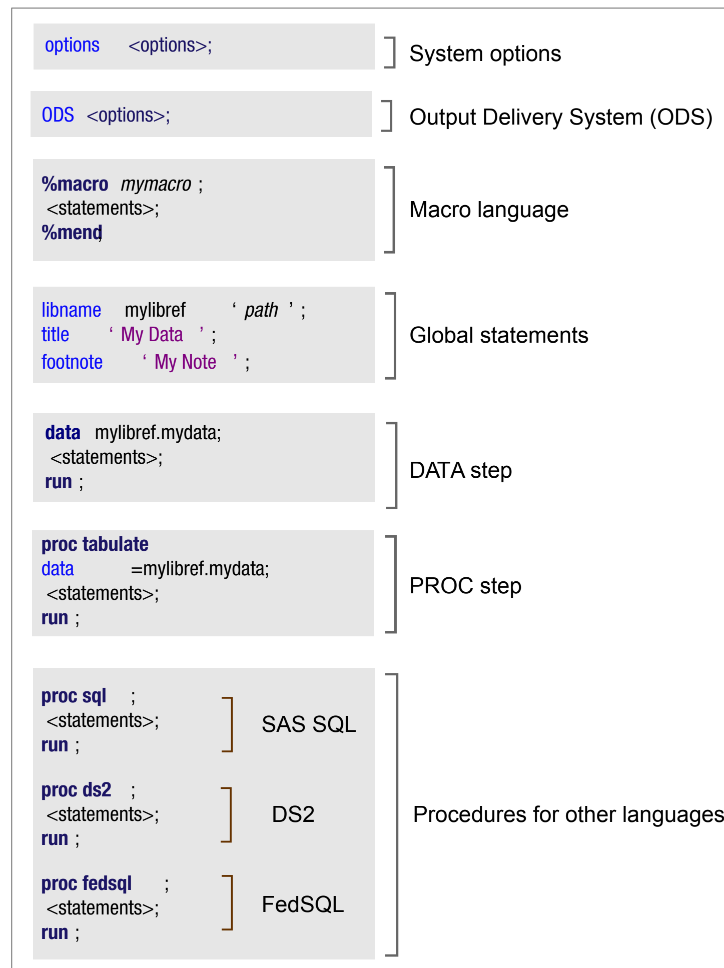
Language	Procedure	Description
SAS SQL	PROC SQL	a SAS implementation of the ANSI SQL: 1992 core standard. A benefit of SAS SQL is that it is fully integrated with the SAS language: it uses SAS data types, can be managed with SAS system options, and fully supports SAS data set options, formats, and informats. It supports explicit SQL pass-through to external DBMS when appropriate SAS/ACCESS software is licensed. SAS SQL statements are submitted from a SAS session by using the SQL procedure. For more information, see *SAS SQL Procedure User's Guide*.
SAS FedSQL	PROC FEDSQL	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. 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).

		For more information, see *SAS FedSQL Language Reference*
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:

Figure 1.3 Anatomy of 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

User Interface	Description	Documentation
SAS Studio	a web-based graphical user interface that enables you to write and run SAS code from your browser and display the results.	Getting Started with Programming in SAS Studio and SAS Studio: User's Guide
SAS Windowing Environment	a desktop graphical user interface that enables you to	Getting Started Under Windows in SAS Companion for Windows Working in the SAS Windowing Environment in SAS 9.4 Companion for UNIX Environments Overview of Windows in the z/OS Environment in SAS 9.4 Companion for z/OS

	write and run SAS code and display the results.	
SAS Enterprise Guide	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.	SAS Enterprise Guide Online Documentation and SAS Enterprise Guide Tutorial
Non-interactive batch mode	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.	Running SAS in Batch Mode in *SAS Companion for Windows Environments* "Noninteractive and Batch Modes in UNIX Environments" in *SAS Companion for UNIX Environments* Destinations of SAS Output Files and Directing SAS Log and Procedure Output in *SAS Companion for z/OS*
Interactive line mode	a mode in which you enter program	Use SAS Interactively or in Batch Mode in *SAS Companion for Windows Environments*

	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).	Interactive Line Mode in UNIX Environments in *SAS Companion for UNIX Environments*

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 Output Delivery System: User's Guide*
- *SAS Logging: Configuration and Programming Reference*
- *SAS National Language Support (NLS): Reference Guide*
- *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.

2

SAS Processing

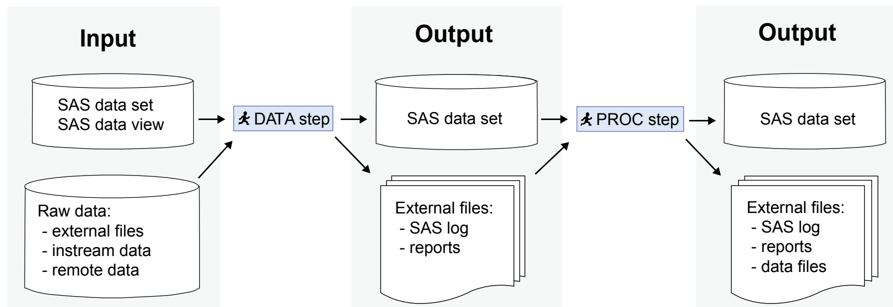
Definition of SAS Processing 17
Types of Input to a SAS Program 18
The DATA Step 20
Definition of the DATA Step 20
DATA Step Output 20
The PROC Step 21
What Does the PROC Step Do? 21
PROC Step Output 21
SAS Processing Restrictions for Servers in a Locked-Down State 22
General Information 22
z/OS Specific Information 23
Specifying Functions in the Lockdown Path List 24

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 669.

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:

Functions and CALL Routines	Access Methods	Procedures	Other
ADDR function	EMAIL	GROOVY procedure	DATA step Java object
ADDRLONG function	FTP	HADOOP procedure	
CALL MODULE	HADOOP	HTTP procedure	
CALL POKE routine	HTTP	JAVAINFO procedure	
CALL POKELONG routine	SOCKET	SOAP procedure	
PEEK function	TCPIP		
PEEKC function	URL		
PEEKCLONG function			
PEEKLONG function			

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 Administraton: 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	SOURCE
PDSCOPY	TAPECOPY
RELEASE	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)

3

DATA Step Processing

Why Use a DATA Step? 25
About Creating a SAS Data Set with a DATA Step 26
Creating a SAS Data Set or a SAS View 26
Sources of Input Data 26
Reading Raw Data: Examples 27
Reading External File Data 27
Reading Instream Data Lines 28
Reading Instream Data Lines with Missing Values 29
Using Multiple Input Files in Instream Data 30
Reading Data from SAS Data Sets 33
Example Code 33
Key Ideas 33
See Also 33
Generating Data from Programming Statements 34
Example Code 34
Key Ideas 35
See Also 35
DATA Step Processing Time 35
Stored Compiled DATA Step Programs 37

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:

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

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

```
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 353

Reference

- “DATA Statement” in *SAS DATA Step Statements: Reference*
- “INFILE Statement” in *SAS DATA Step Statements: Reference*
- “INPUT Statement” in *SAS DATA Step Statements: Reference*

Reading Instream Data Lines

Example Code

This example reads raw data from instream data lines.

```
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 353

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.

```
data weight2;  1
   infile datalines missover; 2
   input IDnumber $ Week1 Week16; 3
   WeightLoss2=Week1-Week16;
   datalines;
2477 195  163 4
2431
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 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 353

Reference

- “INFILE Statement” in *SAS DATA Step Statements: 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.

```
data all_errors; 1
   length filelocation $ 60; 2
   input filelocation; 3
   infile daily filevar=filelocation 4
                filename=daily 5
                end=done; 6
   do while (not done); 7
      input Station $ Shift $ Employee $ NumberOfFlaws;
      output;
   end;
   put 'Finished reading ' daily=;
   datalines;
pathmyfile_A 8
pathmyfile_B
```

```
pathmyfile_C
;

proc sort data=all_errors out=sorted_errors; 9
   by Station;
run;

proc print data = sorted_errors; 10
   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.

7 Use the DOWHILE 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.

8 Specify the file pathnames following the DATALINES statement.

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

10 The print procedure prints the results.

Output 3.1 *Multiple Input Files in Instream Data*

Flaws Report sorted by Station

Obs	Station	Shift	Employee	NumberOfFlaws
1	Amherst	2	Lynne	0
2	Goshen	2	Seth	4
3	Hadley	2	Jon	3
4	Holyoke	1	Walter	0
5	Holyoke	1	Barb	3
6	Orange	2	Carol	5
7	Otis	1	Kay	0
8	Pelham	2	Mike	4
9	Stanford	1	Sam	1
10	Suffield	2	Lisa	1

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 353

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;   1
   set weight;   2
   Percent=round((AverageLoss * 100) / Week1);   3
run;   4
```

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 321

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 321
- Chapter 12, “SAS Libraries,” on page 247
- Chapter 13, “SAS Engines,” on page 289
- Chapter 16, “Database and PC Files,” on page 385.

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 863.

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 647.

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 638.

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 635.

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*

```
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 41

Chapter 5
Variables 75

Chapter 6
Data Types 179

Chapter 7
SAS Expressions 191

Chapter 8
SAS Constants 195

Chapter 9
Operators 215

Chapter 10
Dates and Times 241

Chapter 11
Component Objects 243

4

Words and Names

SAS Words 42
- Definition of a SAS Word 42
- Types of Words or Tokens 42
- Placement and Spacing of Words 43

SAS Names 44
- Definition of a SAS Name 44
- Rules for Most SAS Names 44
- Length Rules for Names 45
- Extending SAS Names 47
- See Also 48

SAS Name Literals 49
- Definition of SAS Name Literals 49
- Important Restrictions 50
- Using Name Literals in BY Groups 51
- Avoiding Errors When Using Name Literals 51
- See Also 51

Variable Names 51
- Rules for Variable Names 51
- Extended Rules for SAS Variables 52
- See Also 54

Member Names 54
- Rules for Member Names 54
- Extended Rules for SAS Member Names 55
- See Also 56

Data Set Names 56
- Definition of a Data Set Name 56
- Structure of a Data Set Name 57
- Special Data Set Names 59
- Rules for Naming SAS Data Sets and Variables 60
- Extended Rules for Naming SAS Data Sets and Variables 61
- See Also 62

Examples: SAS Words and Names 63
- Example: Create a Variable Name Containing Blanks 63
- Example: Create a SAS Data Set Name Containing a Special Character 64
- Example: Manage Placement and Spacing of Words 65
- Example: Specify the FIRST. and LAST. BY Variables as Name Literals 66

Example: Create a One-Level Data Set Name . . . 68
Example: Create a Two-Level Data Set Name . . . 70
Example: Use the Automatic Naming Feature for Naming Data Sets . . . 71
Example: Use the _LAST_= System Option to Specify the Default Input Data Set . 72

SAS Words

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 46. 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, 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
: 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 each token due to cues such as operators, you do not need to include blanks. If SAS cannot determine the beginning of each token, you must use blanks.

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*:

Note: The rules are more flexible for SAS variable names, data set names, view names, and item store names than for other language elements. See Rules for Variable Names and Rules for Member 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

Language Element Names	Maximum Length in Bytes
Array names	32
CALL routines names	16
Catalog names	32
Component object names	32
Engine names	8
Fileref names	8
Format names, character	31
Format names, numeric	32
Function names	16
Generation data set names	28
Index names	32
Informat names, character	30
Informat names, numeric	31
Librefs (library names)	8
Library Member names[1]	32
Macro variable names	32
Macro window names	32
Macro names	32
Password names	8
Procedure names (the first eight characters must be unique and cannot begin with “SAS”)	16
Statement label names	32

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*.

Language Element Names	Maximum Length in Bytes
Variable (includes DATA step variables and SCL variables) names	32
Variable label names	256
View names	32
Window names	32

Extending SAS Names

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

System Option Setting	SAS Name Measured in Bytes	Maximum Length in Bytes
VALIDVARNAME=ANY	variable names	32
VALIDMEMNAME=EXTEND	SAS data set name SAS view name item store name	32

When these system option values are set, the maximum number of characters that you can use for a SAS member name is determined by the number of bytes that are used to store one character. This value is set by the SAS encoding value for your SAS session. For more information, see "Determining the Encoding of a SAS Data Set" in *SAS National Language Support (NLS): Reference Guide*. VALIDVARNAME=ANY or VALIDMEMNAME=EXTEND must be set to allow the use of *National Language Support* (NLS) characters.

When these system options are not set to ANY or EXTEND, the default session encoding is used and only one-byte characters are allowed. For more information about the default session encoding in SAS, see "Default SAS Session Encoding" in *SAS National Language Support (NLS): Reference Guide*.

Note: A *SAS member name* includes names of data sets, DATA step views, PROC SQL views, catalogs, stored, compiled DATA step programs, and item stores.

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:

     ```
     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*.

See Also

Examples

- "Example: Create a Variable Name Containing Blanks"
- "Example: Manage Placement and Spacing of Words"

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*

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*.

When the VALIDVARNAME system option is set to V=7, the first character of a SAS variable name 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.

Name literals enable you to use many more characters in the name, including blanks, national characters, and numerals as the first character in the name. To use a numeral as the first character in the name, however, you must set the VALIDVARNAME= option to ANY:

```
options validvarname=any;
data test;
   "1ABC"n="hello";
run;
```

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 4.2 *SAS Name Literal System Option Requirements*

SAS Name Type	Name Literal Requirements
DBMS column	Set VALIDVARNAME=ANY.
DBMS table name	Set VALIDMEMNAME=EXTEND.

SAS Name Type	Name Literal Requirements
item store	Set VALIDMEMNAME=EXTEND.
SAS data set name	Set VALIDMEMNAME=EXTEND.
SAS view	Set VALIDMEMNAME=EXTEND.
statement label	Set VALIDVARNAME=ANY.
variable	Set VALIDVARNAME=ANY.

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 *n* 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

Variable Names **(with VALIDVARNAME=V7)**
■ can be up to 32 bytes in length. ■ cannot contain special characters (except for the underscore), blanks, or national characters. ■ must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore. ■ 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. 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

Variable Names **(with VALIDVARNAME=UPCASE)**
■ can be up to 32 bytes in length. ■ cannot contain special characters (except for the underscore), blanks, or national characters. ■ must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore.

Variable Names

(with VALIDVARNAME=UPCASE)

- 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.

  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*

Variable Names

(with VALIDVARNAME=ANY)

- can be up to 32 bytes in length.
- can contain special characters including / \ * ? " < > | : - . A name that contains special characters must be specified as a name literal.
- can begin with any character, including blanks, national characters, special characters, and multibyte characters.
- preserves leading blanks, but trailing blanks are ignored.
- 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.

  Note that SAS converts all variable names to uppercase.

  For example, the names `cat`, `Cat`, and `CAT` all represent the same variable.
- cannot contain all blanks.

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 VALIDVARNAME=V7 and you use any characters other than letters, 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

- "VALIDMEMNAME= System Option" in *SAS System Options: Reference*
- "VALIDVARNAME= System Option" in *SAS System Options: Reference*

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

Member Names (with VALIDMEMNAME=COMPATIBLE)
■ The length of the names can be up to 32 characters. ■ 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. ■ Names cannot contain blanks or special characters except for the underscore.

Member Names (with VALIDMEMNAME=COMPATIBLE)
■ 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.

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.

Variable Names (with VALIDMEMNAME=EXTEND)
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 54 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 59.

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

- DATA= option
- MERGE statement

- MODIFY statement
- OPEN function
- SET statement
- SQL Procedure
- UPDATE statement

SAS view names are created using one of the following:

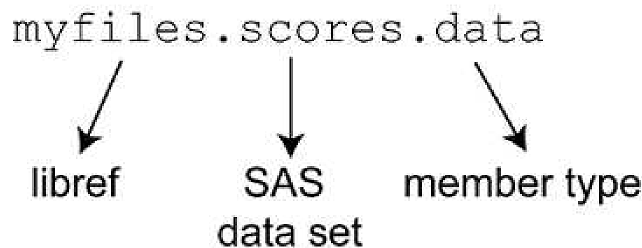
- the VIEW= option in the "DATA Statement" in *SAS DATA Step Statements: Reference*
- the ACCESS procedure

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 (library name)
: 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

SAS data set

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

Work
: a SAS 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.

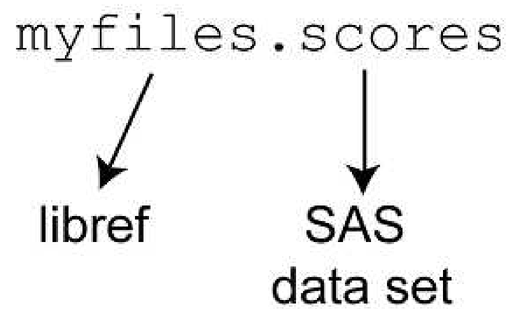
User
: a SAS library that is used for permanently saving data sets with one-level names.

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:



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

Reserved words are keywords in SAS that are “reserved” for system use only and cannot be used to name your data sets. The following special data set names are *reserved words* in SAS:

- _DATA_
- _LAST_
- _NULL_

DATA Data Set

If you do not specify a name for the output data set or specify the reserved name _DATA_ in a DATA statement, then SAS automatically assigns the names DATA1, DATA2, ... DATA9999, after which time the names restart with DATA1 for each successive data set that you create.

This feature is referred to as the DATA*n* naming convention.

For example, when the following program executes, SAS creates three data sets in the WORK library, naming them DATA1, DATA2, and DATA3:

```
data _data_;
   x=1;
run;

data;
   y=2;
run;

data _data_;
   z=3;
run;
```

This feature enables you to automate the naming of output data sets without the risk of overwriting your data. The names are unique when they are written to the

WORK library, and they continue to increment numerically for the duration of the SAS session.

See Splitting a data set into smaller data sets for some practical examples that show how to use the SAS DATA*n* naming convention.

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 72 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:

```
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

SAS Data Set Names, View Names, and Item Store Names (with VALIDMEMNAME=COMPAT)	Variable Names (with VALIDVARNAME=V7)
■ can be up to 32 bytes in length. ■ cannot contain special characters (except for the underscore), blanks, or national characters. ■ must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore. ■ 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.	■ can be up to 32 bytes in length. ■ cannot contain special characters (except for the underscore), blanks, or national characters. ■ must begin with a letter from the Latin alphabet (A–Z, a–z) or the underscore. ■ 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.

IMPORTANT In the Linux operating environment, SAS reads only data set names that are written in all lowercase characters.

Extended Rules for Naming SAS Data Sets and Variables

The following table shows a summary of the rules for naming SAS data sets (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>
<tr><th>SAS Data Set Names, View Names, and Item Store Names
(with VALIDMEMNAME=EXTEND)</th><th>Variable Names
(with VALIDVARNAME=ANY)</th></tr>
<tr><td>
- can be up to 32 bytes in length.
- can contain special characters except for / \ * ? " < > | : -. A name that contains special characters must be specified as a name literal.
- cannot begin with a blank or a period.
- ignores leading and trailing blanks.
- 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. [1]
</td><td>
- can be up to 32 bytes in length.
- can contain special characters including / \ * ? " < > | : - . A name that contains special characters must be specified as a name literal.
- can begin with any character, including blanks, national characters, special characters, and multibyte characters.
- preserves leading blanks, but trailing blanks are ignored.
- 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.
- cannot contain all blanks.
</td></tr>
</table>

IMPORTANT In the Linux operating environment, SAS reads only data set names that are written in all lowercase characters.

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"

1. In the UNIX operating environment, SAS reads only data set names that are written in all lowercase characters

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

Functions

- OPEN function

Statements

- DATA Statement
- SET statement
- MERGE statement
- MODIFY statement
- “OPTIONS Statement” in *SAS Global Statements: Reference*
- UPDATE statement

System Options

- _LAST_= system option
- USER= system option
- “VALIDMEMNAME= System Option” in *SAS System Options: Reference*
- “VALIDVARNAME= System Option” in *SAS System Options: Reference*
- VIEW= option

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

```
options validvarname=any;   /*1*/
data mydata;
   'First Name'n='John';    /*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 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 $:

```
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*.

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.

```
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.

```
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.

```
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 42
- “Types of Words or Tokens”
- “Rules for Most SAS Names” on page 44

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:

```
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:

```
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*

```
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.
NOTE: DATA statement used (Total process time):
      real time           0.06 seconds
      cpu time            0.01 seconds
```

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 in the SAS DATA step. If you do not specify a name for the output data set or specify it as DATA (or data as it is case insensitive), SAS automatically assigns names to the data sets that you create as DATA1, DATA2, and so on up to DATA9999. SAS assigns a successive name to every unnamed data set.

The example contains three unnamed output data sets and one named data set.

```
data;
   setsashelp.class;
run;

data DATA;
   set sashelp.air;
run;

data sample;
   set sashelp.cars;
run;

data data;
   setsashelp.flags;
run;
```

The following is printed to the SAS log.

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.DATA1 has 19 observations and 5 variables.
NOTE: There were 144 observations read from the data set SASHELP.AIR.
NOTE: The data set WORK.DATA2 has 144 observations and 2 variables.
NOTE: There were 428 observations read from the data set SASHELP.CARS.
NOTE: The data set WORK.SAMPLE has 428 observations and 15 variables.
NOTE: There were 220 observations read from the data set SASHELP.FLAGS.
NOTE: The data set WORK.DATA3 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

- _DATA_ 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

5

Variables

Definition of SAS Variables 77

Ways to Create Variables 77

Creating a New Variable in a Formatted INPUT Statement 79

Manage Variables 80
- Modify Variables 80
- Specifying a New Variable in a FORMAT or INFORMAT Statement 82
- Specifying a New Variable in the ATTRIB Statement 84
- Control Output of Variables 85

Variable Attributes 88
- Definition 88
- See Also 90

Data Types 91
- Definitions 91
- Numeric Data 92
- Character Data 93

Variable Type Conversions 94
- Automatic Character-to-Numeric Conversion 94
- Explicit Character-to-Numeric Conversion 95
- Automatic Numeric-to-Character Conversion 95
- Explicit Numeric-to-Character Conversion 96
- See Also 96

Automatic Variables 96
- Automatic DATA Step Variables 97
- Automatic Macro Variables 98
- See Also 98

SAS Variable Lists 99
- Definition of a Variable List 99
- Types of Variable Lists 99
- The OF Operator with Functions and Variable Lists 100
- See Also 101

Missing Variable Values 102
- Definition of Missing Values 102
- How to Represent Missing Values in Raw Data 102
- Special Missing Values 103
- Order of Missing Values 103

When Variable Values Are Automatically Set to Missing by SAS 105
When Missing Values Are Generated by SAS 106
See Also 107

Numeric Precision **107**
Overview 107
Truncation in Binary Numbers 108
How SAS Stores Numeric Values 108
Floating-Point Representation Using the IEEE Standard 113
Floating-Point Representation on Windows 114
Floating-Point Representation on IBM Mainframes 116
Troubleshooting Errors in Precision 117
Transferring Data between Operating Systems 119
See Also 120

Examples: Create and Modify SAS Variables **121**
Example: Create a SAS Variable Using an Assignment Statement 121
Example: Create a Variable Using the INPUT Statement 122
Example: Modify a SAS Variable Using the LENGTH Statement 123
Example: Modify a SAS Variable Using the FORMAT and INFORMAT Statement . 124
Example: Modify a SAS Variable Using the ATTRIB Statement 125
Example: View Variable Attributes 126
Example: Change Variable Attributes 128

Examples: Control Output of Variables **130**
Example: Select Specific Variables for Output 130
Example: Select Specific Variables for Processing 132

Examples: Reorder and Align Variables **133**
Example: Reorder Variables Using the ATTRIB Statement 133
Example: Reorder Variables Using the LENGTH Statement 135
Example: Reorder Variables Using the RETAIN Statement 137
Example: Reorder Variables Using the FORMAT Statement 139

Examples: Convert Variable Types **141**
Example: Convert Character Variables to Numeric 141
Example: Convert Numeric Variables to Character 142
Example: Use Automatic Type Conversions 144

Examples: Use Automatic Variables **145**
Example: Use the _N_ Automatic Variable 145
Example: Use the _ERROR_ and _INFILE_ Automatic Variable with the IF/THEN Statement 147

Examples: Use Variable Lists **148**
Example: Create a Variable Numbered Range List 148
Example: Create a Variable Name Range List 149
Example: Use the OF Operator with a Variable List 151
Example: Use the OF Operator to Create Multiple Variable Lists 152

Examples: Manage Missing Variable Values **153**
Example: Automatically Replacing Missing Values 153
Example: Creating Special Missing Values 154
Example: Preventing Propagation of Missing Values 155

Examples: Manage Problems Related to Precision **157**
Example: Compare Imprecise Values in SAS 157
Example: Convert a Decimal Value to a Floating Point Representation 158

Example: Convert a Decimal Value to a Hexadecimal Floating-Point Representation 161
Example: Round Values to Avoid Computational Errors 163
Example: Use the LENGTH Statement to Compare Values 164
Example: Compare Values That Have Imprecise Representations 166
Example: Confirm Precision Errors Using Formats 167
Example: Determine How Many Bytes Are Needed to Store a Number Accurately 169
Example: Generate Inaccurate Results For Large Data 171

Examples: Encrypt Variable Values **172**
Example: Create a Simple 1-Byte-to-1-Byte Swap Using the TRANSLATE Function 172
Example: Use a 1-Byte-to-2-Byte Swap Using the TRANWRD Function 174
Example: Use Different Functions to Encrypt Numeric Values as Character Strings 176

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 5.1 Ways to Create Variables

Method	Example	Explanation

Assignment statement	`data vars;`
	`a='Tom Hanks';`
	`a='Rita Wilson';`
	`put a;`
	`run;`
	Example: Create a SAS Variable Using an Assignment Statement
		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`**.

	`data vars;`
	`a=36;`
	Example: Create a SAS Variable Using an Assignment Statement
		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.

	`data vars;`
	`length a $ 4 b $ 6 c $ 2;`
	`x=a||b||c;`
	`run;`
		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`.

INPUT statement, List	`data vars;`
	`input Gem $`
	`Carats`
	`Color $;`
	`datalines;`
	`emerald 2 green`
	`;`
	Example: Create a Variable Using the INPUT Statement
		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.

<table>
<tr><th>Method</th><th>Example</th><th>Explanation</th></tr>
<tr><td></td><td></td><td>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>INPUT statement, Formatted</td><td><pre>data vars;
   input Gem $char10.
         Carats comma3.1
         Color $char.;
datalines;
emerald 2 green
;</pre>Example: Create a Variable Using the INPUT Statement</td><td>The INPUT statement creates two character variables: <code>Gem</code> is created with a length specified in its informat, and <code>Color</code> is created without a length specification. The numeric variable <code>Carats</code> 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, <code>Carats</code> is 8 bytes. The length of the character variable, <code>Gem</code>, is 10 bytes. The length of the character variable, <code>Color</code>, 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>
</table>

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 5.2 *Ways to Modify Variables*

Method	Example	Explanation

FORMAT or INFORMAT statement	`Insurance=100.28;`
	`Fee=20.00;`
	`format Insurance Fee dollar10.2;`
	Example: FORMAT Statement
		You can create new variables and specify their format or informat with a FORMAT or INFORMAT statement.[1]
		The FORMAT statement creates two variables `Insurance` and `Fee` with a format of dollar 10.2

LENGTH statement	`length Name $20;` Example: LENGTH Statement	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.

Method	Example	Explanation
		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.
ATTRIB statement	`attrib Class format=$12.;` Example: ATTRIB Statement	The ATTRIB statement enables you to specify one or more of the following variable attributes for an existing variable: ■ FORMAT= ■ INFORMAT= ■ LABEL= ■ LENGTH= If the variable does not already exist, one or more of the FORMAT=, INFORMAT=, and LENGTH= attributes can be used to create a new variable. 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 Creating New Variables Using the FORMAT Statement Without Specifying Length on page 82 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 (COMMAwd), 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 $UPCASEw. 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)*

```
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				
#	Variable	Type	Len	Format
2	Amount	Num	8	COMMA.
1	Flavor	Char	8	$UPCASE.

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

```
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				
#	Variable	Type	Len	Format
2	Amount	Num	8	COMMA10.
1	Flavor	Char	10	$UPCASE10.

In the example, Example Code 5.3 on page 83, 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*

```
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*

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Format
2	Amount	Num	8	COMMA10.
1	Flavor	Char	6	$UPCASE10.

See Also

- Table 5.5 on page 88 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 5.3 *Statements versus Data Set Options for Dropping, Keeping, and Renaming Variables*

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

Using the Input or Output Data Set

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

is one that is specified in the DATA statement.) Consider the following facts when you make your decision:

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

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

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

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

Where Specified	Data Set Option or Statement	Purpose	Status of Variable or Variable Name
	RENAME=	changes name of variables in individual output data sets	use old name in program statements and other output data set options

Order of Application

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*

Variable Attribute	Definition	Possible Values	Default Value
Name	identifies a variable. A variable name must conform to SAS naming rules.	Any valid SAS name.	None
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.	2 to 8 bytes for numeric. 1 to 32,767 bytes for character.	8 bytes for numeric and character

Variable Attribute	Definition	Possible Values	Default Value
	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.		
Format	refers to the instructions that SAS uses when printing variable values. If no format is specified, the default format is BEST12. for a numeric variable, and $w. for a character variable. You can assign SAS formats to a variable in the FORMAT or ATTRIB statement.	See Dictionary of Formats in SAS Formats and Informats: Reference	BEST12. for numeric, $w. for character
Informat	refers to the instructions that SAS uses when reading data values. If no informat is specified, the default informat is w.d for a numeric variable, and $w. for a character variable. You can assign SAS informats to a variable in the INFORMAT or ATTRIB statement. You can also assign an informat in the INPUT statement or INPUT function.	See About Informats in SAS Formats and Informats: Reference	w.d for numeric, $w.for character
Label	refers to a descriptive label up to 256 characters long. A variable label, which can be printed by some SAS procedures, is	Up to 256 characters.	None

Variable Attribute	Definition	Possible Values	Default Value
	useful in report writing. You can assign a label to a variable with a LABEL or ATTRIB statement.		
Position in observation	is determined by the order in which the variables are defined in the DATA step.	1- n	None
Index type	indicates whether the variable is part of an index for the data set.	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.	None
Extended attribute (user-defined)	is a user-defined attribute that is created using the XATTR ADD VAR statement in the DATASETS procedure.	Numeric or character	None

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 92 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*

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

Table 5.7 *Reading Nonstandard Numeric Data*

Data	Description	Solution
2 3	embedded blank	COMMA. or BZ. informat
- 23	embedded blank	COMMA. or BZ. informat
2,341	comma	COMMA. informat
(23)	parentheses	COMMA. informat
C4A2	hexadecimal value	HEX. informat

Data	Description	Solution
`1MAR90`	date value	DATE. informat

Table 5.8 Reading Invalid Numeric Data

Data	Description	Solution
`23 -`	minus sign follows number	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 TRAILSGN informat.
`..`	double instead of single periods	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.
`J23`	not a number	Read as a character value, or edit the raw data to change it to a valid number.

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

Characters in the Data	What to Use	Reason
leading or trailing blanks that you want to preserve	formatted input and the $CHARw. informat	List input trims leading and trailing blanks from a character value

Characters in the Data	What to Use	Reason
		before the value is assigned to a variable.
semicolons in instream data	DATALINES4 or CARDS4 statements and four semicolons (;;;;) to mark the end of the data	With the normal DATALINES and CARDS statements, a semicolon in the data prematurely signals the end of the data.
delimiters, blank characters, or quoted strings	DSD option, with DLM= or DLMSTR= option in the INFILE statement	These options enable SAS to read a character value that contains a delimiter within a quoted string; these options can also treat two consecutive delimiters as a missing value and remove quotation marks from character values.

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`.

```
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*

```
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.

```
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.

See Also

Examples

- “Example: Convert Character Variables to Numeric”
- “Example: Convert Numeric Variables to Character”
- “Example: Use Automatic Type Conversions”

Functions

- “INPUT Function” in *SAS Functions and CALL Routines: Reference*
- PUT Function

Automatic Variables

Automatic variables are system variables that SAS creates automatically when a session is started. SAS creates both automatic macro variables and automatic DATA step variables. 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.

Automatic DATA Step Variables

When you run the DATA step, SAS creates variables that contain information about the data being processed by the DATA step as well as processing information such as return codes for error processing. 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 66

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 66

Automatic Macro Variables

Automatic macro variables are variables that are created by the SAS macro facility rather than by the user. When you start a session, SAS creates automatic macro variables and assigns values to them.The values provide information about your SAS session, such as the date and time your session began, the operating system, and the version of SAS that you are running. For example, the SYSPROCESSID automatic macro variable contains the process ID of the current SAS process. The following statement displays the time at which a SAS session started.

```
%put This SAS session started running at: &systime;
```

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

Type	Definition	Example
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	`sum(of Sales:)`

Type	Definition	Example
	variables that begin with a specified character string.	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_ specifies all numeric variables that are already defined in the current DATA step. _CHARACTER_ specifies all character variables that are already defined in the current DATA step. _ALL_ specifies all variables that are already defined in the current DATA step.	_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 5.10 SAS Variable Lists Used with the OF Operator

Type	Example	Description
Name range lists	Function(OF x-*character*-a)	Performs the function on all the character variables from x to a inclusive.
Name prefix lists	Function(OF x:)	Performs the function on all the variables that begin

		with “x” such as “x1”, “x2” and so on.
Numbered range lists	Function(OF x1 – x*n*)	Performs the function on variable values between x1 and x*n* inclusive.[1]
Arrays	Function(OF *array-name*(*))	Performs the function on the named array.[2]
Special SAS name lists	Function(OF _numeric_)	Performs the function on the _numeric_variable, which specifies all numeric variables that are already defined in the current DATA step.

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

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*.

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

Missing Values	Representation in Data
Numeric	. (a single decimal point)
Character	' ' (a blank enclosed in quotation marks)
Special	. letter (a decimal point followed by a letter, for example, .B)
Special	._ (a decimal point followed by an underscore)

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:

  ```
  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

Sort Order	Symbol	Description
smallest	._	underscore
	.	period
	.A-.Z	special missing values A (smallest) through Z (largest)
	-n	negative numbers
	0	zero
largest	+n	positive numbers

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 `(.333...)`. 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 `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 ...
```

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 x 0.1)` is expected to be equal to `0.3`, so the difference between `(3 x 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, 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 5.13 Largest Integer That Can Be Safely Stored in a Given Length

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

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*.

mantissa exponent

$$987 = .987 \times 10^{3}$$

base

- 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.

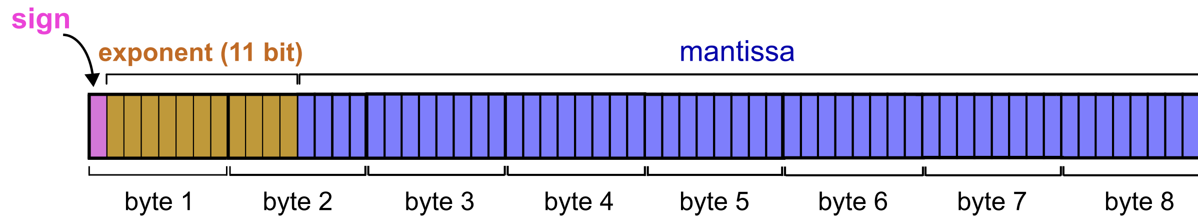
sign exponent mantissa

987 = 0 100 0100 0111 0110

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 111 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

Specifications	IBM Mainframe	IEEE Standard (Windows/UNIX)	Affects
Base	16	2	magnitude
Exponent Bits	7	11	magnitude
Mantissa bits	56	52	precision
Round or Truncate	Truncate	Round	precision
Bias for Exponent	64	1023	

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:

Base 16						
16^7	...	16^4	16^3	16^2	16^1	16^0
268,435,456	...	65,536	4096	256	16	1

$$3000 = (\mathbf{B} \times 16^2) + (\mathbf{B} \times 16^1) + (\mathbf{8} \times 16^0)$$
$$= (\mathbf{11} \times 256) + (\mathbf{11} \times 256) + (\mathbf{8} \times 1)$$

  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:

Base 2						
2^7	...	2^4	2^3	2^2	2^1	2^0
128	...	16	8	4	2	1

$$184 = (\mathbf{1} \times 2^7) + (\mathbf{0} \times 2^6) + (\mathbf{1} \times 2^5) + (\mathbf{1} \times 2^4) + (\mathbf{1} \times 2^3) + (\mathbf{0} \times 2^2) + (\mathbf{0} \times 2^1) + (\mathbf{0} \times 2^0)$$
$$= 128 + 0 + 32 + 16 + 8 + 0 + 0 + 0$$

  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:

S E E E E E E E	E E E E M M M M	M M M M M M M M	M M M M M M M M
Byte 1	Byte 2	Byte 3	Byte 4

M M M M M M M M	M M M M M M M M	M M M M M M M M	M M M M M M M M
Byte 5	Byte 6	Byte 7	Byte 8

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

- The S in byte 1 is the sign bit of the number. A value of 0 in the sign bit is used to represent positive numbers.

- The 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:

```
data _null_;
   x=.500000000000000000000000;
   y=.500000000000000000000000000;
   if x=y then put 'equal';
   else put 'not equal';
run;
```

Log Output

```
not equal
```

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=3FDFFFFFFFFFFFFF
```

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=.500000000000000;
   if x=y then put 'equal';
   else put 'not equal';
   put x=hex16./
   y=hex16.;
run;
```

Log Output

```
equal
x=3FE0000000000000
y=3FE0000000000000
```

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:

S E E E E E E E	M M M M M M M M	M M M M M M M M	M M M M M M M M
Byte 1	Byte 2	Byte 3	Byte 4

M M M M M M M M	M M M M M M M M	M M M M M M M M	M M M M M M M M
Byte 5	Byte 6	Byte 7	Byte 8

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

- The S in byte 1 is the sign bit of the number. A value of 0 in the sign bit is used to represent positive numbers.
- The seven E characters in byte 1 represent a binary integer known as the characteristic. The characteristic represents a signed exponent and is obtained by adding the bias to the actual exponent. The bias is an offset used to 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:  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 (`.3333 …`). 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:

```
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 5.15 *Table Representation of the Numbers 256 to 272 in 8 Bytes*

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

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 = 1/3
```

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

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

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

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

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 `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`RB8.` informat corresponds to a full 8-byte floating point. The `RB4`. informat corresponds to an 8-byte floating point truncated to 4 bytes, exactly the same as a LENGTH 4 in the DATA step.

An 8-byte floating point that is truncated to 4 bytes might not be the same as 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 `.54E−78` to `.72E76` (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 1E308). Therefore, if you are transferring numbers in the magnitude of 1E100 from a PC to a mainframe, you lose that magnitude. During data transfer, the number is set to the minimum or maximum allowable on that operating system, so 1E100 on a PC is converted to a value that is approximately .72E76 on an IBM mainframe.

CAUTION
Transfer of data between 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*

Formats and Informats

- "HEXw. Format" in *SAS Formats and Informats: Reference*
- "Dictionary of Formats" in *SAS Formats and Informats: 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.

```
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.

```
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*

Obs	Sale_Price	Date
1	$49.99	January 10, 2019
2	$29.99	January 14, 2019
3	$32.59	January 16, 2019

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`.

```
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*

Flying Miles Between Ten US Cities

Obs	ATL	ORD	DEN	IAH	LAX	MIA	JFK	SFO	SEA	DCA	City
1	0	.	.	.	.	.	.	.	.	.	Atlanta
2	587	0	.	.	.	.	.	.	.	.	Chicago
3	1,212	920	0	.	.	.	.	.	.	.	Denver
4	701	940	879	0	.	.	.	.	.	.	Houston
5	1,936	1,745	831	1,374	0	.	.	.	.	.	Los Angeles
6	604	1,188	1,726	968	2,339	0	.	.	.	.	Miami
7	748	713	1,631	1,420	2,451	1,092	0	.	.	.	New York
8	2,139	1,858	949	1,645	347	2,594	2,571	0	.	.	San Francisco
9	2,182	1,737	1,021	1,891	959	2,734	2,408	678	0	.	Seattle
10	543	597	1,494	1,220	2,300	923	205	2,442	2,329	0	Washington D.C.

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).

Output 5.6 *Partial Output: PROC CONTENTS*

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
1	Atlanta	Num	8	COMMA8.	ATL
2	Chicago	Num	8	COMMA8.	ORD
11	City	Char	15		
3	Denver	Num	8	COMMA8.	DEN
4	Houston	Num	8	COMMA8.	IAH
5	LosAngeles	Num	8	COMMA8.	LAX
6	Miami	Num	8	COMMA8.	MIA
7	NewYork	Num	8	COMMA8.	JFK
8	SanFrancisco	Num	8	COMMA8.	SFO
9	Seattle	Num	8	COMMA8.	SEA
10	WashingtonDC	Num	8	COMMA8.	DCA

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.
- 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.

See Also

- ATTRIB Statement
- Variable Attributes

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.

```
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*

Obs	Make	Model	Type	DriveTrain	Invoice	MPG_City	MPG_Highway
1	Acura	NSX coupe 2dr manual S	Sports	Rear	$79,978	17	24
2	Audi	RS 6 4dr	Sports	Front	$76,417	15	22
3	Cadillac	XLR convertible 2dr	Sports	Rear	$70,546	17	25
4	Dodge	Viper SRT-10 convertible 2dr	Sports	Rear	$74,451	12	20
5	Jaguar	XKR coupe 2dr	Sports	Rear	$74,676	16	23
6	Jaguar	XKR convertible 2dr	Sports	Rear	$79,226	16	23
7	Mercedes-Benz	G500	SUV	All	$71,540	13	14
8	Mercedes-Benz	CL500 2dr	Sedan	Rear	$88,324	16	24
9	Mercedes-Benz	CL600 2dr	Sedan	Rear	$119,600	13	19
10	Mercedes-Benz	S500 4dr	Sedan	All	$80,939	16	24
11	Mercedes-Benz	SL500 convertible 2dr	Sports	Rear	$84,325	16	23
12	Mercedes-Benz	SL55 AMG 2dr	Sports	Rear	$113,388	14	21
13	Mercedes-Benz	SL600 convertible 2dr	Sports	Rear	$117,854	13	19
14	Porsche	911 Carrera convertible 2dr (coupe)	Sports	Rear	$69,229	18	26
15	Porsche	911 Carrera 4S coupe 2dr (convert)	Sports	All	$72,206	17	24
16	Porsche	911 Targa coupe 2dr	Sports	Rear	$67,128	18	26
17	Porsche	911 GT2 2dr	Sports	Rear	$173,560	17	24

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.

```
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.

```
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.

Output 5.8 Using the ATTRIB Statement to Reorder Variables

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

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

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

```
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.

Output 5.9 *Using the LENGTH Statement to Reorder Variables*

Obs	Height	Name	Sex	Age	Weight
1	69.0000	Alfred	M	14	112.5
2	56.5000	Alice	F	13	84.0
3	65.2969	Barbara	F	13	98.0
4	62.7969	Carol	F	14	102.5
5	63.5000	Henry	M	14	102.5
6	57.2969	James	M	12	83.0
7	59.7969	Jane	F	12	84.5
8	62.5000	Janet	F	15	112.5
9	62.5000	Jeffrey	M	13	84.0
10	59.0000	John	M	12	99.5
11	51.2969	Joyce	F	11	50.5
12	64.2969	Judy	F	14	90.0
13	56.2969	Louise	F	12	77.0
14	66.5000	Mary	F	15	112.0
15	72.0000	Philip	M	16	150.0
16	64.7969	Robert	M	12	128.0
17	67.0000	Ronald	M	15	133.0
18	57.5000	Thomas	M	11	85.0
19	66.5000	William	M	15	112.0

TIP 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
- INFORMAT Statement
- LENGTH Statement
- RETAIN Statement
- CONTENTS Procedure

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).

```
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

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

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
- RETAIN Statement
- CONTENTS Procedure

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).

```
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*

Obs	Item	Material	Investment	Profit
1	SHIRTS	COTTON	$2,256,354.00	$83,952,175.00
2	TIES	SILK	$498,678.00	$2,349,615.00
3	SUITS	SILK	$9,482,146.00	$69,839,563.00
4	BELTS	LEATHER	$7,693.00	$14,893.00
5	SHOES	LEATHER	$7,936,712.00	$22,964.00

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 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.

```
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.

Output 5.12 *PROC PRINT Result*

Obs	EmployeeID	Rate	Hours	Dept	Site	PayChk
1	16102	12	40	OS	1	480
2	12414	10	35	OS	1	350
3	10175	15	40	TS	1	600
4	10477	16	40	TS	2	640
5	10775	10	36	OS	2	360
6	10771	10	32	OS	1	320
7	10739	10	20	OS	2	200
8	17344	15	40	TS	1	600
9	16885	18	40	TS	2	720

Key Ideas

- Explicit conversions help you to control the data type and avoids 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;
```

Output 5.13 *PROC PRINT Results*

Obs	EmployeeID	Rate	Hours	Dept	Site	Loc
1	16102	12	40	OS	1	1/OS
2	12414	10	35	OS	1	1/OS
3	10175	15	40	TS	1	1/TS
4	10477	16	40	TS	2	2/TS
5	10775	10	36	OS	2	2/OS
6	10771	10	32	OS	1	1/OS
7	10739	10	20	OS	2	2/OS
8	17344	15	40	TS	1	1/TS
9	16885	18	40	TS	2	2/TS

Note: No conversion messages appear in the SAS log when you use the PUT function.

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 convertsthe 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.

```
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.

```
data test;
  input x $ y;
  put 'This is row ' _n_;
datalines;
  a 1
  b 2
  c 3
  x 24
  z 26
;
run;
proc print data=test;
run;
```

The following is printed to the SAS log:

Output 5.15 *SAS Log: _N_ Automatic Variable*

```
This is row 1
This is row 2
This is row 3
This is row 4
This is row 5
```

The following is the output from the PRINT procedure.

Output 5.16 *Automatic Variable _N_ PRINT Output*

Obs	x	y
1	a	1
2	b	2
3	c	3
4	x	24
5	z	26

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.

```
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`.

Output 5.17 *PROC PRINT Output: Temperatures*

Average Daily Low Temperature

Day1	Day2	Day3	Day4	Day5	Day6	Day7
74	75	82	84	85	86	89

Average Daily Low Temperature in Celsius

Day1	Day2	Day3	Day4	Day5	Day6	Day7
23.3333	23.8889	27.7778	28.8889	29.4444	30	31.6667

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`.

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.

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

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

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=sum(x1-x3);          /* #1 */
  T2=sum(OF x1-x3);      /* #2 */
  put T=;                /* #3 */
  put T2=;               /* #4 */
run;
```

1 The first SUM function returns `T=20`.

2 The second SUM function returns `T2=60`.

3 Writes to the log the difference between `x1` and `x3`.

4 Writes to the log the sum of the values of variables `x1`, `x2`, and `x3`.

Key Ideas

- The OF operator enables you to specify SAS variable lists or SAS arrays as arguments to functions.
- The 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 (x1 – xn) is read in as a range of values only if the list is preceded by the OF operator.
- If the same list is not preceded by the OF operator and it is used with the 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

- SUM Function
- PUT Statement
- Using the OF Operator with Temporary Arrays

Example: Use the 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 OF operator and the lists are separated by spaces or commas.

```
data _null_;
   T=sum(OF x1-x3 y1-y3 z1-z3);
run;
```

or

```
data _null_;
   T=sum(OF x1-x3, OF y1-y3, OF z1-z3);
run;
```

Key Ideas

- The OF operator enables you to specify SAS variable lists or SAS arrays as arguments to functions. 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:

```
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*

Obs	x	y
1	4	.
2	1	2
3	3	.
4	1	2

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:

```
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';
run;
```

The following output is produced:

Output 5.19 *Output with Multiple Missing Values*

Results of Test Period A

Obs	Id	Foodpr1	Foodpr2	Foodpr3	Coffeem1	Coffeem2
1	1001	115	45	65	I	78
2	1002	86	27	55	72	86
3	1004	93	52	X	76	88
4	1015	73	35	43	112	108
5	1027	101	127	39	76	79

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:

```
data test;
  x=.;
  y=5;
  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*

```
184  data test;
185    x=.;
186    y=5;
187    a=x+y;
188    b=sum(x,y);
189    c=5;
190    c+x;
191    put a= b= c=;
192   run;

a=. b=5 c=5
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 x 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 x 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*

```
The difference is -5.55112E-17
```

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.

Base 2										
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	$.2^{-1}$	2^{-2}
	128	64	32	16	8	4	2	1	1/2	1/4

Base 2										
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}

$$255.75 = (\mathbf{1} \times 2^7) + (\mathbf{1} \times 2^6) + (\mathbf{1} \times 2^5) + (\mathbf{1} \times 2^4) + (\mathbf{1} \times 2^3) + (\mathbf{1} \times 2^2) + (\mathbf{1} \times 2^1) + (\mathbf{1} \times 2^0) + (\mathbf{1} \times 2^{-1}) + (\mathbf{1} \times 2^{-2})$$

decimal point

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:

Base 16						
16^7	...	16^4	16^3	16^2	16^1	16^0
268,435,456	...	65,536	4096	256	16	1

$$1030 = (\mathbf{4} \times 16^2) + (\mathbf{0} \times 16^1) + (\mathbf{6} \times 16^0)$$
$$= 1024 + 0 + 6$$

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:

Base 16						
16^7	...	16^4	16^3	16^2	16^1	16^0
268,435,456	...	65,536	4096	256	16	1

$$200 = .200 \times 16^3$$

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:

$$200 = .200 \times 16^3$$

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 reconvert the .6 portion.

The .6 continues to repeat as 9.6, which means that you keep the 9 and reconvert. 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:

$$43 \quad hexadecimal \quad = (4 \times 16^1) + (3 \times 16^0) = 67 \quad decimal \quad = 0100\,0003 \quad binary$$

$$11000003 = 195 \quad in \quad decimal \quad = C3 \quad in \quad hexadecimal$$

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.

```
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 `.33333.` Because `1/3` is an imprecise number, the value

is not equal to .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:

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

Output 5.23 *Log Output: Using the ROUND Function to Avoid Comparison Errors*

```
MATCH
```

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:

```
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;
                -
                352
ERROR 352-185: The length of numeric variables is 3-8.

339    x=257;
340    y1=x+1;

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

Obs	x
1	3.8

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.

```
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 HEX*w.d* format to show floating-point representation.

```
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=400E666666666664
```

Key Ideas

- The HEX*w*. 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`:

```
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*

The SAS System

value	minlen
269	3
270	3
271	3
272	2

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.

```
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.

Output 5.29 *PROC PRINT Result: A Simple 1-Byte-to-1-Byte Swap*

Obs	name	encrypt	decrypt
1	ROBERT	*%14*)	ROBERT
2	JOHN	9%7$	JOHN
3	GREG	6*46	GREG

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.

```
data sample2;                               /* 1 */
 input @1 id $12.;

/*ENCRYPT*/
 encrypt=id;
   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;
 end;

/*DECRYPT*/
 decrypt=encrypt;
   j=21;
 do to_2 = "C","F","E","A","D","B";     /* 3 */
   from_2=put(j,2.);
   decrypt=tranwrd(decrypt,from_2,to_2);
    j+1;
 end;
 drop i j to_1 from_1 to_2 from_2;

datalines;
ABCDEF
FEDC
ACE
BDFA
CAFDEB
BADCF
ABC
;
proc print;
run;
```

1 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.

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. 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*

Obs	id	encrypt	decrypt
1	ABCDEF	242621252322	ABCDEF
2	FEDC	22232521	FEDC
3	ACE	242123	ACE
4	BDFA	26252224	BDFA
5	CAFDEB	212422252326	CAFDEB
6	BADCF	2624252122	BADCF
7	ABC	242621	ABC

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.

```
data
sample3;
  /* 1 */
  input num;
  array from(3) $ 10 from1-from3
('0123456789','0123456789','0123456789');    /* 2 */
  array to(3) $ 10 to1-to3 ('ABCDEFGHIJ','KLMNOPQRST','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;
  encrypt_num=cats(of new1-
new5);                                                   /* 6 */
  keep num encrypt_num;
  datalines;
  12345
  70707
  99
  1111
  ;
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;
  end;
  decrypt_num=input(cats(of new1-new5),5.);
  keep num encrypt_num decrypt_num;
run;

proc print;
run;
```

1 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.

2 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.

3 The PUT function converts the numeric value to a character value.

4 The first DO loop uses the SUBSTR function to split the value into five separate values and assigns each to the old ARRAY.

5 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

Obs	num	encrypt_num	decrypt_num
1	12345	BMXEP	12345
2	70707	HKBAR	70707
3	99	JT	99
4	1111	BLVB	1111

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

6

Data Types

Data Types in SAS 179

Data Types in SAS Viya 179
Summary of CAS LIBNAME Engine Data Types 179
VARCHAR Data Type 181
Support for Implicit Declaration of Data Types 189

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 91.

- Arrays in SAS on page 573
- Chapter 8, "SAS Constants," on page 195
- Chapter 5, "Variables," on page 75

Data Types in SAS Viya

Summary of CAS LIBNAME Engine Data Types

The CAS engine supports the storage of three data types. The following table provides information for each type:

Table 6.1 CAS LIBNAME Engine Data Types

Data Type	Description	Example	How Missing Values are Designated

CHAR $*n*	Stores a fixed-length character string. *n* is the number of bytes used to store all values. If the value stored contains fewer than *n* bytes, then SAS adds blanks (spaces) to the end of the value to fill the column.
	
	Length: 1 - 32767 bytes
		`data casuser.names;`
		`  length fname $20;`
		`  fname= "Sam";`
		`run;`
			zero or more blank spaces surrounded by single or double quotation marks:
			
			`fname="";`
			
			or
			
			`f='    ';`
			
			Note: The length of `fname="";` (defined with no blank spaces) is 1.

VARCHAR(*n*)[2]	Stores a varying-length character string. *n* is the maximum number of characters to store.
	
	Length: 0 - 536,870,911 characters (UTF-8 encoding)
		`data casuser.names;`
		` length lname varchar(40);`
		`  lname="Adams";`
		`run;`
			zero or more blank spaces surrounded by single or double quotation marks:
			
			`lname="";`
			
			`lname='    ';`
			
			Note: The length of `lname="";` (defined with no blank spaces) is 0.

DOUBLE	Stores a numeric value, including dates and times, as a floating-point number.
	
	Length: 8 bytes for the CAS Engine[1]
		`data casuser.names;`
		`  length num 8;`
		`  num = 25;`
		`run;`
			a single period
			
			`num1=.;`
			
			For special numeric missing variables on page 103, the upper or lowercase letters A-Z, or the underscore (_) character:
			
			`.`
			`.A-.Z`
			`._`

1 If a shorter length is specified, then 8 bytes are used and a note is printed to the SAS log.

2 The CAS engine is required for storing VARCHAR variables in the output table of a DATA step. The DATA step can read CAS tables containing VARCHAR variables, but it cannot store them unless a CAS engine libref is specified on the output table.

The CHAR data type does not support zero-length character strings. Strings defined with no blank spaces (double or single quotation marks with no characters or blank spaces) have a length of 1.

VARCHAR data type does support zero-length character strings. Zero-length VARCHAR variables are specified in an assignment statement and they consist of single or double quotation marks that contain no characters or blank spaces.

To determine whether a CHAR variable is missing, use the MISSING function:

```
 /*  lc=1, lvc=0 */
data _null_;
  length vc varchar(32);
  c = '';   /* no blanks */
  vc = '';
  lc = lengthc(c);
  lvc= lengthc(vc);
  put 'char length is ' lc;
  put 'varchar length is ' lvc;
  if missing(c) then put 'c is missing';
  if missing(vc) then put 'vc is missing';
run;
```

The DATA step above returns `lc=1` and `lvc=0`.

For more information about data types in CAS, see “Data Types” in *SAS Cloud Analytic Services: User’s Guide*.

The SAS V9 engine supports only the CHAR and NUMERIC types. For more information about data types that are supported by the SAS V9 Engine, see “Data Types” on page 91.

SAS Cloud Analytic Services does not support the LENGTH statement for numeric data types and does not support DOUBLE columns that use less than 8 bytes. If you use the LENGTH statement to store numeric data with less than 8 bytes and then load that data into CAS, the server stores the data with 8 bytes. If you have several numeric columns that use less than 8 bytes, the in-memory table size can be much larger than what is required for a .sas7bdat file.

VARCHAR Data Type

Definition

VARCHAR(*length* | *)

Stores a varying-length character string. *length* is the user-defined maximum length of the character string, and * indicates that the maximum storage size is 536,870,911 characters.

Syntax

```
LENGTH (variable-naame) VARCHAR(length|*);
ARRAY array-name[N] VARCHAR(length|*);
ARRAY array-name[*] VARCHAR-variables;
```

variable-name

specifies one or more variables that are assigned the type VARCHAR.

length

specifies a numeric constant that is the user-defined maximum number of characters that can be stored in the VARCHAR variable. This value can be up to 536,870,911 characters in length. Uninitialized VARCHAR variables are given a length of 1 by default. This value is based on the defined range.

```
length xyz varchar(32);
```

Uninitialized VARCHAR variables are given a length of 1 by default. 1 is the minimum length of a VARCHAR variable.

Range 1 to 536,870,911 (UTF-8) characters (2^{31} bytes)

See "PROC CONTENTS Output for VARCHAR Variables" on page 186

*

specifies that SAS uses the maximum length allowed, which is 536,870,911 characters. When assigning a character constant to a VARCHAR variable, the character constant is limited to 32,767 bytes.

```
length xyz varchar(*);
```

Uninitialized or missing VARCHAR variables are given a length of 1 by default. This value is based on the defined range for VARCHAR variables, in which 1 is the minimum length a VARCHAR variable can be.

See "PROC CONTENTS Output for VARCHAR Variables" on page 186

array-name

specifies the name of the array. Defines the elements in an array as a list of VARCHAR variables.

When using a list of VARCHAR variables with the ARRAY statement, you can use the hyphen (–), colon / prefix, and double-dash lists:

```
array arr1[*] v1-v5;
array arr2[*] v:;
array arr3[*] v1--v5;
```

You cannot use VARCHAR character lists specified as _CHARACTER_.

N

describes the number and arrangement of elements in the array

*

specifies the maximum length allowed, 536,870,911 characters. When assigning a character constant to a VARCHAR variable, the character constant is limited to 32767 bytes.

```
array myArray{*} varchar(*) a1 a2 a3 ('a','b','c');
```

Requirements requires a CAS engine libref on the output table

requires additional storage space. See VARCHAR Variable Storage. on page 188

Engine CAS engine only

Note: If you have sites that support multiple languages, consider running your SAS session using UTF-8 encoding and using the VARCHAR data type to minimize character conversion issues.

Details

The VARCHAR type is a varying length character data type whose length represents the maximum number of characters you want to store in a column. VARCHAR variables have the following characteristics:

- their length is measured in terms of characters rather than bytes
- their length varies depending on the values present

These characteristics are in contrast to those of the CHAR data type, in which length is fixed and measured in bytes.

For example, a VARCHAR(100) can store up to 100 characters, but the actual storage used in any given row depends on the lengths of the individual values in the row and column.

For example, if a VARCHAR variable, **vc2**, is defined as a VARCHAR(100) variable, this means that it can store up to 100 characters. But because the value “abc” contains only 3 characters and each character uses one byte, only 3 bytes of memory are allocated for the value in that row. A fixed-length CHAR column, on the other hand, takes up the defined number of bytes regardless of the actual value. In the case of the example, the fixed-width CHAR with a defined length of 100 would use all 100 bytes even though the actual value “abc” needs only 3 bytes.

Example: Create a VARCHAR Using the LENGTH Statement

```
libname mycas cas;
data mycas.roman;
   length vc32 varchar(32);
      do i = 1 to 10;
         vc32 = put(i, ROMAN.);
         output;
      end;
run;
```

Example: Create a VARCHAR Variable Using the ARRAY Statement

```
data mycas.test;
   array test{*} varchar(*) a1 a2 a3 ('a','b','c');
   put test[1]; put test[2]; put test[3];
```

```
run;
```

When to Use a VARCHAR Data Type

The VARCHAR data type is useful because it can save space when the lengths of the column values vary. With fixed-width data types, any space that is not used by the value in the column is padded with trailing blanks, which wastes space. The entire space is blocked out in memory whether the value needs the space or not. With varying-length data types, such as VARCHAR, only the space that is needed is used (there are no trailing blanks).

- CHAR – use a fixed-width CHAR when the sizes of the column data entries are similar. Fixed-width columns are usually accessed faster.
- VARCHAR(n) – use when the sizes of the column data vary considerably but you are reasonably certain they will not exceed a certain width.
- VARCHAR(*) – use when the sizes of the column data vary considerably and the column width might exceed any limits you might place on it.

In most cases, you should take advantage of VARCHAR support. However, if values are consistently short, such as an ID column of airport codes, then a fixed-width CHAR variable uses less memory and runs faster. This is because VARCHAR values require 16 bytes plus the memory needed to store the VARCHAR value. So, if your values are always smaller than 16 bytes, you can save memory and processing time by using a CHAR type variable instead.

Range

SAS defines the length of a VARCHAR data type in terms of characters rather than bytes. The maximum length of a VARCHAR variable is 536,870,911 Unicode characters, or, 2^{31} bytes. This means that up to 536,870,911 characters, or 2,147,483,644 bytes of data or can be stored in a VARCHAR variable. The maximum length in bytes is calculated by multiplying 536,870,911 by the maximum length that any one character in the UTF-8 character set can be, which is 4 bytes.

Setting Missing VARCHAR Variables

You can use the CALL MISSING routine to set a VARCHAR variable to missing:

```
if var1 = "abc" then call missing(var1);
```

You can use the MISSING function to test whether a VARCHAR variable is missing.

```
if missing(var2) then var2 = "missing";
```

VARCHAR Support for Implicit and Explicit Data Type Conversion

Type conversion happens when a SAS DATA step or procedure moves data from a CAS table into a SAS data set. The data must be converted from the data type supported by the CAS engine to a data type supported by the SAS V9 engine. VARCHAR type variables are not supported by the Base SAS V9 engine, so they are automatically converted from VARCHAR types to fixed-length CHAR types.

The DATA step supports the processing of VARCHAR data. However, only the CAS engine supports the VARCHAR data type. This means that the DATA step can read in and process VARCHAR data, but the data is converted to a CHAR when it is stored as a SAS data set.

When you convert between CAS data and SAS data, the supported data type conversions are defined by the engine.

If character strings declared as VARCHAR data types are converted to the CHAR data type, values that are too long for the CHAR data type are truncated.

Data types can be converted from one type to another either implicitly or explicitly.

In *implicit conversions* SAS automatically converts data from one type to another and the conversions are not visible to the user. An example is when you save a CAS table containing a VARCHAR as a SAS data set. The VARCHAR is implicitly converted to a CHAR in the output data set.

In *explicit conversions*, users deliberately convert one type to another using programming statements.

SAS language elements that can explicitly convert one data type to another are the PUT function and the INPUT function. The following converts the numeric value of a VARCHAR to a DOUBLE data type and writes the output to a CAS table.

```
data mycas.new;
   length vc varchar(40);
   vc = '5000';
   num = input(vc,8.);
run;
proc contents data=mycas.new; run;
```

Output 6.1 *PROC CONTENTS Output Showing Explicit Data Type Conversion*

Alphabetic List of Variables and Attributes					
			Len		Max Bytes Used
#	Variable	Type	Bytes	Chars	
2	num	Num	8		
1	vc	Varchar	40	40	4

PROC CONTENTS Output for VARCHAR Variables

The CONTENTS procedure provides metadata about a CAS table including detailed information about the variables in the table.

When a table contains a VARCHAR variable, PROC CONTENTS displays extra length information about the variable that is not normally displayed when a VARCHAR is not present in the table:

PROC CONTENTS Output for VARCHAR Variables

Alphabetic List of Variables and Attributes					
			Len		Max Bytes Used
#	Variable	Type	Bytes	Chars	
1	vc1	Varchar	.	.	1
2	vc2	Varchar	400	100	9
3	vc3	Varchar	40	10	3

PROC CONTENTS Output for CHAR Variables

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	ch1	Char	10
2	ch2	Char	100
3	ch3	Char	3

- **Len** displays the length specified by the user when the variable is declared. The defined length is given in 2 sub-columns, Chars and Bytes:
  - **Chars** displays the *defined* length in characters, or the length that the user specifies n the VARCHAR(*n* | *) statement when the variable is declared.
  - **Bytes** displays a length in bytes that SAS calculates based on the defined length and the SAS session encoding. SAS calculates the length by multiplying the defined length by the largest possible number of bytes required to store any one character in the character set encoding. The defined length is the value defined by the user in the VARCHAR(*n*) statement.

    For example, if your SAS session encoding is UTF-8, then the **Len Bytes** value is calculated as (*n* x 4), where *n* is the length defined in the VARCHAR(*n*) statement and 4 is the largest possible number of bytes required to store any character in the UTF-8 character set. Similarly, if the local SAS session encoding is Latin1, then the **Len Bytes** value is calculated as (*n* x 1), where *n* is the length defined in the VARCHAR(*n*) statement and 1 is the largest number of bytes required to store any character in the Latin1 character set.
- **Max Bytes Used** displays the number of bytes that are used to store the longest string value in the column.

```
data mycas.new;
  length vc1 varchar(*)   /* defined length of vc1, shown in "Len
Chars" column */
         vc2 varchar(100) /* defined length of vc2, shown in "Len
Chars" column */
         vc3 varchar(10); /* defined length of vc3, shown in "Len
Chars" column */
  vc2 = "123456789";   /* actual bytes used for vc2, shown in Max
Bytes Used column */
  vc3 = "abc";         /* actual bytes used for vc3, shown in Max
Bytes Used column */
run;
proc contents data=mycas.new; run;
```

defined length / actual length

#	Variable	Type	Len		Max Bytes Used
			Bytes	Chars	
1	vc1	Varchar	.	.	1
2	vc2	Varchar	400	100	9
3	vc3	Varchar	40	10	3

In a UTF-8 session, the length in Bytes is 4 times the length in Chars.

If you specify VARCHAR(*), then SAS defines the length as the maximum allowable length.

VARCHAR Length with Implicit Type Conversion

VARCHAR variables are not supported by the SAS V9 engine. SAS automatically converts VARCHAR variables to CHAR variables if you try to store them in anything other than a CAS table.

When converting a variable from a VARCHAR to a CHAR, the length of the CHAR depends on how the VARCHAR is originally defined.

- VARCHAR(*) – If a table that contains a VARCHAR(*) definition is saved as a SAS data set, the VARCHAR is automatically converted to a CHAR that has a length equal to the Max Bytes Used for the original VARCHAR.
- VARCHAR(*n*) – If a table that contains a VARCHAR(*n*) definition is converted to a SAS data set, then the length of the variable depends on the local SAS session encoding. The length is calculated as follows: SAS multiplies the current length of the VARCHAR by the maximum value that a character's length can be in the local SAS session encoding.
  - If the local SAS session encoding uses single-byte characters, then the VARCHAR is converted to a CHAR with a length of (*n* x 1). *n* is the length of the original VARCHAR and 1 is the largest number of bytes required to store any character in the character set.
  - If the local SAS session encoding uses double-byte characters, then the VARCHAR is converted to a CHAR with length (*n* x 2). *n* is the length of the original VARCHAR and 2 is the largest number of bytes required to store any character in the character set.
  - If the local SAS session encoding uses UTF-8 encoding, then the VARCHAR is converted to a CHAR with length (*n* x 4). *n* is the length of the original VARCHAR and 4 is the largest number of bytes required to store any character in the character set.

Note: When assigning a character constant to a VARCHAR variable, the character constant is limited to 32767 bytes.

VARCHAR Variable Storage

When CHAR values are stored, they are right-padded with spaces to the specified length. VARCHAR values are not padded. When VARCHAR values are stored, they are stored with descriptor information in each row that takes space along with the data value. The descriptor is a 16-byte prefix that is stored with the data value in each row and that contains information about the length of the data value.

Restrictions for the VARCHAR Data Type in the CAS Engine

Not all SAS language elements support the VARCHAR data type, even with the CAS engine. There are also differences in how some SAS language elements behave with VARCHAR variables. These limitations and behavior differences are listed in the table below.

Table 6.2 Restrictions and Notable Behaviors for the VARCHAR Data Type in the CAS Engine

Feature	Description
ATTRIB	You cannot use the ATTRIB statement to create VARCHAR variables.
BY statement	The BY statement uses a fixed width for VARCHAR variables. Using a VARCHAR(*) type in the BY statement might cause unexpected results.
Formats	The width of VARCHAR formats is measured in bytes rather than characters.
Functions	When passing a character value to a function, numbers indicating a length or position that are passed to the function or returned by the function are in units of bytes. When passing a VARCHAR variable to functions, these numbers are in units of characters. Of note, this includes the INDEX and SUBSTR functions. See “Index CHAR and VARCHAR Character Strings” in *SAS Cloud Analytic Services: DATA Step Programming* for a related example.

KEY= on SET and MODIFY statements	VARCHAR variables are not supported by the KEY= option in either the SET or MODIFY statements.
PUT statement (to ODS output)	VARCHAR variables are not supported with the PUT statement when the DATA step writes output using ODS.
VARCHAR	The VARCHAR data type is supported by the CAS engine but not by the V9 engine. This means that to create or store a VARCHAR variable, you must use the CAS engine. The SAS DATA step (with the V9 engine) can read data containing VARCHAR variables but it converts and stores them as CHAR data types.
Variable Lists	Selecting a character variable range for character variable lists (for example, *a-character-f*) is not supported for VARCHAR variables because VARCHAR variables are not fixed-width character variables. For example, you cannot specify VARCHAR variables using the following shorthand forms: _CHARACTER_ *var1*–CHARACTER-*varN*

Concatenating variables	When concatenating character values, the result is a character value that is limited to 32767 bytes. In this example, the character result of the concatenation is assigned to a VARCHAR variable and the result value is limited to 32767 bytes.
	`length vc varchar(*);`
	`vc = "a string that is 32000 bytes long"`
	`     || "another string that is 32000 bytes long";`
	To go beyond the 32767 byte limit, include a VARCHAR variable in the concatenation. The concatenation result is a VARCHAR that can go beyond 32767 bytes.
	`length vc1 vc2 varchar(*);`
	`vc1 = "a string that is 32000 bytes long";`
	`vc2 = vc1 || "another string that is 32000 bytes long";`

Support for Implicit Declaration of Data Types

The CAS engine supports implicit data type declaration for DOUBLE (NUMERIC) and CHAR type variables.

Implicit declaration means that you do not have to explicitly declare a variable's type or length before using it. You can create a new variable and use it for the first time in an assignment statement without having to explicitly declare its type or

length. When you create a variable in this way, SAS determines the type based on the values that you assign to the variable.

- Variables that are assigned a character string value are implicitly defined as a CHAR types with a default length of 8 bytes.
- Variables that are assigned an integer value are implicitly defined as DOUBLE types with a default length of 8 bytes.

  Note: This is different from the V9 engine, which supports a length range of 1 - 8 bytes for NUMERIC types.

- Implicit type declaration is not supported for VARCHAR variables. VARCHAR variables must be explicitly declared in either a LENGTH statement or an ARRAY statement.

In the following DATA step, the type and length for variables **x** and **y** are set implicitly:

```
libname mycas cas;
data mycas.datatypes;
   x=1;
   y='hello';
run;

proc contents data mycas.datatypes;
run;
```

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
1	x	Num	8
2	y	Char	5

For information about data types supported by the SAS V9 engine, see “Data Types” on page 91.

7

SAS Expressions

SAS Expressions 191
Definitions 191
Use of SAS Expressions 192
SAS Constants in Expressions 192
SAS Variables in Expressions 193
SAS Functions in Expressions 193
SAS Operators in Expressions 193

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 195.

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 77.

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 215.

8

SAS Constants

SAS Constants in Expressions 195
Definitions 195
Character Constants 196
Numeric Constants 199
Date, Time, and Datetime Constants 200
Bit Testing Constants 201

Examples: Expressions and Constants 203
Example: Use Character Constants in Expressions 203
Example: Compare Character Constants with Character Variables 204
Example: Use Quotation Marks Within Strings 206
Example: Define Character Constants in Hexadecimal Notation 207
Example: Define Numeric Constants in Standard Notation 208
Example: Define Numeric Constants in Scientific Notation 209
Example: Define Numeric Constants in Hexadecimal Notation 210
Example: Define Date, Time, and Datetime Values in Date Constants 211
Example: Bit Test a Variable’s Value 212
Example: Avoiding a Common Error with Constants 213

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
- date, time, and datetime constants
- bit testing 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 207.

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

Character	Possible Interpretation	Example
b	bit testing constant	'00100000'b
d	date constant	'01jan04'd
dt	datetime constant	'18jan2005:9:27:05am'dt
n	name literal	'My Table'n
t	time constant	'9:25:19pm't
x	hexadecimal notation	'534153'x

See Also

Examples

- "Example: Use Character Constants in Expressions" on page 203
- "Example: Compare Character Constants with Character Variables"
- "Example: Use Quotation Marks Within Strings"
- "Example: Define Character Constants in Hexadecimal Notation"

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*

Numeric Constant	Description
1	is an unsigned integer
-5	contains a minus sign
+49	contains a plus sign
1.23	contains decimal places
01	contains a leading zero, which is not significant

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 2x10^4 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

Examples

- "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. For an example of how to use D, T, and DT to create to create date, time, and datetime constants, see "Example: Define Date, Time, and Datetime Values in Date Constants".

Constant	Format	Examples
Date	'*ddmmm<yy>yy*'D "*ddmmm<yy>yy*"D	`'1jan2018'D` `"01jan18"D`

Constant	Format	Examples
Time	*'hh:mm<:ss.s>'*T *"hh:mm<:ss.s>"*T	`'9:25'T` `"9:25:19pm"T`
Datetime	*'ddmmm<yy>yy:hh:mm<:ss.s>'*DT *"ddmmm<yy>yy:hh:mm<:ss.s>"*DT	`'01may18:9:30:00'DT` `"18jan2018:9:27:05am"DT`
Datetime with time zone designators (ISO 8601 standard)	*'yyyy-mm-dd*T*hh:mm:ss*Z'DT *'yyyy-mm-dd*T*hh:mm:ss+|-hh:ss'*DT	`'2018-07-20T12:00:00Z'DT` `'2018-05-17T09:15:30-05:00'DT`

IMPORTANT UTC or ISO 8601 Datetime constants, which have the Zulu timezone indication or a numeric offset from the Universal Coordinate Time, are converted to local time by adjusting the internal value according to the system timezone offset. This adjustment occurs regardless of whether the system TIMEZONE option has been explicitly set. If you have specified the TIMEZONE= system option, then SAS converts UTC and ISO 8601 DATETIME constants based on the value that you specify in the TIMEZONE= system option. Otherwise, if you do not specify the TIMEZONE= system option, then SAS converts UTC and ISO 8601 DATETIME constants based on the system UTC time zone offset.

Note: This information is also included in "Common Functions" in *SAS Cloud Analytic Services: CASL Reference*.

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

"Example: Define Date, Time, and Datetime Values in Date Constants" on page 211

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.

  When SAS tests a character value, it aligns the left-most bit of the mask with the left-most bit of the string; the test proceeds through the corresponding bits, moving to the right.

  When SAS tests a numeric value, the value is truncated from a floating-point number to a 32-bit integer. The right-most bit of the mask is aligned with the right-most bit of the number, and the test proceeds through the corresponding bits, moving to the left.

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:

```
x='0101'b;     /* incorrect*/
```

TIP The $BINARY*w*. and BINARY*w*. formats and the $BINARY*w*., BINARY*w.d*, and BITS*w.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:

```
data example;
   char_const = 'Tom';          /* 1 */
   apostrophe = "Tom's";        /* 2 */
   singlequote = 'Tom''s';      /* 3 */
run;
proc print data=example;
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*

Obs	char_const	apostrophe	singlequote
1	Tom	Tom's	Tom's

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*

Obs	var1	var2	name1	name2
1	abc	def	abc	def

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 Character Constants in Expressions”

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*

Obs	book1	book2	book3	book4
1	Uncle Tom's Cabin	Uncle Tom's Cabin	"Ben Hur"	"Ben Hur"

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 Code

The following example illustrates character constants in hexadecimal notation:

```
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.

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:

```
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:

```
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*

```
hex1=193
hex2=9
```

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*

Obs	date	time	datetime
1	01JAN18	21:25:19	18JAN18:09:27:05

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

- "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.

```
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*

```
For value 8 bit 4 is 1
For value 7 bit 4 is 0
```

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'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

9

Operators

SAS Operators ... **215**
Definitions for SAS Operators ... 215
Arithmetic Operators ... 216
Comparison Operators ... 217
Logical Operators ... 221
MIN and MAX Operators ... 223
Concatenation Operators ... 224
Order of Operation in Compound Expressions ... 225
Short-Circuit Evaluation in SAS ... 227

Summary of Ways to Use Operators ... **229**
Summary of Ways to Use Operators Tables ... 229

Examples: Operators ... **231**
Example: Subset Data Using a Comparison Operator ... 231
Example: Create Variables Using Comparison Operators ... 232
Example: Search an Array of Numeric Variable Values Using the IN Operator ... 233
Example: Search an Array of Character Variable Values Using the IN Operator ... 235
Example: Compare a Specified Prefix of a Character Expression ... 236
Example: Compare Variables Using Boolean Operators ... 237
Example: Use the NOT Operator to Reverse the Logic of a Comparison ... 238
Example: Remove Trailing Blanks Using the TRIM Function in a Concatenation Operation ... 240

SAS Operators

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:

```
+y
```

- Constants:

```
-25
```

- Functions:

```
-cos(angle1)
```

- Parenthetical expressions:

```
+(x*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

Symbol	Description	Example
**	Raise A to the third power.	`a**3`

Symbol	Description	Example
*[1]	Multiply 2 by the value of Y.	`2*y`
/	Divide the value of VAR by 5.	`var/5`
+	Add 3 to the value of NUM.	`num+3`
-	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 225 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 102 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*

Symbol/Mnemonic	Definition	Example
= or EQ	equal to	`a=3`
^=, ¬=, ~=, or NE [1]	not equal to	`a ne 3`

Symbol/Mnemonic	Definition	Example
> or GT	greater than	`num>5`
< or LT	less than	`num<8`
>= or GE[2]	greater than or equal to	`sales>=300`
<= or LE[3]	less than or equal to	`sales<=100`
or IN	equal to one of a list	`num in (3, 4, 5)`

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 102 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 236.
- 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*

	Use Cases	Examples
Numeric Variables	Specify a range of sequential integers to search. You can also use multiple ranges in the same IN list.	■ `y=x in(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);` ■ `y=x in(1 2 3 4 5 6 7 8 9 10);` ■ `y=x in(1:10);`[1]
	Use both ranges and constants in the list of values to compare.	`if x in (0,9,1:5);`
	Search an array of numeric values.	"Example: Search an Array of Numeric Variable Values Using the IN Operator"
Character Variables	Determine whether a variable's value is among a list of character values.[2]	■ `if state in ('NY','NJ','PA') then region+1;` ■ `if state in ('NY' 'NJ' 'PA') then region+1;` ■ `if state='NY' or state='NJ' or state='PA' then region+1;` [1]

	Use Cases	Examples
	Search an array of character values.	"Example: Search an Array of Character Variable Values Using the IN Operator"

1 These statements produce the same results.
2 The IN operator accepts character variable values of different lengths.

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

Symbol/Mnemonic	Description	Example
& or AND	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.	`(a>b & c>d)`
| or OR[1]	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.	`(a>b or c>d)`
! or OR		
¦ or OR		
¬ or NOT[2]		`not(a>b)`
∘ or NOT		

Symbol/Mnemonic	Description	Example
~ or NOT		

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 238.

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

Symbol/Mnemonic	Description	Examples
>< 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)`

Symbol/Mnemonic	Description	Examples
		■ `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 103.

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 <inlineCode>level='grade '||'A'</inlineCode>. The length of the resulting variable is the sum of the lengths of each variable or constant in the concatenation operation. You can use a LENGTH or an ATTRIB statement to specify a different length for the new variable.

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>
<tr><th>Use Case</th><th>Example</th></tr>
<tr><td>Concatenate the value of a variable with a character constant.</td><td><code>newname='Mr. or Ms. ' ||oldname;</code> If the value of OldName is <code>'Jones'</code>, then NewName has the value <code>'Mr. or Ms. Jones'</code></td></tr>
</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 240

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. Fore more information, see "Short-Circuit Evaluation in SAS".

Table 9.7 Order of Operation in Compound Expressions

Priority	Order of Evaluation	Symbol/ Mnemonic	Action	Example
Group 1	right to left	**	exponentiation	■ `y=a**2;`

Priority	Order of Evaluation	Symbol/ Mnemonic	Action	Example
				■ `x=2**3**4` is evaluated as `x=(2**(3**4))`
		+	positive prefix [1]	`y=+(a*b);`
		-	negative prefix[2]	`z=-(a+b);`
		∘ ¬ ~ or NOT	logical not[3]	`if not z then put x;`
		>< or MIN	minimum	■ `x=(a><b);` ■ `-3><-3` is evaluated as `-(3><-3)`. These are equal to -(-3), which equals +3.
		<> or MAX	maximum	`x=(a<>b);`
Group 2	left to right	*	multiplication	`c=a*b;`
		/	division	`f=g/h;`
Group 3	left to right	+	addition	`c=a+b;`
		-	subtraction	`f=g-h;`
Group 4	left to right	|| ¦¦ !!	concatenate character values[4]	`name= 'J'||'SMITH';`
Group 5[5]	left to right[6]	< or LT	less than	`if x<y then c=5;`
		<= or LE	less than or equal to	`if x le y then a=0;`
		= or EQ	equal to	`if y eq (x+a) then output;`
		¬= or NE	not equal to	`if x ne z then output;`
		>= or GE	greater than or equal to	`if y>=a then output;`
		> or GT	greater than	`if z>a then output;`
		IN	equal to one of a list	`if state in ('NY','NJ','PA') then region='NE';` `y = x in (1:10);`

Priority	Order of Evaluation	Symbol/ Mnemonic	Action	Example
Group 6	left to right	& or AND	logical and	`if a=b & c=d then x=1;`
Group 7	left to right	| ¦ ! or OR	logical or[7]	`if y=2 or x=3 then a=d;`

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 102 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 AND Y)`, if the first argument `(X)` evaluates to FALSE, then the overall expression `(X 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.

```
data test;
   a=0;
   if (a>0 AND a=1/a) then put 'hello';
   else put 'goodbye';
run;
```

```
goodbye
NOTE: The data set WORK.TEST has 1 observations and 1 variables.
```

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.

```
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.

```
data test;
   a=0;
   IF a>0 THEN
      DO;
         IF a=1/a THEN put 'hello';
            ELSE put 'goodbye';
      END;
      ELSE
         put 'goodbye again';
run;
```

```
goodbye again
NOTE: The data set WORK.TEST has 1 observations and 1 variables.
```

Summary of Ways to Use Operators

Summary of Ways to Use Operators Tables

Table 9.8 Summary of Ways to Use SAS Operators

Symbol	Description	Example
Arithmetic Operators		
**	Raise A to the third power.	a**3
*	Multiply 2 by the value of Y.	2*y
/	Divide the value of VAR by 5.	var/5
+	Add 3 to the value of NUM.	num+3
-	Subtract the value of DISCOUNT from the value of SALE.	sale-discount
Comparison Operators		
= or EQ	equal to	■ a=3 ■ 'Jones' EQ 'Jones'
^=, ¬=, ~=, or NE	not equal to	■ a ne 3 ■ 'Jones' NE 'JONES'
> or GT	greater than	■ num>5 ■ 'CharlesJones'>'C.Jones'
< or LT	less than	num<8
>= or GE	greater than or equal to	sales>=300
<= or LE	less than or equal to	sales<=100
IN	equal to one of a list	■ num in (3, 4, 5) ■ state in ('NY','NJ','PA')

Symbol	Description	Example
Logical Operators		
& or AND	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.	(a>b & c>d)
| or OR	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.	(a>b or c>d)
! or OR		
¦ or OR		
¬ or NOT		not(a>b)
∘ or NOT		
~ or NOT		
MIN and MAX Operators		
>< or MIN	Returns the lower of the two values.	■ where x = (b min c); ■ if x = (y><z);
<> or MAX	Returns the higher of the two values.	■ `where a=(b max c)` ■ `if x = (a<>b)`
Concatenation Operator		
||	Combines character values.	Address= StreetNum || CityStateZip

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.

```
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.

Obs	Make	Model	MPG_City	MPG_Highway	MPG_AVG	Green_Rating
1	Acura	RSX Type S 2dr	24	31	27.5	2
2	Acura	TSX 4dr	22	29	25.5	2
3	Audi	A4 1.8T 4dr	22	31	26.5	2
4	Audi	A41.8T convertible 2dr	23	30	26.5	2
5	Audi	TT 3.2 coupe 2dr (convertible)	21	29	25.0	2
6	BMW	330i 4dr	20	30	25.0	2
7	BMW	330Ci 2dr	20	30	25.0	2
8	BMW	530i 4dr	20	30	25.0	2
9	BMW	Z4 convertible 3.0i 2dr	21	29	25.0	2
10	Buick	Century Custom 4dr	20	30	25.0	2
11	Buick	Regal LS 4dr	20	30	25.0	2
12	Chevrolet	Aveo 4dr	28	34	31.0	1
13	Chevrolet	Aveo LS 4dr hatch	28	34	31.0	1
14	Chevrolet	Cavalier 2dr	26	37	31.5	1
15	Chevrolet	Cavalier 4dr	26	37	31.5	1

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 229
- You can use comparisons in expressions in Assignment statements. For example, see “Example: Create Variables Using Comparison Operators” on page 232.
- Missing numeric values have their own sort order. For more information, see “Missing Variable Values” on page 102.

Example: Create Variables Using Comparison Operators

Example Code

The following example shows how to use comparisons in the Assignment Statement to create a variable.

```
data test;          /* 1 */
   x=6;
   y=8;
   c=5*(x<y)+12*(x>=y);    /* 2 */
   put c;                  /* 3 */
run;
```

1 Assign values to the variables x and y so that x equals 6 and y equals 8.

2 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.

3 Print the value for c in the log.

```
314 data test;
315   x=6;
316   y=8;
317   c=5*(x<y)+12*(x>=y);
318   put c;
319 run;
5
```

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 229

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;
proc print data=test;run;
```

1 Create an array named **a** that contains the following 10 values: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5.

2 Create a variable, x, and assign it the value 99.

3 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.

Obs	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	x	y
1	1	2	3	4	5	1	2	3	4	5	99	0

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`.

```
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;
```

1 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.

Obs	a1	a2	a3	a4	a5	a6	a7	a8	a9	a10	x	y
1	1	2	3	4	99	1	2	3	4	5	99	1

```
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;
y=1
```

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 219

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.

```
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)*

```
190  data _null_;
191     array a{5} $ (5*'');
192     x='b1';
193     y = x in a;
194     put y=;
195     a{5} = 'b1';
196     y = x in a;
197     put y=;
198  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 220

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:

Obs	Action	Name	Eval_1	Eval_2	Eval_3	Status
1	Print Card	Lilac and Lavender	97	95	99	Passed
2	Contact	The Salty Pearl	85	70	65	F
3	Print Card	Taste the Range	90	92	90	Pass
4	Print Card	The Underground	85	90	90	P
5	Contact	When Pigs Fly	70	70	67	Fail
6	Print Card	Basil	85	90	90	Passed

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.

See Also

- *SAS Functions and CALL Routines: Reference*
- “Character Comparisons” on page 219

Example: Compare Variables Using Boolean Operators

Example Code

This example shows how you can use Boolean operators to compare variables.

```
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;
```

Obs	Restock	ItemNum	Quantity	PriorityLev
1	1	6530	10	8
2	1	8759	3	1
3	0	4573	22	10
4	1	9237	2	10
5	0	4329	12	4
6	1	9831	9	7
7	0	9830	15	4
8	0	9458	25	1
9	1	5673	4	10
10	1	7562	7	3
11	1	3291	3	10

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

- "Boolean Numeric Expressions" in *SAS Language Reference: Concepts*.

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.

```
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`.

Obs	ProdID	Avg	Category
1	8954	3.8	Pass
2	9183	5.0	Pass
3	6839	1.0	Fail
4	3493	2.0	Fail
5	2908	2.8	Fail
6	5419	4.8	Pass
7	3759	3.3	Pass
8	5301	3.8	Pass

Key Ideas

- You can use the NOT operator with other operators to reverse the logic of the comparison.

See Also

- "NOT Operator" on page 222

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*

10
Dates and Times

Dates, Times, and Intervals 241

Dates, Times, and Intervals

SAS provides date, time, and datetime intervals for counting different periods of elapsed time. See the following resources for more information:

- "Date, Time, and Datetime Constants" on page 200
- "Example: Define Date, Time, and Datetime Values in Date Constants"
- "About Date and Time Intervals" in *SAS Formats and Informats: Reference*

11

Component Objects

DATA Step Component Objects 243

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*

PART 3

Accessing Data

Chapter 12
SAS Libraries 247

Chapter 13
SAS Engines 289

Chapter 14
SAS Data Sets 321

Chapter 15
Raw Data 353

Chapter 16
Database and PC Files 385

Chapter 17
SAS Views 393

Chapter 18
SAS Dictionary Tables 395

12

SAS Libraries

Definitions for SAS Libraries 248

Elements of a Library Assignment 250
Introduction to Assigning Libraries 250
Library Name (Libref) 251
Library Engine 252
Library Location 252
Library Options 253
Benefits of Using a Libref 253
Accessing Data without Using a Libref 254

Library Concatenation 255
Definition of SAS Library Concatenation 255
How to Concatenate Libraries 255
Rules for Library Concatenation 255

SAS Default Libraries 256
Introduction to SAS Default Libraries 256
Work Library (Temporary) 257
User Library 257
Sashelp Library 258
Sasuser Library 258

Examples: Access Data by Using a Libref 259
Example: Assign a Libref by Using the LIBNAME Statement 259
Example: Assign a Libref by Using a Function 260
Example: Concatenate SAS Libraries 262
Example: Access a Remote SAS Library by Using SAS/CONNECT Software 263
Example: Access a Remote SAS Library on a WebDAV Server 265
Example: Access DBMS Data as a SAS Library 266
Example: Access DBMS Data to Create a SAS View 267
Example: Automatically Create a New Subdirectory for a Library 269
Example: Deassign (Clear) a Libref 270

Examples: Access Data without Using a Libref 272
Example: Use the Work Library for Temporary Data 272
Example: Assign the User Library for Permanent Data 273
Example: Access a Data Set by Specifying a Path Name in Quotation Marks 275
Example: Specify a Fileref for a File That Is Not a Library Member 276

Examples: View Information about a Library 278
Example: Print Information about a Library and Its Members 278

Example: Return Library Attributes by Using the LIBNAME Statement 280
Example: Return Library Location by Using a Function 281

Examples: Manage SAS Libraries **282**
Example: Copy a SAS Library 282
Example: Migrate a SAS Library 283
Example: Copy a SAS Library by Using a Transport File 285

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*

File	Member Type	Examples
data set or table	DATA	■ "Example: Create and Print a V9 Engine Data Set" in *SAS V9 LIBNAME Engine: Reference* ■ "Example: Access DBMS Data as a SAS Library" ■ "Example: Read and Write SAS Data in Hadoop by Using the SPD Engine" ■ "Using an XMLMap to Import an XML Document as One SAS Data Set" in *SAS XMLV2 and XML LIBNAME Engines: User's Guide*
view	VIEW	■ "Examples: SAS Views" in *SAS V9 LIBNAME Engine: Reference* ■ "Example: Access DBMS Data to Create a SAS View"
catalog	CATALOG	■ "Examples: Catalogs" in *SAS V9 LIBNAME Engine: Reference*

File	Member Type	Examples
stored compiled DATA step program	PROGRAM	■ "Examples: Stored, Compiled DATA Step Programs" in *SAS V9 LIBNAME Engine: Reference*
item store	ITEMSTOR	■ "Listing Templates in a Template Store" in *SAS Output Delivery System: Procedures Guide* ■ "Where the SAS Registry Is Stored" ■ For information about linear model item stores, see the PLM procedure in *SAS/STAT User's Guide*

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*

SAS File	Member Type
audit trail	AUDIT
extended attributes	XATTR
index	INDEX

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:

- "File Extensions and Member Types in UNIX Environments" in *SAS Companion for UNIX Environments*
- "File Extensions for SAS Files" in *SAS Companion for Windows*
- "UFS Libraries" in *SAS Companion for z/OS*

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

Method	Persistence Beyond the Current Session
LIBNAME statement (example) LIBNAME function (example)	Place the LIBNAME statement in the SAS Registry, an autoexec file, or the configuration file.
New Library window	Check the **Enable at start-up** box in the SAS windowing environment or the **Re-create this library at start-up** box in SAS Studio.
Administrative interface such as SAS Management Console	A SAS administrator pre-assigns and persists the library.
Operating environment commands, such as JCL statements on z/OS	See information about deallocation in “Assigning SAS Libraries Externally” in *SAS Companion for z/OS*.
Environment variable	You can store the location of a library in an environment variable and then use the variable name in place of a libref. However, you cannot include an engine name or library options. The default engine is used.

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 44.
- Do not use the reserved names Sashelp, Sasuser, or Work as librefs, except as intended. (See “SAS Default Libraries” on page 256.) 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.

- 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 289. See also "Example: Set a Default Engine" on page 307.

Library Location

The physical location is required in a library assignment:

```
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:

```
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:

- *SAS V9 LIBNAME Engine: Reference*
- *SAS/ACCESS for Relational Databases: Reference*
- *SAS/ACCESS Interface to PC Files: Reference*
- *SAS Scalable Performance Data Engine: Reference*
- *SAS XMLV2 and XML LIBNAME Engines: User's Guide*

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 262.

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 272.

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 273. 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 761.)

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 761.)

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.

```
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.

3 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*

Operating Environment	Examples
DOS, Windows	`libname mylibref v9 'c:\myfiles\data;`

Operating Environment	Examples
UNIX	`libname mylibref v9 '/myfiles/data;`
z/OS	`libname mylibref v9 'userid.myfiles.data;` `libname mylibref v9 '/myfiles/data;`

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 250
- "Example: Assign the User Library for Permanent Data" on page 273
- "Working with Libraries" in SAS Studio: User's Guide
- "Creating a SAS Library"in SAS Enterprise Guide: User's Guide

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.

```
%macro test;
   %let mylibref=new;                                    /* 1 */
   %let mydirectory=c:\example;                          /* 2 */
   %if %sysfunc(libname(&mylibref,&mydirectory)) %then /* 3 */
      %put %sysfunc(sysmsg());
   %else %put success;
   %if %sysfunc(libref(&mylibref)) %then                 /* 4 */
```

```
      %put %sysfunc(sysmsg());
   %else %put library &mylibref is assigned to &mydirectory;
%mend test;

%test
```

1 The macro variable `mylibref` specifies the libref name, `new`.

2 The macro variable `mydirectory` specifies the library's physical location, `c:\example`.

3 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.

4 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*
- "Elements of a Library Assignment" on page 250

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*

lib1

Name	Member Type
APPLES	DATA
FORMATS	CATALOG
PEARS	DATA

+

lib2

Name	Member Type
APPLES	DATA
APPLES	INDEX
FORMATS	CATALOG
FRUIT	CATALOG
ORANGES	DATA
ORANGES	INDEX

=

lib3

Name	Member Type
APPLES	DATA
FORMATS	CATALOG
FRUIT	CATALOG
ORANGES	DATA
ORANGES	INDEX
PEARS	DATA

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 255
- "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.

```
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.

Output 12.2 Portion of PROC DATASETS Output Showing the Remote Directory Information

Directory	
Libref	REPORTS
Engine	REMOTE
Physical Name	/u/mysasdata
Accessed through server	MYSERVER.MYPORT
Server's libref	REPORTS
Server's engine	V9
Server's SAS release	9.04.01M4P11092016 (150)
Server's host type	z/OS - IBM 2964
Server's versus user's data representation	DIFFERENT
Views interpreted in server's execution	YES
Owner	USERID
Group	.
File Size	4096
Device number (dev)	773
Dir serial number (ino)	83863546

Key Ideas

- 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.

See Also

- "Types of Sign-ons" in *SAS/CONNECT User's Guide*
- "COMAMID=" 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:

```
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.

```
libname mytddata teradata server=mytera user=myid password=mypw; /* 1 */
data mytddata.grades;                                             /* 2 */
   input 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*

Directory	
Libref	MYTDDATA
Engine	TERADATA
Physical Name	mytera
Schema/User	myid

Output 12.4 PROC DATASETS Output Showing the Library Contents for DBMS Content

#	Name	Member Type	DBMS Member Type
1	grades	DATA	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 289

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 266. This code creates a SAS view, not a native Teradata view.

```
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 266.

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

Obs	Student	test1	test2	final
1	Wilma	97	91	98

If you run PROC DATASETS, you can see that the member type of `highgrades` is VIEW.

```
proc datasets library=target;
run;
quit;
```

Output 12.6 PROC DATASETS Output for the Target Library

#	Name	Member Type	File Size	Last Modified
1	HIGHGRADES	VIEW	5KB	03/19/2019 12:47:03

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.

```
proc datasets library=mytddata;
run;
quit;
```

Output 12.7 *PROC DATASETS Output for the Mytddata Library*

#	Name	Member Type	DBMS Member Type
1	grades	DATA	TABLE
2	gradessql	DATA	TABLE
3	gradessqlview	DATA	VIEW

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 289

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`.

```
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 260:

```
%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`.

```
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*

Data Set Name	WORK.MYTABLE	**Observations**	1
Member Type	DATA	**Variables**	1
Engine	V9	**Indexes**	0
Created	01/30/2019 14:25:51	**Observation Length**	8
Last Modified	01/30/2019 14:25:51	**Deleted Observations**	0
Protection		**Compressed**	NO
Data Set Type		**Sorted**	NO
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

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 257
- "Example: Assign the User Library for Permanent Data" on page 273
- "Example: Access a Data Set by Specifying a Path Name in Quotation Marks" on page 275

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 259.

```
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
5    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:

```
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 257
- “USER= System Option” in *SAS System Options: Reference*
- “Example: Use the Work Library for Temporary Data” on page 272
- “Example: Access a Data Set by Specifying a Path Name in Quotation Marks” on page 275

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 259. A Windows path name is used for this demonstration.

```
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 254.

See Also

- “LIBNAME Statement: V9 Engine” in *SAS V9 LIBNAME Engine: Reference*
- “Elements of a Library Assignment” on page 250
- “Example: Assign the User Library for Permanent Data” on page 273

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.

```
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*

Obs	Account	Name	Type	Transaction
1	1118	ART CONTUCK	D	57.69
2	2287	MICHAEL WINSTONE	D	145.89
3	6201	MARY WATERS	C	45.00
4	7821	MICHELLE STANTON	A	304.45
5	6621	WALTER LUND	C	234.76
6	1086	KATHERINE MORRY	A	64.98
7	0556	LEE McDONALD	D	70.82
8	7821	ELIZABETH WESTIN	C	188.23
9	0265	JEFFREY DONALDSON	C	78.90
10	1010	MARTIN LYNN	D	150.55

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 353
- “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*

Directory	
Libref	MYFILES
Engine	V9
Physical Name	c:\example
Filename	c:\example
Owner Name	userid
File Size	8KB
File Size (bytes)	8192

Output 12.11 *PROC DATASETS Output Showing the Library Members*

#	Name	Member Type	File Size	Last Modified
1	CONTAINERS	VIEW	5KB	07/19/2018 08:33:19
2	FLOWERS	DATA	192KB	01/18/2019 17:24:24
	FLOWERS	INDEX	9KB	01/18/2019 17:24:24
3	FORMATS	CATALOG	17KB	07/03/2018 14:06:19
4	RESTOCK	DATA	128KB	11/09/2018 11:50:50

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
      File Size=              8KB
      File Size (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 250
- "The SAS Log" on page 677

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`.

```
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*
- "ATTRN Function" in *SAS Functions and CALL Routines: Reference*
- "EXIST Function" in *SAS Functions and CALL Routines: Reference*
- "PATHNAME 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.

```
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 737.

- 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 CPORT 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 CPORT 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 CPORT begins to transport data set SOURCE.RESTOCK
NOTE: The data set contains 4 variables and 7 observations.
      Logical record length is 40.
NOTE: PROCEDURE CPORT 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*

13

SAS Engines

Definitions for SAS Engines **289**

How Engines Work with Files **290**

Engine Characteristics **292**
- Summary of SAS Engines 292
- Default Base SAS Engine (V9 Engine) 294
- Legacy Engines 295
- Access Patterns 297
- Levels of Locking 297
- Operating Environment Information 298

Examples: Use a SAS Engine to Process SAS Data **298**
- Example: Assign the V9 Engine in a LIBNAME Statement 298
- Example: Assign the SPD Engine in a LIBNAME Statement 300
- Example: Read and Write SAS Data in Hadoop by Using the SPD Engine 301
- Example: Avoid Truncation by Using the CVP Engine with the V9 Engine 302
- Example: Load a SAS Data Set to a CAS Server 305
- Example: Set a Default Engine 307

Examples: Use a SAS Engine to Process External Data **308**
- Example: Read a Comma-Delimited File 308
- Example: Read Microsoft Excel Data by Using a SAS/ACCESS Engine and PROC IMPORT 310
- Example: Create Data in a DBMS by Using a SAS/ACCESS Engine 311
- Example: Embed a SAS/ACCESS LIBNAME Statement in a PROC SQL View 313
- Example: Access DBMS Data by Using the SQL Pass-Through Facility 315
- Example: Import XML Data by Using the XMLV2 Engine 316
- Example: Import JSON Data by Using the JSON Engine 318

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 247.

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 298.
- 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 308.

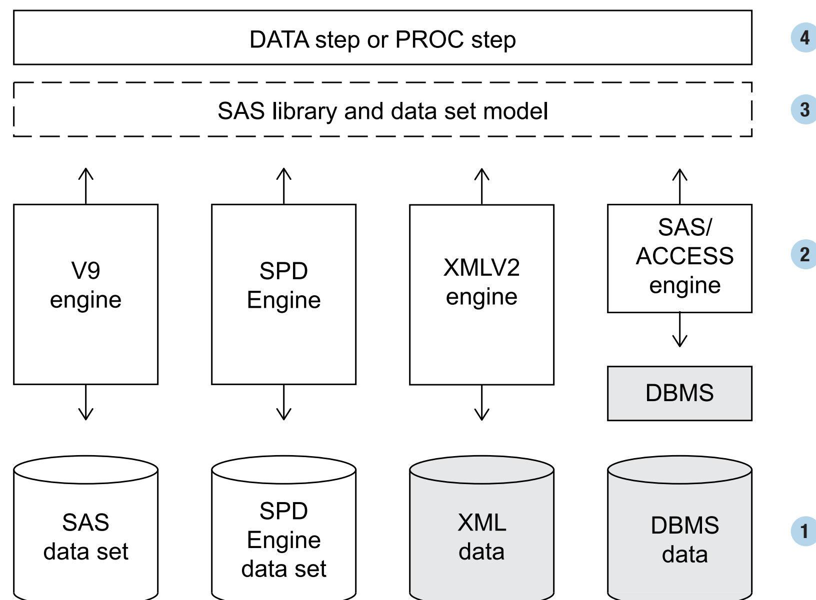
You specify an engine in a library assignment or in the ENGINE system option:

- 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 250.
- 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 307.

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 353.

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

LIBNAME Engine Nickname	Uses	Examples and Documentation
V9 or BASE	Shipped default Base SAS engine SAS data sets	"Example: Assign the V9 Engine in a LIBNAME Statement" on page 298 "Example: Read a Comma-Delimited File" on page 308 "Example: Set a Default Engine" on page 307
		"Default Base SAS Engine (V9 Engine)" on page 294 *SAS V9 LIBNAME Engine: Reference*
SAS/ACCESS engines (multiple)	External data from other applications	"Example: Read Microsoft Excel Data by Using a SAS/ACCESS Engine and PROC IMPORT" on page 310 "Example: Create Data in a DBMS by Using a SAS/ACCESS Engine" on page 311 "Example: Embed a SAS/ACCESS LIBNAME Statement in a PROC SQL View" on page 313 "Example: Access DBMS Data by Using the SQL Pass-Through Facility" on page 315
		SAS/ACCESS for Relational Databases: Reference *SAS/ACCESS for Nonrelational Databases: Reference* *SAS/ACCESS Interface to PC Files: Reference* *SAS/ACCESS Interface to R/3: User's Guide* SAS/ACCESS documentation

LIBNAME Engine Nickname	Uses	Examples and Documentation
CAS	Big data Multi-threaded distributed processing Cloud computing	"Example: Load a SAS Data Set to a CAS Server" on page 305
		"CAS LIBNAME Engine Overview" in *SAS Cloud Analytic Services: User's Guide*
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 300 "Example: Read and Write SAS Data in Hadoop by Using the SPD Engine" on page 301
		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
		SAS Viya Engine for ORC: Reference
CVP	National language support	"Example: Avoid Truncation by Using the CVP Engine with the V9 Engine" on page 302
		SAS National Language Support (NLS): Reference Guide
XMLV2	Import and export XML in Base SAS	"Example: Import XML Data by Using the XMLV2 Engine" on page 316
		SAS XMLV2 and XML LIBNAME Engines: User's Guide
JSON	Import and export JSON in Base SAS	"Example: Import JSON Data by Using the JSON Engine" on page 318
		"LIBNAME Statement: JSON Engine" in *SAS Global Statements: Reference*
SASESOCK	SAS/CONNECT TCP/IP piping	"LIBNAME: SASESOCK Engine" in *SAS/CONNECT User's Guide*
SASIOLA and SASHDAT	SAS LASR Analytic Server	SAS LASR Analytic Server: Reference Guide

LIBNAME Engine Nickname	Uses	Examples and Documentation
	Multi-threaded distributed processing Optional Hadoop storage	
FEDSVR	External data or SAS data by using the SAS Federation Server	*SAS LIBNAME Engine for SAS Federation Server: User's Guide*
EDOC	ODS DOCUMENT output objects	"LIBNAME Statement: SASEDOC" in *SAS Output Delivery System: User's Guide*
JMP	JMP statistical discovery software	"LIBNAME Statement: JMP Engine" in *SAS/ACCESS Interface to PC Files: Reference*
INFOMAP	SAS Information Maps	*Base SAS Guide to Information Maps*
META	SAS Metadata Server	*SAS Language Interfaces to Metadata*

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 298. See also "Example: Set a Default Engine" on page 307.

The file format for SAS 9, SAS 8, and SAS 7 data sets is very similar, so SAS does not differentiate between them. See "Cross-Release and Cross-Environment Compatibility" 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 298.
- 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 CNTLLEV= 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.

```
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*

```
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 259
- “Elements of a Library Assignment” on page 250
- “Default Base SAS Engine (V9 Engine)” on page 294

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.

```
libname mylib spde 'c:\examples' /*1*/
   datapath=('d:\mydatadir')     /*2*/
   indexpath=('e:\myindexdir');  /*3*/
```

1 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.

2 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.

3 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*

```
1    libname mylib spde 'c:\examples'
2       datapath=('d:\mydatadir')
3       indexpath=('e:\myindexdir');
NOTE: Libref MYLIB was successfully assigned as follows:
      Engine:        SPDE
      Physical Name: c:\examples\
3   !
```

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 292

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.

```
options set=SAS_HADOOP_CONFIG_PATH='\\myconfigpath';        /* 1 */
options set=SAS_HADOOP_JAR_PATH='\\myconfigpath';

libname mydata spde '/user/abcdef' hdfs=yes accelwhere=yes; /* 2 */
```

1 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.

2 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 298. Run PROC CONTENTS to see the length of the variables:

```
libname myfiles v9 'c:\examples';
proc contents data=myfiles.myclass;
run;
```

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*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	Age	Num	8
4	Height	Num	8
1	Name	Char	8
2	Sex	Char	1
5	Weight	Num	8

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 × 2.5 = 20.

For Sex, 1 × 2.5 = 2.5, which is 3 when rounded up to a whole number.

Output 13.2 *PROC CONTENTS Showing Variable Lengths after Expansion*

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
3	Age	Num	8
4	Height	Num	8
1	Name	Char	20
2	Sex	Char	3
5	Weight	Num	8

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.

See Also

- *SAS National Language Support (NLS): Reference Guide*
- "Compatibility and Migration" in *SAS V9 LIBNAME Engine: Reference*
- "COPY Procedure" in *Base SAS Procedures Guide*

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.

```
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.

Output 13.3 *Portion of PROC CONTENTS Output for mycas.cars*

Data Set Name	MYCAS.CARS	Observations	428
Member Type	DATA	Variables	15
Engine	CAS	Indexes	0
Created	05/17/2019 17:52:37	Observation Length	160
Last Modified	05/17/2019 17:52:37	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

Engine/Host Dependent Information	
Data Limit	100MB
Caslib	CASUSERHDFS(userid)
Scope	Session

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

- "LIBNAME Statement: CAS Engine" in *SAS Cloud Analytic Services: User's Guide*
- "Example: Load SAS Data to CAS by Using the CASUTIL Procedure" in *SAS V9 LIBNAME Engine: Reference*
- *An Introduction to SAS Viya Programming*

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 294
■ "Default Engine for a Mixed Library" in *SAS V9 LIBNAME Engine: Reference*
■ "Operating Environment Information" on page 298

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.

```
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.

Output 13.4 *PROC PRINT of work.mycholesterol*

patnr	measurement	cholesterol
A2	1	150
A2	2	145
A2	3	148
B1	1	301
B1	2	280
B1	3	275
C1	1	212
C1	2	220
C1	3	240

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 294
- Chapter 15, "Raw Data," on page 353
- "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 and PROC IMPORT

Example Code

This example uses the XLSX engine and PROC IMPORT to import Excel data. To create the data for this example, start Microsoft Excel and open the cholesterol.csv file that you used in "Example: Read a Comma-Delimited File" on page 308. (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. The following code imports cholesterol.xlsx as a SAS data set, mycholesterol.

```
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*
- "IMPORT Procedure" in *Base SAS Procedures Guide*
- "VALIDVARNAME= System Option" in *SAS System Options: 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 mytddata teradata server=mytera user=myid password=mypw; /*1*/
data mytddata.grades;                                            /*2*/
   input 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 13.5 PROC DATASETS Output Showing the DBMS Directory Information

Directory	
Libref	MYTDDATA
Engine	TERADATA
Physical Name	mytera
Schema/User	myid

Output 13.6 PROC DATASETS Output Showing the Library Contents for DBMS Content

#	Name	Member Type	DBMS Member Type
1	grades	DATA	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.

```
libname viewlib v9 '/mydata';                        /* 1 */
proc sql;
   create view viewlib.mygrades as                   /* 2 */
      select *
         from mytddata.grades                        /* 3 */
         using libname mytddata teradata
                       server=mytera
                       user=myid password=mypw;  /* 4 */
quit;
proc print data=viewlib.mygrades noobs;              /* 5 */
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 `mytddata.grades` table is referenced in the view.

4 The USING argument embeds the LIBNAME statement for the SAS/ACCESS Interface to Teradata engine.

5 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*

student	test1	test2	final
Wilma	97	91	98
Fred	66	80	70

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. You can embed a LIBNAME statement for most SAS/ACCESS engines, the V9 engine, and other engines.
- 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.

  However, if you need the data to remain constant for multiple steps, a better choice is to read the data once, and store it in a SAS data set. This practice can also benefit performance. Reading a SAS data set multiple times is usually more efficient than reading a DBMS table multiple times.

See Also

- "LIBNAME Statement: External Databases" in *SAS/ACCESS for Relational Databases: Reference*
- "CREATE VIEW" 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.

```
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*

student	test1	test2	final
Wilma	97	91	98

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.
- You can use SQL 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.

See Also

- "SQL Pass-Through Facility" in *SAS/ACCESS for Relational Databases: Reference*
- "SAS Views of DBMS Data" in *SAS/ACCESS for Relational Databases: Reference*
- "SAS Views" in *SAS V9 LIBNAME Engine: Reference*
- SAS/ACCESS documentation

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 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.

```
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*

TEAM_name	TEAM_abbrev
Thrashers	ATL
Hurricanes	CAR
Panthers	FLA
Lightning	TB
Capitals	WSH
Stars	DAL
Kings	LA
Ducks	ANA
Coyotes	PHX
Sharks	SJ

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*

Directory	
Libref	MYDATA
Engine	JSON
Access	READONLY
Physical Name	c:\examples\example.json

#	Name	Member Type
1	ALLDATA	DATA
2	INFO	DATA
3	INFO_PHONE	DATA
4	ROOT	DATA

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*

14

SAS Data Sets

Definitions for SAS Data Sets ... **322**
Definition of a SAS Data Set ... 322
Parts of a SAS Data Set ... 322

Managing SAS Data Sets ... **323**
Ways to Read and Create SAS Data Sets ... 323
Ways to Manage Variables in SAS Data Sets ... 323
Ways to Read Rows in SAS Data Sets ... 326
Rules for Naming SAS Data Sets ... 327
Data Set Name Lists ... 327

Generation Data Sets ... **329**

Examples: Create and Read SAS Data Sets ... **329**
Example: Read a Single SAS Data Set ... 329
Example: Read Multiple SAS Data Sets ... 331
Example: Read a SAS Data Set from a User-defined Library ... 333
Example: Read a Permanent SAS Data Set Without Using a Libref ... 335

Examples: Control Variables and Observations in Data Sets ... **337**
Example: Keep Specific Variables in a Data Set ... 337
Example: Control Which Observations are Read Using the WHERE Statement ... 339
Example: Read a Specific Observation First (FIRSTOBS) ... 341
Example: Read a Range of Observations (FIRSTOBS and OBS) ... 342
Example: Read an Observation Directly Using the POINT= Option ... 344

Examples: View Descriptor and Sort Information for Data Sets ... **346**
Example: View Descriptor Information for a Data Set ... 346
Example: View Sort Information for a Data Set ... 348

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 data views, see “Definitions for SAS Views” on page 393.

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*.
- “Data” in *SAS Cloud Analytic Services: Fundamentals*.

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:

```
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 403.

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 289.

Table 14.1 Ways to Manage Data Sets

Task	More Information
Read and Create a SAS data set on page 329	V9 Engine LIBNAME statement
Copy a SAS data set	DATASETS procedure
Get information about a SAS data set	

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 282.

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

Category	Task	More Information
Control variables (columns)	Drop variables from output Drop variables from input	DROP statement and DROP= data set option

Category	Task	More Information
	Keep variables in the output data set Keep variables in the input data set	KEEP statement and KEEP= data set option
	Rename variables Rename variables in the output data set output	RENAME statement and RENAME= data set option
Control rows	Conditionally select rows by using the WHERE statement Conditionally select rows by using the IF statement	WHERE statement WHERE= data set option subsetting IF statement
	Delete rows based on a condition	DELETE statement
	Remove a row	REMOVE statement
	Write the current row to the output data set Write the current row when a specified condition is true Create multiple rows from one line of input Create one row from multiple lines of input	OUTPUT statement
Directly access rows	Modify rows based on row number “Example: Read an Observation Directly Using the POINT= Option” on page 344	POINT= statement option
	Modify rows based on row number	KEY= statement option
	Read a range of rows using the FIRSTOBS= and OBS= data set options Process a specific row first in the data set	FIRSTOBS= data set option (V9 engine only)
	Specify when to stop processing rows	OBS= data set option (V9 engine only)

Table 14.3 Ways to Control the Reading and Writing of Variables and Rows

Category	Task	More Information
Control variables	Drop variables from output Drop variables from input	DROP statement and DROP= data set option V9 Engine LIBNAME statement
	Keep variables in the output data set Keep variables in the input data set	KEEP statement and KEEP= data set option
	Rename variables Rename variables in the output data set output	RENAME statement and RENAME= data set option
Control rows	Conditionally select rows by using the WHERE statement Conditionally select rows by using the IF statement	WHERE statement and subsetting IF statement
	Delete rows based on a condition	DELETE statement
	Remove a row	REMOVE statement
	Write the current row to the output data set Write the current row when a specified condition is true Create multiple rows from one line of input Create one row from multiple lines of input	OUTPUT statement
Directly access rows	Modify rows based on row number	POINT= statement option
	Modify rows based on row number	KEY= statement option
	Read a range of rows using the FIRSTOBS= and OBS= data set options	FIRSTOBS= data set option
	Process a specific row first in the data set	FIRSTOBS= system option
	Specify when to stop processing rows	OBS= data set option

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 POINT= option in the SET or MODIFY statements. The POINT= option names a temporary variable whose current value determines which row that a SET or 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:

```
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 KEY= option in the SET or 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 KEY= option in the SET statement to perform a "table lookup" on the indexed table to combine only those rows with the first data set.

```
data combine;
   set invtory(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*

Language Element Used	Example
INDEX= data set option	Create an Index Using the INDEX= Data Set Option
INDEX CREATE statement (DATASETS procedure)	"Modifying SAS Data Sets" in *Base SAS Procedures Guide*
CREATE INDEX statement (SQL procedure)	"Creating an Index" in *SAS SQL Procedure User's Guide*

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 56
- Summary of Rules for Naming SAS Data Sets on page 61
- "Special Data Set Names" on page 59
- "Example: Create a SAS Data Set Name Containing a Special Character" on page 64 and "Example: Create a Two-Level Data Set Name" on page 70

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 represents. 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 99 for more information about creating character and numeric variable lists in SAS code.

See Also

"Definitions for SAS Data Sets" on page 322

Generation Data Sets

A generation data set is an archived version of a SAS data set that is created when the data set is updated. For more information about generation data sets, see "Definitions for Generation Data Sets" in *SAS V9 LIBNAME Engine: Reference*.

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`).

```
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*

Obs	Region	Product	Subsidiary	Stores	Sales	Inventory	Returns
1	Canada	Men's Dress	Vancouver	28	$757,798	$1,847,559	$16,833
2	Canada	Slipper	Vancouver	27	$700,513	$2,520,085	$21,247
3	Canada	Women's Dress	Vancouver	21	$756,347	$2,503,387	$19,378
4	Central America/Caribbean	Men's Casual	Kingston	28	$576,112	$1,159,556	$20,005
5	Middle East	Men's Casual	Tel Aviv	11	$1,298,717	$2,881,005	$57,362
6	Western Europe	Women's Casual	Copenhagen	26	$502,636	$1,110,412	$17,448

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 247 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 326.

See Also

- SET statement, DATA statement, and "WHERE Statement" in *SAS DATA Step Statements: Reference*
- "Sashelp Library" on page 258 in *SAS Programmer's Guide: Essentials*
- "Ways to Read Rows in SAS Data Sets" on page 326

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.

```
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

Obs	DATE	AMOUNT
1	01JAN1997	-30000
2	01JAN1998	7500
3	01JAN1999	7500
4	01JAN2000	7500
5	01JAN2001	7500
6	01JAN2002	7500
7	01JAN1997	-60000
8	01JAN1998	13755
9	01JAN1999	13755
10	01JAN2000	13755
11	01JAN2001	13755
12	01JAN2002	23755
13	01JAN1997	-20000
14	01JAN1998	5000
15	01JAN1999	5000
16	01JAN2000	5000
17	01JAN2001	5000
18	01JAN2002	5000

Sashelp.nvst1 (observations 1–6)
Sashelp.nvst2 (observations 7–12)
Sashelp.nvst3 (observations 13–18)

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 478 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 247 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 326.

See Also

- “SET Statement” in *SAS DATA Step Statements: Reference*
- “WHERE Statement” in *SAS DATA Step Statements: Reference*
- “Sashelp Library” on page 258

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.

```
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";
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*

Earthquakes by Magnitude

Obs	Latitude	Longitude	Depth	Magnitude	dNearestStation	RootMeanSquareTime	Type
1	36.8408	-97.875	4.2510	2.5	.	0.4600	earthquake
2	36.0210	-97.097	4.0420	2.5	.	0.6300	earthquake
3	36.7038	-98.041	12.7600	2.5	0.20300	0.2500	earthquake
4	36.7554	-97.556	6.2220	2.5	.	0.7600	earthquake
5	41.9035	-119.662	1.7742	2.5	0.47700	0.2545	earthquake
6	36.3510	-84.995	20.7000	2.5	0.34136	0.2000	earthquake
7	35.7968	-97.444	5.4500	2.5	.	0.3500	earthquake
8	36.7484	-97.649	3.3550	2.5	.	0.5500	earthquake
9	41.8692	-119.615	5.0000	2.5	0.65577	0.6500	earthquake
10	41.8652	-119.673	0.2000	2.5	0.46712	0.4500	earthquake

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 247 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 326.

See Also

- Chapter 12, "SAS Libraries," on page 247
- "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`.

```
data "c:\Users\demo\shoesales2.sas7bdat";
```

```
   set "c:\Users\demo2\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:

Output 14.5 *Log Output*

```
NOTE: There were 395 observations read from the data set c:\Users\demo
\shoesales.sas7bdat.
NOTE: The data set c:\Users\demo2\shoesales2.sas7bdat has 395 observations and 8
variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds
```

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 247 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 326.

See Also

- Chapter 12, “SAS Libraries,” on page 247
- “SAS Default Libraries” on page 256

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.

```
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*

Obs	Make	MSRP	Mpg
1	Acura	$36,945	13.975
2	Acura	$23,820	19.325
3	Acura	$26,990	17.875
4	Acura	$33,195	16.700
5	Acura	$43,755	14.700
6	Acura	$46,100	14.700
7	Acura	$89,765	14.250
8	Audi	$25,940	18.425
9	Audi	$35,940	18.600
10	Audi	$31,840	16.700

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 323 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.

```
data class;
   set sashelp.class;
   where age>12 and height>=67;
run;
proc print data=class; run;
```

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69	112.5
2	Philip	M	16	72	150.0
3	Ronald	M	15	67	133.0

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`**.

```
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*

Obs	Depth	Magnitude	Type
1	11.25	6.02	earthquake
2	9.45	6.60	earthquake
3	13.00	6.30	earthquake
4	13.00	7.00	earthquake
5	4.00	7.20	earthquake
6	10.00	6.20	earthquake
7	10.00	6.90	earthquake
8	14.00	6.60	earthquake

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.

```
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*

Obs	Depth	Magnitude	Type
1	4	7.2	earthquake
2	10	6.2	earthquake
3	10	6.9	earthquake
4	14	6.6	earthquake

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: **`(obs - firstobs) + 1 = 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*

Quakes

Obs	Depth	Magnitude	Type
1	11.25	6.02	earthquake
2	9.45	6.60	earthquake
3	13.00	6.30	earthquake
4	13.00	7.00	earthquake
5	4.00	7.20	earthquake
6	10.00	6.20	earthquake
7	10.00	6.90	earthquake
8	14.00	6.60	earthquake

Quakes2

FIRSTOBS=2, OBS=4

Obs	Depth	Magnitude	Type
1	9.45	6.6	earthquake
2	13.00	6.3	earthquake
3	13.00	7.0	earthquake

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 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.

```
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)*

Comet Data Set

Obs	Dose	Rat	Sample	Length
1	0	1	1	15.3527
2	0	1	1	16.1826
3	0	1	1	14.9378
4	0	1	1	12.4481
5	0	1	1	12.8631

Output 14.14 *PROC Print Output for the Comet Data Set Showing the Directly Accessed Row 3*

Row 3 of Comet Data Set

Obs	Dose	Rat	Sample	Length
1	0	1	1	14.9378

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
- "CALL SYMPUT Routine" in *SAS Macro Language: Reference*

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`.

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

Output 14.15 *PROC CONTENTS Output Showing the Descriptor Information for the Sashelp.Snacks Data Set*

The CONTENTS Procedure

Data Set Name	SASHELP.SNACKS	Observations	35770
Member Type	DATA	Variables	6
Engine	V9	Indexes	0
Created	06/25/2018 03:57:43	Observation Length	80
Last Modified	06/25/2018 03:57:43	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label	Daily snack food sales		
Data Representation	WINDOWS_64		
Encoding	us-ascii ASCII (ANSI)		

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	44
First Data Page	1
Max Obs per Page	817
Obs in First Data Page	796
Number of Data Set Repairs	0
ExtendObsCounter	YES
Filename	c:\snacks.sas7bdat
Release Created	9.0501M0
Host Created	X64_SR12R2
Owner Name	sasuser
File Size	3MB
File Size (bytes)	2949120

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format	Label
3	Advertised	Num	8		Advertised (1=yes)
5	Date	Num	8	DATE9.	Date of sale
4	Holiday	Num	8		Holiday (1=yes)
2	Price	Num	8		Retail price of product
6	Product	Char	40		Product name
1	QtySold	Num	8		Quantity sold

Sort Information	
Sortedby	Product Date
Validated	YES
Character Set	ANSI

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.

See Also

- "CONTENTS Procedure" in *Base SAS Procedures Guide*
- "OPTIONS Statement" in *SAS Global Statements: Reference*

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*

The CONTENTS Procedure

Data Set Name	SASHELP.AIR	Observations	144
Member Type	DATA	Variables	2
Engine	V9	Indexes	0
Created	06/03/2018 21:41:44	Observation Length	16
Last Modified	06/03/2018 21:41:44	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label	airline data (monthly: JAN49-DEC60)		
Data Representation	WINDOWS_64		
Encoding	us-ascii ASCII (ANSI)		

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.

Output 14.17 PROC CONTENTS Output Showing That the Data Set Was Sorted Using SORTEDBY

The CONTENTS Procedure

Data Set Name	WORK.AIR	Observations	144
Member Type	DATA	Variables	2
Engine	V9	Indexes	0
Created	06/07/2018 19:15:45	Observation Length	16
Last Modified	06/07/2018 19:15:45	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

Sort Information	
Sortedby	AIR
Validated	NO
Character Set	ANSI

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 CONTENTS Procedure

Data Set Name	WORK.AIR	Observations	144
Member Type	DATA	Variables	2
Engine	V9	Indexes	0
Created	06/07/2018 19:19:44	Observation Length	16
Last Modified	06/07/2018 19:19:44	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

Sort Information	
Sortedby	DESCENDING AIR
Validated	YES
Character Set	ANSI

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*

15

Raw Data

Definitions for Raw Data 353

Reading Raw Data 354
- Ways to Read Raw Data 354
- Reading Raw Instream Data 355
- Reading Raw Data with the DATA Step (INPUT Statement) 356
- Types of Raw Data 363
- How SAS Handles Invalid Data 366
- How SAS Handles Missing Values 367

Definitions for External Files 369

Reading External Files 370
- Reading an External File Using the DATA Step 370
- Reading and Writing to External Files Directly 371
- Referencing External Files Indirectly By Using a Fileref 372
- Referencing Many External Files Efficiently 373
- Referencing External Files with Other Access Methods 373
- Reading Binary Data 375

Examples: Read External Files Using PROC IMPORT 378
- Example: Read a Comma-Delimited File 378
- Example: Read and Write a Space-delimited File 380

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 385.

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 356
- SAS I/O functions, such as the FOPEN, FGET, and 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 15.

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:

```
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:

```
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
;;;;
```

Reading Raw Data with the DATA Step (INPUT Statement)

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:

```
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 359 .

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 informats:

- **&** (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:

```
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;
```

Output 15.1 *Output from Example with Format Modifiers*

The SAS System

Name	Score1	Score2	Score3	Team	Div
Smith	12	22	46	"Green Hornets, Atlanta"	AAA
Mitchel	23	19	25	"High Volts, Portland"	AAA
Jones	9	17	54	"Vulcans, Las Vegas"	AA

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:

```
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.

```
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.

```
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

Input	Goal	Use
multiple records	create a single observation	#n or / line pointer control in the INPUT statement, with a DO loop
a single record	create multiple observations	trailing @@ in the INPUT statement, trailing @ with multiple INPUT and OUTPUT statements (Example)
variable-length data fields and records	read delimited data	list input with or without a format modifier in the INPUT statement and the TRUNCOVER, DLM=, DLMSTR=, or , or DSD options in the INFILE statement (Examples)
	read non-delimited data	$VARYING*w*. informat in the INPUT statement and the LENGTH= and TRUNCOVER options in the INFILE statement (Example)
a file with varying record layouts		IF-THEN statements with multiple INPUT statements, using trailing @ or @@ as necessary
hierarchical files		IF-THEN statements with multiple INPUT statements, using trailing @ as necessary

Input	Goal	Use
more than one input file or to control the program flow at EOF		EOF= option or END= option in an INFILE statement (Example) multiple INFILE and INPUT statements FILEVAR= in an INFILE statement. (Example) FILENAME statement with concatenation, wildcard, or piping
only part of each record		LINESIZE= option in an INFILE statement
some but not all records in the file		FIRSTOBS= and OBS= options in an INFILE statement; FIRSTOBS= and OBS= system options; #n line pointer control (Example)
instream data lines	control the reading with special options	INFILE statement with DATALINES and appropriate options
starting at a particular column		@ column pointer controls
leading blanks	maintain them	$CHARw. informat in an INPUT statement
a delimiter other than blanks (with list input or modified list input with the colon modifier)		DLM= option or the DSD option both the DLM= option and the DSD option in an INFILE statement
the standard tab character		DLM= or in an INFILE statement both the DLM= and the DLMSTR= options EXPANDTABS option in an INFILE statement
missing values (with list input or	create observations without compromising data integrity	TRUNCOVER option in an INFILE statement DLM=, DLMSTR=, or , or DSD might also be needed

Input	Goal	Use
modified list input with the colon modifier)	protect data integrity by overriding the default behavior	

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

Data	Description	Solution
`  23`	input right aligned	None needed
`23`	input not aligned	None needed
`23`	input left aligned	None needed
`00023`	input with leading zeros	None needed
`23.0`	input with decimal point	None needed
`2.3E1`	in E notation, 2.30 (ss1)	None needed
`230E-1`	in E notation, 230x10 (ss-1)	None needed
`-23`	minus sign for negative numbers	None needed

Table 15.3 Reading Nonstandard Numeric Data

Data	Description	Solution
`2 3`	embedded blank	COMMA. or BZ. informat
`- 23`	embedded blank	COMMA. or BZ. informat
`2,341`	comma	COMMA. informat
`(23)`	parentheses	COMMA. informat
`C4A2`	hexadecimal value	HEX. informat
`1MAR90`	date value	DATE. informat

Table 15.4 Reading Invalid Numeric Data

Data	Description	Solution
`23 -`	minus sign follows number	Put minus sign before number or solve programmatically. It might be possible to use the

Data	Description	Solution
		S370FZDTw.d informat, but positive values require the trailing plus sign (+).
`. .`	double instead of single periods	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.
`J23`	not a number	Read as a character value, or edit the raw data to change it to a valid number.

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 15.5 Reading Instream Data and External Files Containing Leading Blanks and Semicolons

Characters in the Data	What to Use	Reason
leading or trailing blanks that you want to preserve	formatted input and the $CHAR*w*. informat	List input trims leading and trailing blanks from a character value before the value is assigned to a variable.
semicolons in instream data	DATALINES4 or CARDS4 statements and four semicolons (;;;;) to mark the end of the data	With the normal DATALINES and CARDS statements, a semicolon in the data prematurely signals the end of the data.
delimiters, blank characters, or quoted strings	DSD option, with DLM= or DLMSTR= option in the INFILE statement	These options enable SAS to read a character value that contains a delimiter within a quoted string; these options can also treat two consecutive delimiters as a missing value and remove quotation marks from character values.

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 15 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 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 377.

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:

```
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:

```
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:

```
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
;

proc print;
run;
```

Output 15.2 *Output of Data with Special Missing Numeric Values*

The SAS System

Obs	name	Answer1	Answer2	Answer3
1	Smith	2	5	9
2	Jones	4	B	8
3	Carter	A	4	7
4	Reed	3	5	C

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 377 for a 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 15 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 385 and *SAS/ACCESS for Relational Databases: Reference*.
- For information about access to PC files, see Chapter 16, "Database and PC Files," on page 385 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.

```
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 15.6 Reading Data from an External File Directly

Task	Language Element	Example

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

Task	Language Element	Example

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

Task	Language Element	Example
Bring statements or raw data from another file into your SAS job and execute them.	%INCLUDE statement	%include '*myprogram.txt*';

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.

Table 15.9 Referencing External By Using a Fileref

Task	Language Element	Example

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;

1. In some operating environments, you can also use the command '&' to assign a fileref.

Task	Language Element	Example
		`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 15.10 *Referencing Many Files Efficiently*

Task	Language Element	Example
Assign a fileref to aggregate storage location.	FILENAME statement	`filename mydir 'directory-name';`
"Reading from Multiple Input Files" in *SAS DATA Step Statements: Reference*		
"Reading from Multiple Input Files" in *SAS DATA Step Statements: Reference*		

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

Task	Language Element	Example
Assign a fileref to a SAS catalog that is an aggregate storage location.	FILENAME statement with CATALOG specifier	`filename mycat catalog 'catalog' <catalog-options>;`
Assign a fileref to an external file accessed by a data URL.	FILENAME statement with DATAURL specifier	`filename myfile dataurl 'external-file' <dataurl-options>;`
Assign a fileref to an external file accessed with FTP.	FILENAME statement with FTP specifier	`filename myfile FTP 'external-file' <ftp-options>;`
Assign a fileref to an external file accessed on a Hadoop Distributed File System.	FILENAME statement with Hadoop specifier	`filename myfile hadoop 'external-file' <hadoop-options>;`
Assign a fileref to an external file accessed with SFTP.	FILENAME statement with SFTP specifier	`filename myfile SFTP 'external-file' <sftp-options>;`
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	FILENAME statement with WebDAV specifier	`filename myfile WEBDAV 'external-file' <webdav-options>;`

Task	Language Element	Example
on a WebDAV server.		
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*

Description	Native Mode Informats	IBM 370 Mode Informats
ASCII Character	$*w*.	$ASCII*w*.

Description	Native Mode Informats	IBM 370 Mode Informats
ASCII Numeric	*w.d*	$ASCII*w.*
EBCDIC Character	$*w.*	$EBCDIC*w.*
EBCDIC Numeric (Standard)	*w.d*	S370FF*w.d*
Integer Binary	IB*w.d*	S370FIB*w.d*
Positive Integer Binary	PIB*w.d*	S370FPIB*w.d*
Real Binary	RB**w.d**	S370FRB*w.d*
Unsigned Integer Binary	PIB*w.d*	S370FIBU*w.d*, S370FPIB*w.d*
Packed Decimal	PD*w.d*	S370FPD*w.d*
Unsigned Packed Decimal	PK*w.d*	S370FPDU*w.d* or PK*w.d*
Zoned Decimal	ZD*w.d*	S370FZD*w.d*
Zoned Decimal Leading Sign	S370FZDL*w.d*	S370FZDL*w.d*
Zoned Decimal Separate Leading Sign	S370FZDS*w.d*	S370FZDS*w.d*
Zoned Decimal Separate Trailing Sign	S370FZDT*w.d*	S370FZDT*w.d*
Unsigned Zoned Decimal	ZD*w.d*	S370FZDU*w.d*

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.

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*

Informat Name	Description
$CB*w*.	reads standard character data from column-binary files
CB*w*.	reads standard numeric data from column-binary files
PUNCH.*d*	reads whether a row is punched
ROW*w.d*	reads a column-binary field down a card column

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:

```
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.

Examples: Read External Files Using PROC IMPORT

Example: Read a Comma-Delimited File

Example Code

In this example, the IMPORT procedure imports a file that contains comma-separated values.

```
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.

Output 15.3 *PROC PRINT of work.mycholesterol*

patnr	measurement	cholesterol
A2	1	150
A2	2	145
A2	3	148
B1	1	301
B1	2	280
B1	3	275
C1	1	212
C1	2	220
C1	3	240

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 294
- Chapter 15, "Raw Data," on page 353
- "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.

```
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*

Obs	DATE	AIR
1	JAN1949	112
2	FEB1949	118
3	MAR1949	132
4	APR1949	129
5	MAY1949	121
6	JUN1949	135
7	JUL1949	148
8	AUG1949	148
9	SEP1949	136
10	OCT1949	119
11	NOV1949	104

Output 15.5 *Log Output for Reading a Space-Delimited Text File Using PROC IMPORT*

```
1430   /**********************************************************************
1431   *   PRODUCT:   SAS
1432   *   VERSION:   9.4
1433   *   CREATOR:   External File Interface
1434   *   DATE:      17OCT19
1435   *   DESC:      Generated SAS Datastep Code
1436   *   TEMPLATE SOURCE:  (None Specified.)
1437   ***********************************************************************/
1438      data WORK.AIR;              1
1439      %let _EFIERR_ = 0; /* set the ERROR detection macro variable */
1440      infile 'c:\Users\sasuser\txt\air.txt' delimiter = ' '
             MISSOVER                 2
             DSD                      3
             lrecl=32767
1440!        firstobs=2;              4
1441         informat DATE MONYY7.;   5
1442         informat AIR best32. ;
1443         format DATE MONYY7. ;    6
1444         format AIR best12. ;
1445      input                                              7
1446                  DATE
1447                  AIR
1448      ;
1449      if _ERROR_ then call symputx('_EFIERR_',1);  /* set ERROR detection
macro variable */
1450      run;
54
```

1 PROC IMPORT automatically generates this DATA step and prints the step to the SAS Log.

2 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.

3 The DSD option specifies that when data values are enclosed in quotation marks, delimiters within the value are treated as character data.

4 FIRSTOBS= data set option specifies the first observation to be processed in the data set.

5 MONYYw. informat reads month and year date values in the form monyy.

6 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.

7 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*

16

Database and PC Files

Definition of SAS/ACCESS Software 385

Dynamic LIBNAME Engine 386
SAS/ACCESS LIBNAME Statement 386
Using Data Set Options with SAS/ACCESS Librefs 386
Embedding a SAS/ACCESS LIBNAME Statement in a PROC SQL View 387

SQL Procedure Pass-Through Facility 387

ACCESS Procedure and Interface View Engine 389

DBLOAD Procedure 390

Interface DATA Step Engine 390

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 289 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:

```
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:

```
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.

```
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.
      @140 custstat $char2.
      @142 custland $char20.
      @162 custzip  $char10.
      @172 h_phone  $char12.
      @184 o_phone  $char12.;
  if _error_ = 1 then
    abort abend 888;
run;
```

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.

17

SAS Views

Definitions for SAS Views 393

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.

PROC SQL view
: is a stored query expression that is created in PROC SQL. 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. *SAS V9 LIBNAME Engine: Reference*

18

SAS Dictionary Tables

Definition of a DICTIONARY Table 395

How to View DICTIONARY Tables 396
About Dictionary Tables 396
How to View a DICTIONARY Table 396
How to View a Summary of a DICTIONARY Table 396
How to View a Subset of a DICTIONARY Table 397
DICTIONARY Tables and Performance 398

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 “What Are DICTIONARY Tables?” in *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 396.

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.

```
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:

Output 18.1 *Result of the PROC SQL Subsetting WHERE Statement*

Subset of the DICTIONARY.INDEX View
Rows with Column Name equal to STATE

Library Name	Member Name	Column Name
SASHELP	PLFIPS	STATE
MAPS	USAAC	STATE
MAPS	USAAC	STATE
MAPS	USAAS	STATE
MAPSSAS	USAAC	STATE
MAPSSAS	USAAC	STATE
MAPSSAS	USAAS	STATE

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 GENNUM variable value in CONTENTS OUT= data set with the GEN variable value from DICTIONARY tables. GENNUM 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:

```
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

Chapter 19
Grouping Data 403

Chapter 20
Loops and Conditionals 421

Chapter 21
Combining Data 473

Chapter 22
Using Indexes 571

Chapter 23
Using Arrays 573

Chapter 24
Debugging Errors 595

Chapter 25
Optimizing System Performance 635

Chapter 26
Using Parallel Processing 651

19

Grouping Data

Definitions for BY-Group Processing **404**

Syntax **405**
- Syntax 405
- FIRST. and LAST. Automatic DATA Step Variables 405

Understanding BY Groups **405**
- BY Groups with a Single BY Variable 405
- BY Groups with Multiple BY Variables 407

Invoking BY-Group Processing **408**

Determining Whether the Data Requires Preprocessing for BY-Group Processing **409**

Preprocessing Input Data for BY-Group Processing **409**
- Sorting Observations for BY-Group Processing 409
- Indexing for BY-Group Processing 410

FIRST. and LAST. DATA Step Variables **410**
- How the DATA Step Identifies BY Groups 410
- How SAS Determines FIRST.variable and LAST.variable 411
- Using a Name Literal as the FIRST. and LAST. Variable 411
- Example 1: Grouping Observations by State, City, and ZIP Code 412
- Example 2: Grouping Observations by City, State, and ZIP Code 413
- Example 3: A Change Affecting the FIRST. variable 414

Processing BY-Groups in the DATA Step **415**
- Overview 415
- Processing BY-Groups Conditionally 416
- Data Not in Alphabetic or Numeric Order 417
- Data Grouped by Formatted Values 418
- Example 1: Using GROUPFORMAT with Formats 418
- Example 2: Using GROUPFORMAT with Formats 419

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 410 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)

BY variable

ZipCode	State	City	Street	
33133	FL	Miami	Rice St	BY group
33133	FL	Miami	Thomas Ave	
33133	FL	Miami	Surrey Dr	
33133	FL	Miami	Trade Ave	
33146	FL	Miami	Nervia St	BY group
33146	FL	Miami	Corsica St	
33801	FL	Lakeland	French Ave	BY group
33809	FL	Lakeland	Egret Dr	BY group
85730	AZ	Tucson	Domenic Ln	BY group
85730	AZ	Tucson	Gleeson Pl	

***Example Code 19.1** Create the Zip Data Set*

```
data zip;
input zipCode State $ City $ Street $20-29;
datalines;
85730 AZ Tucson    Domenic Ln
85730 AZ Tucson    Gleeson Pl
33133 FL Miami     Rice St
33133 FL Miami     Thomas Ave
33133 FL Miami     Surrey Dr
33133 FL Miami     Trade Ave
33146 FL Miami     Nervia St
33146 FL Miami     Corsica St
33801 FL Lakeland  French Ave
33809 FL Lakeland  Egret Dr
;
```

You can then specify the BY variable in the DATA step using the following code:

***Example Code 19.2** Sort and Group the zipCode Data Set by a Single Variable*

```
proc sort data=zip;
   by zipcode;
run;

data zip;
   set zip; by zipcode;
run;

proc print data=zip noobs;
   title 'BY-Group Uing a Single Variable: ZipCode';
run;
```

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`.

Figure 19.2 BY Groups with Multiple BY Variables (State and City)

BY variables

State	City	Street	ZipCode	
AZ	Tucson	Domenic Ln	85730	BY group
AZ	Tucson	Gleeson Pl	85730	
FL	Lakeland	French Ave	33801	BY group
FL	Lakeland	Egret Dr	33809	
FL	Miami	Nervia St	33146	BY group
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	

Here is the code for creating the output shown in the figure "BY Groups with Multiple BY Variables" on page 407 :

Example Code 19.3 Create the Zip Data Set

```
/* 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*

```
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.

```
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.

```
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 416 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:

```
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`.

```
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=
   first.state= last.state=
   first.ZipCode= last.ZipCode= ;
run;
```

Example Code 19.1 *Grouping Observations by State, City, and ZIP Code*

```
_N_=1  AZ Tucson   85730   FIRST.State=1 LAST.State=0   FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=2  AZ Tucson   85730   FIRST.State=0 LAST.State=1   FIRST.City=0 LAST.City=1
FIRST.ZipCode=0 LAST.ZipCode=1
_N_=3  FL Lakeland 33801   FIRST.State=1 LAST.State=0   FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=1
_N_=4  FL Lakeland 33809   FIRST.State=0 LAST.State=0   FIRST.City=0 LAST.City=1
FIRST.ZipCode=1 LAST.ZipCode=1
_N_=5  FL Miami    33133   FIRST.State=0 LAST.State=0   FIRST.City=1 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=6  FL Miami    33133   FIRST.State=0 LAST.State=0   FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=0
_N_=7  FL Miami    33133   FIRST.State=0 LAST.State=0   FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=0
_N_=8  FL Miami    33133   FIRST.State=0 LAST.State=0   FIRST.City=0 LAST.City=0
FIRST.ZipCode=0 LAST.ZipCode=1
_N_=9  FL Miami    33146   FIRST.State=0 LAST.State=0   FIRST.City=0 LAST.City=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=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*

Observations in Three BY Groups			Corresponding FIRST. and LAST. Variables					
State	**City**	**ZipCode**	**FIRST. State**	**LAST. State**	**FIRST. City**	**LAST. City**	**FIRST. ZipCode**	**LAST. ZipCode**
AZ	Tucson	85730	1	0	1	0	1	0
AZ	Tucson	85730	0	1	0	1	0	1
FL	Lakeland	33801	1	0	1	0	1	1
FL	Lakeland	33809	0	0	0	1	1	1
FL	Miami	33133	0	0	1	0	1	0
FL	Miami	33133	0	0	0	0	0	0
FL	Miami	33133	0	0	0	0	0	0
FL	Miami	33133	0	0	0	0	0	1
FL	Miami	33146	0	0	0	0	1	0
FL	Miami	33146	0	1	0	1	0	1

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=
       first.ZipCode= last.ZipCode=;
run;
proc print data=zip2; title 'By City, State, Zip'; run;
```

Example Code 19.2 *Grouping Observations by City, State, and ZIP Code*

```
_N_=1  Lakeland FL 33801 FIRST.City=1 LAST.City=0 FIRST.State=1 LAST.State=0
FIRST.ZipCode=1 LAST.ZipCode=1
_N_=2  Lakeland FL 33809 FIRST.City=0 LAST.City=1 FIRST.State=0 LAST.State=1
FIRST.ZipCode=1 LAST.ZipCode=1
_N_=3  Miami    FL 33133 FIRST.City=1 LAST.City=0 FIRST.State=1 LAST.State=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=4  Miami    FL 33133 FIRST.City=0 LAST.City=0 FIRST.State=0 LAST.State=0
FIRST.ZipCode=0 LAST.ZipCode=0
_N_=5  Miami    FL 33133 FIRST.City=0 LAST.City=0 FIRST.State=0 LAST.State=0
FIRST.ZipCode=0 LAST.ZipCode=0
_N_=6  Miami    FL 33133 FIRST.City=0 LAST.City=0 FIRST.State=0 LAST.State=0
FIRST.ZipCode=0 LAST.ZipCode=1
_N_=7  Miami    FL 33146 FIRST.City=0 LAST.City=0 FIRST.State=0 LAST.State=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=8  Miami    FL 33146 FIRST.City=0 LAST.City=1 FIRST.State=0 LAST.State=1
FIRST.ZipCode=0 LAST.ZipCode=1
_N_=9  Tucson   AZ 85730 FIRST.City=1 LAST.City=0 FIRST.State=1 LAST.State=0
FIRST.ZipCode=1 LAST.ZipCode=0
_N_=10 Tucson   AZ 85730 FIRST.City=0 LAST.City=1 FIRST.State=0 LAST.State=1
FIRST.ZipCode=0 LAST.ZipCode=1
```

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.

```
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= ;
run;

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= ;
run;
```

```
Grouped by X Y Z
_N_=1 FIRST.x=1 LAST.x=0 FIRST.y=1 LAST.y=0 FIRST.z=1 LAST.z=0
_N_=2 FIRST.x=0 LAST.x=0 FIRST.y=0 LAST.y=1 FIRST.z=0 LAST.z=1
_N_=3 FIRST.x=0 LAST.x=1 FIRST.y=1 LAST.y=1 FIRST.z=1 LAST.z=1
_N_=4 FIRST.x=1 LAST.x=1 FIRST.y=1 LAST.y=1 FIRST.z=1 LAST.z=1
```

```
Grouped by Y X Z
_N_=1 FIRST.y=1 LAST.y=0 FIRST.x=1 LAST.x=0 FIRST.z=1 LAST.z=0
_N_=2 FIRST.y=0 LAST.y=1 FIRST.x=0 LAST.x=1 FIRST.z=0 LAST.z=1
_N_=3 FIRST.y=1 LAST.y=0 FIRST.x=1 LAST.x=1 FIRST.z=1 LAST.z=1
_N_=4 FIRST.y=0 LAST.y=1 FIRST.x=1 LAST.x=1 FIRST.z=1 LAST.z=1
```

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.*variable* and LAST.*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.*variable* and LAST.*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.

```
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.variable to 1 if this is a new        */
      /* department in the BY group.                          */
   if first.Department then Payroll=0;
   Payroll+YearlyWage;
```

```
      /* SAS sets LAST.variable to 1 if this is the last      */
      /* department in the current BY group.                  */
    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

Obs	Department	Payroll
1	BAD	$952,000
2	DDG	$448,000
3	PPD	$522,000
4	STD	$496,000

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
- 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

```
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

```
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;
```

Output 19.2 *SAS Output Using the BY Statement GROUPFORMAT Option*

Score Categories

Score=Low

Obs	name	Score
1	Jon	Low

Score=Medium

Obs	name	Score
2	Anthony	Medium
3	Miguel	Medium
4	Joseph	Medium

Score=High

Obs	name	Score
5	Ian	High
6	Jan	High

20

Loops and Conditionals

Definitions for Loops and Conditionals **422**

Summary of Statements for Conditional Processing in SAS **423**

DO Loops **425**
- Types of DO Loops 425
- Forms of the Iterative DO Loop 427

WHERE Expressions **428**
- WHERE Statement 428
- Variables in WHERE Expressions 428
- Functions in WHERE Expressions 429
- Constants in WHERE Expressions 431
- Operators in WHERE Expressions 432
- Indexes and WHERE Expressions 442
- Data Set and System Options with WHERE Expressions 443

IF Statements **444**
- Types of IF Statements 444
- Subsetting IF Statement versus the WHERE Statement 444

SELECT WHEN Statement **446**
- See Also 447

Other Control Flow Statements **447**

Examples: DO Loops **449**
- Example: Use a Simple Iterative DO Loop 449
- Example: Use a Nested Iterative DO Loop 450
- Example: Use DOLIST Syntax to Iterate over a List of Values 452
- Example: Use Iterative DO TO Syntax to Iterate a Specific Number of Times 453
- Example: Use the Iterative DO TO BY Syntax to Iterate a Specific Number of Times by a Specific Interval 454
- Example: Use the Iterative DO WHILE and DO UNTIL Statements to Execute a Group of Statements Repetitively 455

Examples: WHERE Processing **457**
- Example: Conditionally Select Rows Using the WHERE Statement 457
- Example: Conditionally Select Rows Using a Compound WHERE Expression 459
- Example: Conditionally Select Rows Based on Multiple Conditions 460
- Example: Conditionally Select Rows Using the WHERE= Data Set Option 463
- Example: Use an Index to Improve Performance of WHERE Processing 463

Examples: IF Statements **464**

Example: Subset Data Using the Subsetting IF Statement 464
Example: Conditionally Select Specific Rows Using the IF Statement 466

Example: SELECT WHEN Statement **467**
Example Code 467
See Also 469

Example: Data Set Options **469**
Example Code 469
See Also 471

Definitions for Loops and Conditionals

The following terms are related to conditional and iterative programming in SAS:

Conditional processing
: statements or expressions in SAS that perform different computations or actions depending on whether a programmer-specified condition is true or false. For a list of these statements, see "Summary of Statements for Conditional Processing in SAS" on page 423.

Control flow
: programming constructs in SAS that control the sequence and flow of program execution. For example, conditional processing statements and DO loops are types of control flow statements. For a list of additional control flow statements in SAS, see "Other Control Flow Statements" on page 447.

  For more information, see "Altering the Flow for a Given Observation" on page 874.

DO group
: a group of SAS DATA step statements that begins with the DO statement and ends with the END statement. DO group statements are executed as a unit. A DO group can consist of an iterative DO statement (also known as a DO loop), or it can consist of a simple DO statement. Simple DO statements are often used with conditional IF-THEN/ELSE statements.

DO loop
: instructions that are continually repeated until a certain condition is reached. The iterative DO statement enables you to create DO loops. For more information, see "Types of DO Loops" on page 425.

DOLIST syntax
: a type of syntax that is based on the iterative DO loop, in which the loop iterates over the elements of a list. The list serves as the start-value in the DO statement and the items in the list are separated by commas. DOLIST syntax can also be specified in the iterative DO WHILE and DO UNTIL statements. For more information, see "Types of DO Loops" on page 425.

WHERE expression

a WHERE expression is a type of SAS expression that defines a condition for selecting rows to be processed in a SAS data set. The following WHERE statement defines a single condition for selecting rows:

```
where sales > 600000;
```

A *compound WHERE expression* consists of multiple conditions that are joined by logical operators:

```
where sales > 600000 and salary < 100000;
```

WHERE expression processing can improve efficiency of a request. For example, when you use a WHERE expression with a SAS index, then it is not necessary for SAS to read all rows in the data set in order to perform the request.

Summary of Statements for Conditional Processing in SAS

Table 20.1 Conditionally Selecting Data Using SAS

Language Element	Description	Example

CASE WHEN expression in PROC SQL	selects result values that satisfy specified conditions.	`proc sql;`
		`   select Title, length,`
		`   case`
		`      when length<120 then 'Short'`
		`      when length>=120 then 'Long'`
		`         else 'No Length'`
		`   end as m_length from movies;`
		`quit;`

DO loop	executes statements between a DO and an END statement repetitively.,	`data simple_do;`
		`   years=5;`
		`   if years>0 then`
		`   do;`
		`      months=years*12;`
		`      output;`
		`   end;`

IF statement in the DATA step	conditionally executes statements based on whether a condition is true.	`data if_ex;`
		`   set sashelp.class;`
		`   if age>14;`
		`   output;`
		`run;`

SELECT WHEN statement in the DATA step	conditionally executes	`data class;`
		`   set sashelp.class;`
		`   select (Name);`
		`     when ('Philip');`

Language Element	Description	Example
	statements when a condition is true.	otherwise delete; end; run;
WHERE statement in the DATA step	conditionally selects rows so that SAS processes only the rows that meet specified conditions.	data class; set sashelp.class; where Age>14; run;

WHERE statement in the PROC step	conditionally selects rows so that the SAS procedure processes only the rows that meet specified conditions.
	More information
		proc print data=sashelp.class;
		where age > 14;
		run;
		
		proc means data=prdsale;
		where region = 'EAST';
		run;

WHERE= data set option in the DATA step.	conditionally selects rows to read in from an input data set or to write out to an output data set.	data class(where=(age>14));
		set sashelp.class;
		run;
		
		data class;
		set sashelp.class(where=(age>14));
		run;

WHERE= data set option in the PROC step	conditionally selects rows so that SAS prints only the rows that meet specified conditions. More information	proc freq data=sashelp.cars(where=(weight>6000)); tables make * model; run; quit;
CASE expression in the PROC SQL step	conditionally selects rows that meet specified conditions.	proc sql; select Name, case when Continent = 'North America' then 'Group A' else 'Need to assign' end as Region from states; run; quit;
WHERE clause in the PROC SQL step	conditionally selects rows that meet specified conditions.	proc sql; select state from crime where murder > 7; run; quit;
DATA step view, PROC SQL view, and SAS/ACCESS that is stored with the definition	from a view, conditionally selects rows that meet specified conditions.	proc sql; create view stat as select * from crime where murder > 7; run; quit;

Language Element	Description	Example
	More information	

Table 20.2 Base SAS Procedures That Support WHERE Expression Processing

APPEND procedure	REPORT procedure
COMPARE procedure	SORT procedure
DATASETS procedure	STANDARD procedure
FCMP procedure	SUMMARY procedure
MEANS procedure	TABULATE procedure
PRINT procedure	TRANSPOSE procedure
RANK procedure	

DO Loops

Types of DO Loops

In programming, you might need to execute the same set of statements repeatedly or execute the same set of statements for a specific number of times. This type of processing requires the use of loops. In SAS, you create loops by specifying the DO statement. There are four basic DO loops used in SAS programming:

Table 20.3 Types of DO Loops

Type	Description	Example

DO statement	■ executes statements between the DO and END statements as a unit.
	■ often used with IF-THEN/ELSE statements, where the statements in the DO group are executed depending on whether the value of the IF statement condition is true or false.
		`data example;`
		`   x=1;`
		`   if x>0 then`
		`      do;`
		`         x=x*10;`
		`         put x=;`
		`      end;`
		`run;`

<table>
<tr><th>Type</th><th>Description</th><th>Example</th></tr>
<tr><td></td><td></td><td>Example: Use a Simple Iterative DO Loop</td></tr>
<tr><td>Iterative DO statement</td><td>■ executes statements between the DO and END statements repetitively, based on the value of an index variable.
■ There are different forms of the iterative DO statement.</td><td><pre>data example;
   x=0;
   do i=1 to 3;
      x=x+1;
      put x=;
   end;
run;</pre>Log output<pre>x=1 i=1
x=2 i=2
x=3 i=3</pre>Example: Use Iterative DO TO Syntax to Iterate a Specific Number of Times</td></tr>
<tr><td>Conditional DO WHILE statement</td><td>■ executes statements in the DO loop repetitively while a condition is true
■ evaluates the condition at the top of the loop so that the DO WHILE loop never executes if the condition is false.</td><td><pre>data do_while;
x=7;
   do while(x < 10);
      x=x+1;
      put x=;
   end;
run;</pre>Log output<pre>x=8
x=9
x=10</pre></td></tr>
<tr><td>Conditional DO UNTIL statement</td><td>■ executes statements in the DO loop repetitively until a condition is true
■ evaluates the condition at the bottom of the loop so that the DO UNTIL loop always executes at least once, even if the condition is false.</td><td><pre>data do_until;
 x=7;
   do until(x > 10);
      x=x+1;
      put x=;
   end;
run;</pre>Log output<pre>x=8
x=9
x=10
x=11</pre>Example: Use the Iterative DO WHILE and DO UNTIL Statements to Execute a Group of Statements Repetitively</td></tr>
</table>

Forms of the Iterative DO Loop

As listed in the previous table, one of the forms of the DO loop in SAS is the iterative DO loop. There are several forms of the iterative DO loop:

Table 20.4 Forms of the Iterative DO Statement

Syntax	Description	Example

DO *index-variable* = *start-value*;	Simple **DO** loop syntax
	The start-value is the initial value of the index-variable and it is a number or an expression that evaluates to a number.
		`data simple_do;`
		`x = 5;`
		`do i= x*2;`
		`end;`
		`run;`
		Example: Use a Simple Iterative DO Loop

DO *index-variable* = *start-value* **TO** *stop-value*;	**DO** *start-value* **TO** *stop-value* syntax
	The *start-value* and *stop-value* are numbers or expressions that evaluate to numbers.
		`data do_to;`
		`x = 5;`
		`do i=(x+1) to (x*2);`
		`x=x+1;`
		`output;`
		`end;`
		`run;`
		Example: Use Iterative DO TO Syntax to Iterate a Specific Number of Times

DO *index-variable* = *start-value* **TO** *stop-value***BY** *increment-value*;	**BY**-*increment* syntax
	The *start-value* and the *stop-value* are numbers or expressions that evaluate to numbers.
		`data do_to_by;`
		`x = 1;`
		`do i=0 to 10 by (x*2);`
		`x=x+1;`
		`output;`
		`end;`
		`run;`
		Example: Use the Iterative DO TO BY Syntax to Iterate a Specific Number of Times by a Specific Interval

DO *index-variable* = *start-value-1*,< *start-value-2*>, <*start-value-n*>;	**DOLIST** syntax
	The *start-value* is a list of values separated by commas. The values in the list can be individual numbers or expressions that evaluate to numbers.
		`data do_list;`
		`do i=5, 10, 20+1;`
		`output;`
		`end;`
		`run;`
		Example: Use DOLIST Syntax to Iterate over a List of Values

WHERE Expressions

WHERE Statement

You can use the WHERE statement in SAS to select rows from a SAS data set that meet a particular condition.

```
data cars;
   set sashelp.cars;
   where weight>6000;
run;
```

For complete syntax information, see “WHERE Statement” in *SAS DATA Step Statements: Reference*.

Note: By default, a WHERE expression does not evaluate added and modified rows. 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*.

See Also

“Example: Conditionally Select Rows Using the WHERE Statement” on page 457

Variables in WHERE Expressions

Variable Types and WHERE Expressions

A variable is a name that refers to a column of data in a SAS data set. Each SAS variable has attributes, such as its name and its data type. The variable’s data type determines how operations are performed in the WHERE expression. For example, the WHERE expression in the DATA step below selects rows in which `sales` are greater than **550**. Because `sales` is a numeric variable, the comparison operator (greater than) compares numeric values in the WHERE expression.

```
data retail;
   set sashelp.retail;
   where sales > 550;
run;
```

This example creates a SAS data set that selects only rows in which the date is later than January 1, 1953. This value is an example of a SAS date constant and it is also used with the comparison operator, greater than (>) to perform a comparison of date values.

```
data air;
   set sashelp.air;
   where date>'01jan1953'd;
run;
```

Automatic DATA Step Variables and WHERE Expressions

You cannot use automatic DATA step variables in WHERE expressions. For example, you cannot use the FIRST.variable, LAST.variable, _N_ variable, or variables created in assignment statements in WHERE expressions. Variables in the WHERE expression must exist in the source data set that is being read or created.

Stand-Alone Variables and WHERE Expressions

As with other SAS expressions, the names of numeric variables can stand alone. SAS treats numeric values of 0 or missing as false and all other values as true.

For example, in the following DATA step, the WHERE expression returns all rows from the `sashelp.heart` data set in which the numeric values for `AgeCHDdiag` are not missing or zero and the character values for `DeathCause` are not blank or missing:

```
data heart;
   set sashelp.heart;
   where AgeCHDdiag and DeathCause;
run;
```

Functions in WHERE Expressions

A SAS function returns a value from a computation or system manipulation. You can use a SAS function in a WHERE expression.

The following DATA step uses the SUBSTR function to produce a SAS data set that contains only rows in which `name` begins with **`Jan`**:

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

data J_class;
   set class;
   where substr (name,1,3) = 'Jan';
run;
proc print data=J_class;
```

```
run;
```

Output 20.1 *PROC PRINT Output Showing the SUBSTR Function Used in a WHERE Statement*

Obs	Name	Sex	Age	Height	Weight
1	Jane	F	12	59.8	84.5
2	Janet	F	15	62.5	112.5

The following SAS functions are commonly used with WHERE expressions:

- SUBSTRN function to extract a substring.
- TODAY function to return the current date.
- PUT function returns a given value using a given format.

IMPORTANT You cannot use WHERE expressions with functions that contain the OF operator.

OF syntax is permitted in some SAS functions. For example, in the following DATA step, OF is used with the RANGE function to return the difference between the largest and smallest values in the list **d1-d3**. This is permitted as long as the function containing the OF operator is not specified in a WHERE expression.

```
data test;
   n1=1;
   n2=2;
   n3=3;
   num = range(of n1-n3);
run;
```

If you specify this same function with OF in a WHERE expression, SAS returns an error to the log:

```
ERROR: Syntax error while parsing WHERE clause.
```

Instead of using an OF in a WHERE clause, you can enumerate all the variables from the `OF` list:

```
where num < range(n1,n2,n3);
```

Note: You can use the TRIM function and the SUBSTR function in a WHERE expression to improve performance on indexed data. For more information, see "Indexes and WHERE Expressions" on page 442.

See Also

- "The OF Operator with Functions and Variable Lists" on page 100
- *SAS Functions and CALL Routines: Reference*

Constants in WHERE Expressions

A constant is a fixed value such as a number or quoted character string. You can specify SAS constants in WHERE expressions to find a particular value in a SAS data set. The constant values are created within the WHERE expression itself. Constants are also called literals.

A constant can be a numeric value, a character string, or a datetime value. The following table lists the different types of constants:

Table 20.5 SAS Constants That Can Be Specified in WHERE Expressions

Constant Type	Description	Example
Numeric constants	do not enclose the value in quotation marks.	`where price > 200;`

Character constants	enclose the value in either single or double quotation marks.
	ensure that quoted values are exact matches, including case.
	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 lastname eq 'Martin';`
		`where item = '6" decorative pot';`
		`where name = "D'Amico";`

"Date, Time, and Datetime Constants"	enclose the value in either single or double quotation marks.
	after the closing quotation mark, specify the letter **d** for date constants, the letter **t** for time constants, or the letters **dt** for datetime constants.
		`where birthday = '24sep1975'd;`
		`where time>'9:25:19pm't`
		`where year='2018-05-17T09:15:30'dt;`

Operators in WHERE Expressions

Arithmetic Operators

Arithmetic operators enable you to perform a mathematical operation. The arithmetic operators include the following:

Table 20.6 Arithmetic Operators

Operator Symbol	Description	Example
*	multiplication	`where bonus = salary * .10;`
/	division	`where f = g / h;`
+	addition	`where c = a+b;`
-	subtraction	`where f = g-h;`
**	exponentiation	`where y = a**2;`

Comparison Operators

Comparison operators (also called binary operators) compare a variable with a value or 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 rows that have the value 78753 for the numeric variable `zipcode`:

```
where zipcode eq 78753;
```

The following table lists the comparison operators used in SAS:

Table 20.7 Comparison Operators

Symbol	Alias	Definition	Example
=	EQ	equal to	`where empnum eq 3374;`

Symbol	Alias	Definition	Example
^= or ~= or ¬= or <>	NE	not equal to	where status ne full-time;
>	GT	greater than	where hiredate gt '01jun1982'd;
<	LT	less than	where empnum < 2000;
>=	GE	greater than or equal to	where empnum >= 3374;
<=	LE	less than or equal to	where empnum <= 3374;
	IN	equal to one from a list of values	where state in ('NC','TX');

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 rows with names beginning with the letter **S**:

```
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.

Logical Operators

You can combine or modify WHERE expressions by using the Boolean logical 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*

The logical operators and their equivalent symbols are shown in the following table:

Table 20.8 Logical (Boolean) Operators

Symbol	Mnemonic Equivalent	Description	Example
&	AND	combines two or more expressions and finds rows that satisfy all conditions.	where skill eq 'java' and years eq 4;

Symbol	Mnemonic Equivalent	Description	Example
! or | or ¦	OR	combines two or more expressions and finds rows that satisfy either of the conditions or all of them.	`where skill eq 'java' or years eq 4;`
^ or ~ or ¬	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.	`where skill not eq 'java' or years not eq 4;`

Order of Evaluation for 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 the following order:

1. **NOT** – processed first.
2. **AND** – processed next.
3. **OR** – processed last.

Using Parentheses to Control the Order of Evaluation

You can control the order of evaluation of a SAS expression by placing the expression in parentheses. The expression within the innermost set of parentheses is processed first, followed by the next, outer parentheses, moving outward until all expressions in parentheses have been processed.

For example, suppose that you want a list of all the Canadian sites that have either SAS/GRAPH or SAS/STAT software. You issue the following expression without using parentheses:

```
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';
```

IN Operator

The IN operator is also a comparison operator. It searches for character and numeric values that are equal to one of the values 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 that you want all sites that are in North Carolina or Texas. You could specify:

```
where state = 'NC' or state = 'TX';
```

However, it is easier to use the IN operator, which selects any state in the list:

```
where state in ('NC','TX');
```

In addition, you can use the NOT logical operator to exclude a list:

```
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:

- `y = x in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);`
- `y = x in (1:10);`

See Also

"Types of Variable Lists" on page 99

Interval Comparisons

An interval comparison (or, 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. In this case, since the comparison operator, less than (<) is accompanied by an equal sign, this comparison is considered an inclusive interval:

```
where 500 <= empnum <= 1000;
```

Note that the previous range condition expression is equivalent to the following:

```
where empnum >= 500 and empnum <= 1000;
```

You can combine the NOT logical operator with a fully bounded range condition to select rows that fall outside the range. Note that parentheses are required:

```
where not (500 <= empnum <= 1000);
```

BETWEEN-AND Operator

The BETWEEN-AND operator is also considered a fully bounded range condition that selects rows 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:

WHERE variable BETWEEN value AND value;

For example:

```
where empnum between 500 and 1000;
where taxes between salary*0.30 and salary*0.50;
```

You can combine the NOT logical operator with the BETWEEN-AND operator to select rows that fall outside the range:

```
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:

```
where 500 <= empnum <= 1000;
where empnum between 500 and 1000;
```

CONTAINS Operator

The most common usage of the CONTAINS (?) operator is to select rows 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 rows that have the values **Mobay** and **Brisbayne** for the variable Company, but they do not select rows that contain **Bayview**:

```
where company contains 'bay';
where company ? 'bay';
```

You can combine the NOT logical operator with the CONTAINS operator to select rows that are not included in a specified string:

```
where company not contains 'bay';
```

You can also use the CONTAINS operator with two variables 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 rows in which the value of a variable is missing. The operator selects rows 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 rows by pattern matching; that is, it selects rows by comparing the values of a character variable to a specified pattern.

The LIKE operator is case sensitive and it uses two special characters for specifying a pattern:

percent sign (%)

specifies that any number of characters can occupy that position. For example, the following WHERE expression selects all employees with a name that starts with the letter **N**. The names can be of any length.

```
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 20.9 Names Selected by Using the LIKE Operator

Pattern	Name Selected
`where frstname like 'D_an'`	Dyan
`where frstname like 'D_an_'`	Diana, Diane
`where frstname like 'D_an__'`	Diana
`where frstname like 'D_an%'`	all names from list

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:

```
where frstname not like 'D_an%';
```

The % and _ characters have a special meaning for the LIKE operator. When searching for patterns that contain the % and _ characters, you must use an escape character.

An escape character is a single character that, in a sequence of characters, signifies that which 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 selects the values **a_b** and **axb**. The 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 escape character in the pattern-matching expression, and then the keyword ESCAPE followed by the escape-character expression.

```
LIKE 'pattern-matching-expression' ESCAPE 'escape-character-expression'
```

```
WHERE x LIKE a/_b ESCAPE '/';
```

When including an escape character with the LIKE operator in a WHERE expression, note the following points:

- the pattern-matching expression must be a character string that is enclosed in quotation marks.
- the pattern-matching expression cannot contain a column name.
- the escape-character expression must be a character string that evaluates to a single character.
- if the escape-character expression is a single character, enclose it in quotation marks.

Sounds-like Operator

The sounds-like (=*) operator selects rows 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: 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 rows 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. Here is an example of what you cannot do:

```
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 with a WHERE expression when 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 rows 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 the 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:

```
where name = 'John'||'Smith';
```

Prefix Operators

The plus sign (+) and the 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.

```
where z = -(x + y);
```

Note: The NOT operator is also considered a prefix operator.

Guidelines for Writing Efficient WHERE Expressions

In addition to creating indexes for the data set, here are some guidelines for writing efficient WHERE expressions:

Table 20.10 Constructing Efficient WHERE Expressions

Guideline	Efficient	Inefficient
Avoid using the LIKE operator that begins with % or _.	`where country like 'A%INA';`	`where country like '%INA';`
Avoid using arithmetic expressions that contain only constants.	`where salary > 48000;`	`where salary > 12*4000;`

Indexes and WHERE Expressions

An index is an optional file that provides direct access to specific rows without SAS having to read every row sequentially. An index enables SAS to locate an observation by value. Indexing a SAS data set can significantly improve the performance of WHERE processing.

For example, the following DATA step creates an index on the `sashelp.prdsal2` data set that is optimized for selecting rows based on the values of the variable `Actual`:

```
data prdIndx(index=(Actual));
   set sashelp.prdsal2;
run;
proc print data=myindex(where=(Actual<10));
run;
```

Assume that the data set `sashelp.prdsal2`, which has over 23,000 rows, is sorted by the variable `Actual`, without an index. To process the expression **`where Actual<10`**, SAS starts reading rows sequentially, beginning with the first row in the data set, and stops reading rows only after it finds a row that is greater than 10. SAS can determine when to stop reading rows, but without an index, there is no indication of where to begin. So, SAS begins with the first row. This can require reading a lot of rows.

Having an index enables SAS to determine which rows satisfy the criteria. This is called optimizing the WHERE expression.

Note: However, by default, SAS decides whether to use the index or to read the entire data set sequentially.

For information about SAS indexes, see "Using an Index for WHERE Processing" in *SAS V9 LIBNAME Engine: Reference*.

Data Set and System Options with WHERE Expressions

When you conditionally select a subset of observations with a WHERE expression, you can further subset the data by specifying the FIRSTOBS= and OBS= data set options or the FIRSTOBS= and OBS= system options.

The following example uses the FIRSTOBS= and OBS= data set options in the PROC PRINT statement to print a subset of the data returned by the WHERE statement.

```
data shoes;
   set sashelp.shoes;
   keep Product Sales;
run;
proc print data=shoes(firstobs=2 obs=5);
   title 'Subset of Shoes WHERE Sales<800';
   title2 'Print rows 2 - 5';
   where Sales<800;
run;
```

Subset of Shoes WHERE Sales<800
Print rows 2 - 5

Obs	Product	Sales
176	Sport Shoe	$449
197	Sandal	$325
223	Sport Shoe	$450
297	Sandal	$601

Note: Notice that the values for the data set options are 2 and 5 and not 176 and 297. This is because the PRINT output preserves the physical numbers from the original `Shoes` input data set. The data set option numbers represent the logical row numbers from the filtered data set and not the physical row numbers.

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 might 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 rows for each SAS view. The result might be that no rows meet the criteria.

See Also

"Example: Data Set Options" on page 469

IF Statements

IF statements execute code only if a condition is satisfied. The subsetting IF and the WHERE statements both test a condition to determine whether SAS should process a row. However, the two statements are not equivalent.

For a comparison of these two statements, see "Subsetting IF Statement versus the WHERE Statement" on page 444.

Types of IF Statements

There are two types of IF statements in SAS:

Table 20.11 Types of IF Statements

Statement Type	Description	Example

Subsetting IF statement	Continues processing only those rows that meet the condition of the specified expression.	`data if_example;`
		`   set sashelp.class;`
		`   if age>14 and weight<115;`
		`   output;`
		`run;`

IF-THEN/ELSE statement	Executes a SAS statement for rows that meet specific conditions.	`data if_then;`
		`   set sashelp.class;`
		`   if age>14 and weight<115 then output;`
		`   else delete;`
		`run;`

Subsetting IF Statement versus the WHERE Statement

To conditionally select rows from a SAS data set, you can use either a WHERE expression or a subsetting IF statement. Often, you can use either one of the statements to perform the same task. However, for some tasks, a specific

statement is required. The following table provides tasks that require you to use either one of the statements.

Table 20.12 Tasks Requiring Either a WHERE Expression or a Subsetting IF Statement

Task	Statement to Use
Make the selection in a procedure without using a preceding DATA step.	WHERE statement
Take advantage of the efficiency available with an indexed data set.	WHERE statement
Use one of a group of special operators, such as BETWEEN-AND, CONTAINS, IS MISSING or IS NULL, LIKE, SAME-AND, and Sounds-Like.	WHERE statement
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.	subsetting IF statement
Make the selection at some point during a DATA step rather than at the beginning.	subsetting IF statement
Execute the selection conditionally	subsetting IF statement

Table 20.13 Comparison of Behavior between IF Statements and WHERE Expressions

WHERE expressions	Subsetting IF statements
WHERE expressions can be used in a DATA step, a SAS procedure, or as a data set option.	Subsetting IF statements can be used only in a DATA step.
WHERE expressions test the condition *before* a row is read into the Program Data Vector.	A subsetting IF statement tests the condition *after* a row is read into the Program Data Vector.
If the condition is true, SAS reads the row into the PDV and processes the row. If the condition is false, SAS does not read the row into the PDV and continues processing with the next row.	If the condition is true, SAS processes the row. If the condition is false, SAS removes the row from the PDV and continues processing with the next row.

WHERE expressions	Subsetting IF statements
Testing the condition *before* the row is read can yield substantial savings when rows contain many variables or very long character variables (up to 32K bytes).	Not testing the condition *before* the row is read can be expensive, especially when rows contain many variables or very long character variables. If the data set contains rows with very few variables or short character variables, moving data to the PDV is likely to be fast. However, a variable containing 32K bytes of character data takes longer to move even though the data is contained in only one variable. In this case, a WHERE expression is generally more efficient because it avoids reading unnecessary data to the PDV.
WHERE expressions can be optimized with an index.	
WHERE expressions can be used with operators, such as LIKE and CONTAINS.	

See Also

- “Examples: IF Statements” on page 464
- “Other Control Flow Statements” on page 447
- “Creating the Input Buffer and the Program Buffer” on page 867.

SELECT WHEN Statement

The SELECT statement executes one of several statements or groups of statements. The following DATA step uses a SELECT statement to select only those rows in `sashelp.holiday` for which the `holiday` name contains the specified values:

```
data holiday;
  set sashelp.holiday;
   keep Name Cat;
   select (Name);
      when ('CHRISTMAS')  Cat=1;
      when ('MEMORIAL')   Cat=2;
      when ('MLK')        Cat=3;
```

```
    otherwise   delete;
end;
run;
proc print;
```

Obs	name	Cat
1	CHRISTMAS	1
2	CHRISTMAS	1
3	MLK	3
4	MEMORIAL	2

See Also

"Example: SELECT WHEN Statement" on page 467

Other Control Flow Statements

In addition to the SAS language elements specified in "Summary of Statements for Conditional Processing in SAS" on page 423, the following statements also control the flow of program execution.

Table 20.14 *Other SAS Control Flow Statements*

Statement	Description
ABORT statement	stops the execution of the current DATA step and resumes processing of the next DATA or PROC step. It can also stop executing a SAS program altogether, depending on the options that are specified in the ABORT statement and on the method of operation.
CONTINUE statement	stops the execution the current DO-loop iteration (based on a condition) and resumes execution of the next iteration of the DO-loop based on a condition.

DELETE statement	stops the execution of the current row (deletes the row) and resumes execution of the next iteration of the DATA step.
EOF= option in the INFILE statement	directs the execution of the SAS program immediately to the statements that follow the label specified in EOF= when the end of the input file is reached.
END statement	stops the execution of DO group or SELECT group processing.
ENDSAS statement	stops the execution of the SAS program and terminates the SAS session as soon as the statement is encountered in a SAS program.
GOTO statement	directs the execution of the SAS program immediately to the statement label that is specified in the GO TO statement. If followed by a RETURN statement, GO TO returns execution to the beginning of the DATA step.
HEADER= option in the FILE statement	whenever a PUT statement causes a new page of output to begin, executes statements that follow the label specified in the HEADER= option until a RETURN statement is encountered. When a RETURN statement is encountered, SAS returns control to the point from which the HEADER= option was activated.
LEAVE statement	stops the execution of the current DO-loop or SELECT group and resumes with the next statement in the DATA step that is outside of the DO-loop or SELECT group.

LINK statement	directs the execution of the SAS program immediately to the statement label that is specified in the LINK statement. If followed by a RETURN statement, returns execution to the statement that follows the LINK statement.
RETURN statement	stops the execution of statements at a specific point in the DATA step and returns to a predetermined point in the step.
STOP statement	stops the execution of the current DATA step and resumes processing of any statements that follow the current DATA step.

Examples: DO Loops

Example: Use a Simple Iterative DO Loop

Example Code

This example uses an iterative DO statement to repeatedly decrement the values for the variable `balance` and write these values to the output data set.

```
data loan;
   balance=10000;
   do payment_number=1 to 10;
      balance=balance-1000;
      output;
   end;
run;
proc print data=loan;
run;
```

Output 20.2 *PROC PRINT Output Showing Balance Decrease in a SAS Data Set Using an Iterative DO Loop*

Obs	balance	payment_number
1	9000	1
2	8000	2
3	7000	3
4	6000	4
5	5000	5
6	4000	6
7	3000	7
8	2000	8
9	1000	9
10	0	10

Key Ideas

- The iterative DO statement enables you to execute a group of statements between DO and END repetitively based on the value of an index variable.
- There are multiple forms of the iterative DO loop.
- The DO UNTIL and DO WHILE statements enable you to conditionally select data from a SAS data set.

See Also

"DO Statement: Iterative" in *SAS DATA Step Statements: Reference*

Example: Use a Nested Iterative DO Loop

Example Code

The following example shows how you can place one DO loop within another to compute the value of a one-year investment that earns 7.5% annual interest, compounded monthly.

```
data earn;
   Capital=2000;
   do Year=1 to 10;
      do Month=1 to 12;
         Interest=Capital*(.075/12);
         Capital+Interest;
         output;
      end;
```

```
   end;
run;
proc print data=earn; run;
```

Output 20.3 *Partial PROC PRINT Output Showing Compound Interest Earned Using Nested DO Loops*

Obs	Capital	Year	Month	Interest
1	2008.33	1	1	8.3333
2	2016.70	1	2	8.3681
3	2025.10	1	3	8.4029
4	2033.54	1	4	8.4379
5	2042.02	1	5	8.4731
6	2050.52	1	6	8.5084
7	2059.07	1	7	8.5438
8	2067.65	1	8	8.5794
9	2076.26	1	9	8.6152
10	2084.91	1	10	8.6511
11	2093.60	1	11	8.6871
12	2102.32	1	12	8.7233
13	2111.08	2	1	8.7597
14	2119.88	2	2	8.7962
15	2128.71	2	3	8.8328
16	2137.58	2	4	8.8696
17	2146.49	2	5	8.9066
18	2155.43	2	6	8.9437
19	2164.41	2	7	8.9810
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

Key Ideas

- Iterative DO statements can be executed within a DO loop. Putting a DO loop within a DO loop is called nesting.
- Nested loops are useful when, for each pass through the data in the outer loop, you need to repeat some action on the data in the inner loop.

  For example, you might need to read a file line-by-line and count how many times a particular word appears in each line. The outer loop reads the lines and the inner loop searches for the word by looping through the words in the line.
- The iterative DO statement enables you to execute a group of statements between DO and END repetitively based on the value of an index variable.
- There are multiple forms of the iterative DO loop.

See Also

"DO Statement: Iterative" in *SAS DATA Step Statements: Reference*

Example: Use DOLIST Syntax to Iterate over a List of Values

Example Code

This example uses DOLIST syntax to iterate over a list of values.

```
data do_list;
  x=-5;
  do i=5,               /*a single value*/
       5, 4,            /*multiple values*/
       x + 10,          /*an expression*/
       80 to 90 by 5,   /*a sequence*/
       60 to 40 by x;   /*a sequence with a variable*/
    output;
  end;
run;
proc print data=do_list; run;
```

Output 20.4 *PROC PRINT Output Showing DOLIST Syntax Output*

Obs	x	i
1	-5	5
2	-5	5
3	-5	4
4	-5	5
5	-5	80
6	-5	85
7	-5	90
8	-5	60
9	-5	55
10	-5	50
11	-5	45
12	-5	40

Key Ideas

- A DOLIST is a type of syntax that is based on the iterative DO loop, in which the loop iterates over the elements of a list. The list serves as the start-value in the DO statement and the items in the list are separated by commas.

- The iterative DO statement enables you to execute a group of statements between DO and END repetitively based on the value of an index variable.

See Also

- "Types of DO Loops" on page 425.
- "DO Statement: Iterative" in *SAS DATA Step Statements: Reference*

Example: Use Iterative DO TO Syntax to Iterate a Specific Number of Times

Example Code

This example uses the iterative DO TO-*value* syntax to repeatedly increment the values of x. The loop also specifies the OUTPUT statement to write each incremented value of x to the output data set.

```
data do_to;
   x=0;
   do i=0 to 10;
      x=x+1;
      output;
   end;
run;
proc print data=do_to;
run;
```

Output 20.5 *PROC PRINT Output Showing Iterative DO TO-value Syntax*

Obs	x	i
1	1	0
2	2	1
3	3	2
4	4	3
5	5	4
6	6	5
7	7	6
8	8	7
9	9	8
10	10	9
11	11	10

Key Ideas

- With DO TO-*value* syntax, the TO value specifies the ending value for the index-variable in an iterative DO loop.

See Also

- "DO Statement: Iterative" in *SAS DATA Step Statements: Reference*
- "Using Various Forms of the Iterative DO Statement" in *SAS DATA Step Statements: Reference*

Example: Use the Iterative DO TO BY Syntax to Iterate a Specific Number of Times by a Specific Interval

Example Code

This example uses the iterative DO TO-*value* BY-*increment* syntax to repeatedly increment values of x by **1**. The BY-*increment* value specifies that the index variable increments by **2** each time that the loop executes.

The loop also specifies that the OUTPUT statement writes each incremented value of x to the output data set.

```
data do_to_by;
   x=0;
   do i=0 to 10 by 2;
      x=x+1;
      output;
   end;
run;
proc print data=do_to_by;
run;
```

Output 20.6 *PROC PRINT Output Showing Iterative DO TO BY Syntax*

Obs	x	i
1	1	0
2	2	2
3	3	4
4	4	6
5	5	8
6	6	10

Key Ideas

- With DO TO-*value* BY-*increment* syntax, the BY increment value specifies a positive or negative number (or an expression that yields a number) to control how the index-variable is incremented in an iterative DO loop.
- The iterative DO statement enables you to execute a group of statements between DO and END repetitively based on the value of an index variable.

See Also

- "DO Statement: Iterative" in *SAS DATA Step Statements: Reference*
- "Using Various Forms of the Iterative DO Statement" in *SAS DATA Step Statements: Reference*

Example: Use the Iterative DO WHILE and DO UNTIL Statements to Execute a Group of Statements Repetitively

Example Code

This example uses a DO loop to execute a group of statements repetitively *while* a condition is true. The program calculates the number of payments that must be made for a specified loan amount by iterating repeatedly to decrement `Balance` *while* the balance is greater than zero.

```
data loan;
   balance=10000;
   payment=0;
```

```
      do while (balance>0);
         balance=balance-1000;
         payment=payment+1;
         output;
      end;
   run;
   proc print data=loan;
   run;
```

Output 20.7 *PROC PRINT Output Showing Balance Decrease in a SAS Data Set Using a DO WHILE Statement*

Obs	balance	payment
1	9000	1
2	8000	2
3	7000	3
4	6000	4
5	5000	5
6	4000	6
7	3000	7
8	2000	8
9	1000	9
10	0	10

Key Ideas

- The DO UNTIL statement enables you to execute statements in a DO loop repetitively until a condition is true.
- The DO WHILE statement enables you to execute statements in a DO loop repetitively while a condition is true.

See Also

- "DO UNTIL Statement" in *SAS DATA Step Statements: Reference*
- "DO WHILE Statement" in *SAS DATA Step Statements: Reference*
- "DO Statement: Iterative" in *SAS DATA Step Statements: Reference*

Examples: WHERE Processing

Example: Conditionally Select Rows Using the WHERE Statement

Example Code

This example uses the WHERE statement to conditionally select rows from the `sashelp.class` data set and writes the selected rows to the SAS data set, `shoes`.

The first DATA step runs in SAS and subsets the data, selecting only the rows in which `Sales` are greater than **`$500,000`**. The filtered data is loaded to `shoes`.

The second DATA step runs on the `shoes` data set to further subset the data. This DATA step selects only the rows in which the values for the variable `Region` is **`Canada`**. The selected rows are then written to the output SAS data set `shoes2`.

```
data shoes;
   set sashelp.shoes;
   where Sales>=500000;
run;
proc print data=shoes;
   var Region Product Sales;
 run;

data shoes2;
   set shoes;
   where Region="Canada";
run;

proc print data=shoes2;
   var Region Product Sales;
run;
```

Output 20.8 *PROC Print Output for the shoes Table in Which Sales Are Greater Than $500,000*

Obs	Region	Product	Sales
1	Canada	Men's Dress	$757,798
2	Canada	Slipper	$700,513
3	Canada	Women's Dress	$756,347
4	Central America/Caribbean	Men's Casual	$576,112
5	Middle East	Men's Casual	$1,298,717
6	Western Europe	Women's Casual	$502,636

Output 20.9 *PROC Print Output for the shoes2 Table in Which Region Is Canada*

Obs	Region	Product	Sales
1	Canada	Men's Dress	$757,798
2	Canada	Slipper	$700,513
3	Canada	Women's Dress	$756,347

Key Ideas

- A WHERE expression is a type of SAS expression that defines a condition for selecting rows to be processed in a SAS data set.
- When the WHERE statement is specified in a DATA step, SAS does not have to read all of the rows from the input data set. Therefore, the WHERE statement can improve the performance of your SAS programs. Using the WHERE statement can improve the efficiency of your SAS programs because SAS is not required to read all rows from the input data set.
- You cannot use the WHERE statement as part of an IF-THEN statement.
- WHERE statements can contain multiple WHERE expressions that are joined by logical operators. These expressions are called compound WHERE expressions.

See Also

"WHERE Statement" in *SAS DATA Step Statements: Reference*

Example: Conditionally Select Rows Using a Compound WHERE Expression

Example Code

This example uses a compound WHERE expression to select a subset of data from the SAS data set `sashelp.shoes`.

The DATA step conditionally select rows in which `Sales` are greater than **`$500,000`** and `Region` is **`Canada`**. The compound WHERE expression consists of two WHERE expressions that are joined by the AND operator.

```
data shoes;
   set sashelp.shoes;
   where Sales>=500000 and Region="Canada";
   keep Region Product Sales;
run;

proc print data=shoes; run;
```

Output 20.10 *PROC PRINT Output for Conditionally Selecting Rows Using a Compound WHERE Expression*

Obs	Region	Product	Sales
1	Canada	Men's Dress	$757,798
2	Canada	Slipper	$700,513
3	Canada	Women's Dress	$756,347

Key Ideas

- A *compound WHERE expression* consists of multiple conditions joined by logical operators.
- When SAS encounters a WHERE expression that contains multiple conditions, it processes the conditions in the following order:
  1. NOT expressions first.
  2. AND expressions next.
  3. OR expressions last.

See Also

- "WHERE Statement" in *SAS DATA Step Statements: Reference*
- compound WHERE expression

Example: Conditionally Select Rows Based on Multiple Conditions

Example Code

In the following example, the AND operator is used in the WHERE statement to find rows based on conditions for `Age` and `Sex`.

```
data class;
   set sashelp.class;
   where sex="M" and age >= 15;
run;
proc print data=class;
run;
```

The output data set contains information about Males who are 15 years or older:

Output 20.11 *PROC PRINT Output for Conditionally Selecting Rows Based on Multiple Conditions*

Obs	Name	Sex	Age	Height	Weight
1	Philip	M	16	72.0	150
2	Ronald	M	15	67.0	133
3	William	M	15	66.5	112

In the following example, OR finds rows that satisfy either condition.

```
data class;
   set sashelp.class;
   where sex="M" or age>=15;
run;
proc print data=class;
   title 'OR finds all Males and Anyone 15 Years or Older';
run;
```

Output 20.12 PROC PRINT Output for Conditionally Selecting Rows Based on Multiple Conditions

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Henry	M	14	63.5	102.5
3	James	M	12	57.3	83.0
4	Janet	F	15	62.5	112.5
5	Jeffrey	M	13	62.5	84.0
6	John	M	12	59.0	99.5
7	Mary	F	15	66.5	112.0
8	Philip	M	16	72.0	150.0
9	Robert	M	12	64.8	128.0
10	Ronald	M	15	67.0	133.0
11	Thomas	M	11	57.5	85.0
12	William	M	15	66.5	112.0

In the following example, the less than symbol (<) finds rows that satisfy the criteria for age and sex.

```
data class;
   set sashelp.class;
   where age < 15 and sex NE "M";
run;
proc print data=class;
   title 'Finds Females
          Older less than 15 Years';
run;
```

Output 20.13 PROC PRINT Output for Conditionally Selecting Rows Based on Multiple Conditions

Females 15 Years Old and Younger

Obs	Name	Sex	Age	Height	Weight
1	Alice	F	13	56.5	84.0
2	Barbara	F	13	65.3	98.0
3	Carol	F	14	62.8	102.5
4	Jane	F	12	59.8	84.5
5	Joyce	F	11	51.3	50.5
6	Judy	F	14	64.3	90.0
7	Louise	F	12	56.3	77.0

The order in which SAS processes expressions that are joined by Boolean operators affects the output. The default order of operations is to process the NOT expressions first, the AND expressions next, and the OR expressions last:

```
data class;
   set sashelp.class;
   where age>15 or height<60 and sex="F";
run;
proc print data=class;
   title 'age > 15 OR height < 60 AND sex = F';
run;
```

Output 20.14 PROC PRINT Output for Conditionally Selecting Rows Based on Multiple Conditions Using Multiple Boolean Operators

Obs	Name	Sex	Age	Height	Weight
1	Alice	F	13	56.5	84.0
2	Jane	F	12	59.8	84.5
3	Joyce	F	11	51.3	50.5
4	Louise	F	12	56.3	77.0
5	Philip	M	16	72.0	150.0

To control the order of evaluation, you use parentheses. Here is the same example except parentheses are used to specify that the OR expression is evaluated first and then the AND expression.

```
data class;
   set sashelp.class;
   where (age>15 or height<60) and sex="F";
run;
proc print data=class;
   title '(age > 15 OR height < 60) AND sex = F';
run;
```

Output 20.15 PROC PRINT Output for Conditionally Selecting Rows and Controlling the Order of Operations

Obs	Name	Sex	Age	Height	Weight
1	Alice	F	13	56.5	84.0
2	Jane	F	12	59.8	84.5
3	Joyce	F	11	51.3	50.5
4	Louise	F	12	56.3	77.0

Key Ideas

- To conditionally select rows based on multiple conditions, use the Boolean operators AND, OR, and NOT.
- When SAS encounters a WHERE expression that contains multiple conditions, it processes the conditions in the following order:

  1. NOT expressions first.
  2. AND expressions next.
  3. OR expressions last.

Example: Conditionally Select Rows Using the WHERE= Data Set Option

Example Code

This example uses the WHERE= data set option to conditionally select rows from the `sashelp.shoes` data set.

```
data sales;
   set sashelp.shoes(where=(Region="Canada" and Sales<2000));
run;
proc print data=sales; run;
```

Obs	Region	Product	Subsidiary	Stores	Sales	Inventory	Returns
1	Canada	Sandal	Toronto	2	$1,190	$6,763	$38

Key Ideas

- The WHERE= data set option cannot be used with the POINT= option in the SET and MODIFY statements.

See Also

- "WHERE= Data Set Option" in *SAS Data Set Options: Reference*

Example: Use an Index to Improve Performance of WHERE Processing

Example Code

In the following example, the first DATA step creates an output data set, `mybaseball`, and the index= option adds a simple index to the `team` variable. The

second DATA step reads the data set and selects for processing only those rows in which the team name is **Atlanta**.

```
data mybaseball(index=(team));
   set sashelp.baseball;
run;

data mybaseball;
   set sashelp.baseball;
   where Team="Atlanta";
   keep Name Team Position;
run;
proc print data=mybaseball; run;
```

Key Ideas

- Indexing a SAS data set can significantly improve the performance of WHERE processing. An index is an optional file that you can create for a SAS data set in order to provide direct access to specific rows.
- To create indexes in a DATA step when you create the data set, use the INDEX= data set option.

See Also

"Indexes" in *SAS V9 LIBNAME Engine: Reference*

Examples: IF Statements

Example: Subset Data Using the Subsetting IF Statement

Example Code

This example shows how to use a subsetting IF statement to subset data from an input SAS data set and write out the rows to an output data set. The DATA step executes the SET statement on a row if the condition `Age=13` is true. Therefore, the program selects only the 3 rows in which `Age` equals **13**.

```
data class;
   set sashelp.class;
```

```
   if Age = 13;
run;
proc print data=class; run;
```

Output 20.16 PROC PRINT Output Showing How to Filter Data Using a Subsetting IF Statement

Obs	Name	Sex	Age	Height	Weight
1	Alice	F	13	56.5	84
2	Barbara	F	13	65.3	98
3	Jeffrey	M	13	62.5	84

The following DATA step executes the SET statement on the rows in which the value for the variable `Age` is missing. Since the input data does not contain any missing data for `Age`, SAS evaluates the IF statement to **`false`** and no rows are written to the output data set.

```
data class;
   set sashelp.class;
   if Age = .;
run;
proc print data=class; run;
```

Example Code 20.1 Log Output for Using the Subsetting IF Statement to Subset Data

```
82   proc print data=class; run;
NOTE: No observations in data set WORK.CLASS.
```

Key Ideas

- The subsetting IF statement filters data from an input SAS data source and writes out only the rows that satisfy a specified condition.
- If the expression is false (its value is 0 or missing), no further statements are processed for that row, the current row is not written to the output, and the remaining program statements in the DATA step are not executed.

See Also

- subsetting IF statement.
- "IF-THEN/ELSE Statement" in *SAS DATA Step Statements: Reference*

Example: Conditionally Select Specific Rows Using the IF Statement

Example Code

This example shows how to use an IF-THEN/ELSE statement to conditionally select rows in which `Region` is **`Canada`** and `Product` is **`Slipper`**.

```
data slipper;
   set sashelp.shoes;
   format commission comma6.;
   if Region = "Canada" and Product="Slipper"
      then commission = (Sales - Returns) * .10;
      else delete;
   keep Region Product Sales Returns Commission;
run;
proc print data=slipper;
```

Output 20.17 *PROC PRINT Output Showing How to Conditionally Select Data Using an IF-THEN/ELSE Statement*

Obs	Region	Product	Sales	Returns	commission
1	Canada	Slipper	$5,676	$253	542
2	Canada	Slipper	$135,305	$3,395	13,191
3	Canada	Slipper	$30,905	$1,397	2,951
4	Canada	Slipper	$80,352	$1,908	7,844
5	Canada	Slipper	$700,513	$21,247	67,927

Key Ideas

- The IF-THEN/ELSE statement executes a SAS statement for observations that meet specific conditions. An optional ELSE statement gives an alternative action if the THEN clause is not executed.
- SAS evaluates the expression in an IF-THEN statement to produce a result that is either nonzero, zero, or missing.
- A nonzero and nonmissing result causes the expression to be true; a result of zero or missing causes the expression to be false.
- Use a SELECT WHEN statement rather than a series of IF-THEN statements when you have a long series of mutually exclusive conditions.

- Use subsetting IF statements, without a THEN clause, to continue processing only those observations or records that meet the condition that is specified in the IF clause.

See Also

- subsetting IF statement
- "IF-THEN/ELSE Statement" in *SAS DATA Step Statements: Reference*
- "SELECT Statement" in *SAS DATA Step Statements: Reference*

Example: SELECT WHEN Statement

Example Code

This example shows how to use the DATA step SELECT statement to subset a SAS data set.

```
data shoes;
   set sashelp.shoes(where=(Region='Canada' or Region='Pacific' or
Region='Asia'));
   keep Region Sales Product;
   select(Region);
      when('Canada') sales = sales * .10;
      when('Pacific') sales = sales * .09;
      when('Asia') sales = sales * .07;
   end;
run;
proc print data=shoes; run;
```

Output 20.18 *Partial PROC PRINT Output Showing How to Use the SELECT Statement to Conditionally Select Rows from a SAS Data Set*

Obs	Region	Product	Sales
1	Asia	Boot	$140
2	Asia	Men's Dress	$212
3	Asia	Sandal	$226
4	Asia	Slipper	$211
5	Asia	Women's Casual	$377
6	Asia	Boot	$4,250
7	Asia	Men's Casual	$823
8	Asia	Men's Dress	$8,143
9	Asia	Sandal	$348
10	Asia	Slipper	$10,431
11	Asia	Sport Shoe	$86
12	Asia	Women's Casual	$1,431
13	Asia	Women's Dress	$5,476
14	Asia	Sport Shoe	$81
15	Canada	Boot	$1,772
16	Canada	Men's Dress	$1,278
17	Canada	Sandal	$289
18	Canada	Slipper	$568
19	Canada	Sport Shoe	$975
20	Canada	Women's Dress	$1,260
21	Canada	Boot	$4,021
2[illegible]	Cana[illegible]	Men's Casual	$5[illegible]93

Key Ideas

- The SELECT WHEN statement enables you to conditionally process one or more statements in a group of statements based on the value of a single categorical variable.
- A group of statements between the SELECT and END statements is called a SELECT group. SELECT groups contain WHEN statements that identify SAS statements that are executed when a particular condition is true.
- The END statement and at least one WHEN statement is required when specifying the SELECT statement.
- Using a SELECT group is more efficient than using a series of IF-THEN statements when you have a long series of mutually exclusive conditions.
- Large numbers of conditions make a SELECT group more efficient than IF-THEN/ELSE statements because CPU time is reduced. SELECT groups also make the program easier to read and debug.
- An optional OTHERWISE statement specifies a statement to be executed if no WHEN condition is met. For example, if your data contains missing or invalid data.

- In null statements that are used in an OTHERWISE statement, the SELECT WHEN statement prevents SAS from issuing an error message.

See Also

- "SELECT Statement" in *SAS DATA Step Statements: Reference*
- subsetting IF statement
- "IF-THEN/ELSE Statement" in *SAS DATA Step Statements: Reference*

Example: Data Set Options

Example Code

The following example shows how to use the FIRSTOBS= and OBS= data set options with the WHERE statement to specify a segment of data to process conditionally.

In this example, the DATA step creates a data set named `Example` that contains 10 rows and two variables: `i` and `x`.:

```
data example;
   do i=1 to 10;
      x=i + 1;
      output;
   end;
run;

proc print data=example; run;
```

Output 20.19 *PROC PRINT Output Showing the Data to Be Subset*

Obs	i	x
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11

The following PROC step contains two separate subsetting actions: the WHERE statement and the data set options FIRSTOBS= and OBS=. The WHERE statement is processed first on the original input data set `example`, and then the two data set options are processed on the results of the WHERE statement.

WHERE **`i > 5`** tells SAS to select only the rows in which `i` is greater than 5 from the original input data set. This is a subset of the original data set containing only 5 rows, which are rows 6 through 10 of the original input data set.

SAS then processes the data set options, `FIRSTOBS=2` and `OBS=4`, on the resulting 5 rows and prints the 2nd through 4th rows of the WHERE results. The PRINT procedure prints rows 7, 8, and 9 from the original input data set.

```
proc print data=example (firstobs=2 obs=4);
   where i > 5;
run;
```

Output 20.20 *PROC PRINT Selects Rows Where i>5 and Then from That Subset, Rows 2 through 4*

Obs	i	x
1	1	2
2	2	3
3	3	4
4	4	5
5	5	6
6	6	7
7	7	8
8	8	9
9	9	10
10	10	11

FIRSTOBS = 2

OBS = 4

i > 5

The result of PROC PRINT is rows 7, 8, and 9.

Output 20.21 *PROC PRINT Output Showing Data Set Example Containing Rows 7, 8, and 9*

Obs	i	x
7	7	8
8	8	9
9	9	10

Key Ideas

- FIRSTOBS= specifies the first observation that SAS processes in a SAS data set and OBS= specifies the last observation that SAS processes in a data set.
- When used in the DATA step, these options are valid for input (read) processing only; that is, they are valid in the SET statement and in the INFILE statement.
- 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.

See Also

- "FIRSTOBS= Data Set Option" in *SAS Data Set Options: Reference*
- "OBS= Data Set Option" in *SAS Data Set Options: Reference*

21

Combining Data

Overview of Combining Data **474**
- Definition of Combining Data 474
- Terms 474
- Data Relationships 476
- Summary of Ways to Combine SAS Data Sets 478
- Concatenating 479
- Interleaving 480
- Match-Merging 481
- One-to-One Merging 482
- One-to-One Reading 483
- Hash Table Merging 484
- Updating 485
- Modifying 487

Comparing Methods **488**
- PROC SQL versus Match-Merging 488
- Updating and Modifying Comparison 489
- One-to-One Reading versus One-to-One Merging 491
- Comparison of SAS Language Elements for Combining Data Sets 492

Examples: Prepare Data **495**
- Example: Examine the Data to Be Combined 495
- Example: Find the Common Variables in Multiple Input Data Sets 498
- Example: Creating Unique BY Values When Data Contains Duplicate BY Values 499
- Example: Prepare Data in Which Common Variables Have Different Data Types . 501
- Example: Prepare Data in Which Common Variables Have Different Lengths 504
- Example: Prepare Data in Which Common Variables Have Different Formats and Labels 506
- Example: Prepare Data in Which Common Variables Represent Different Data .. 509

Examples: Concatenate Data **510**
- Example: Concatenate Using the SET Statement 510
- Example: Concatenate Using PROC SQL 512
- Example: Append Files Using PROC APPEND 513
- Example: Add a Message to the Log When Variables Are Missing from Input Data Sets 515

Examples: Interleave Data **518**
- Example: Interleave Data Sets 518
- Example: Interleave with Duplicate Values of the BY Variable 520
- Example: Interleave with Different Values of the Common BY Variable 523

Examples: Match-Merge Data 525
Example: Match-Merge Observations Based on a BY Variable 525
Example: Match-Merge Observations with Common Variables That Are Not the BY Variables 527
Example: Match-Merge Observations with Duplicate Values of the BY Variable . 530
Example: Match-Merge Observations with Different Values of the BY Variable . . 532
Example: Match-Merge and Remove Unmatched Observations 534
Example: Match-Merge Data Sets with Duplicate Values in More Than One Data Set 536
Example: Match-Merge One-to-Many with Missing Values in Common Variables 539

Examples: Merge Data One-to-One 542
Example: Merge Data Sets with an Equal Number of Observations 542
Example: Merge Data Sets with an Unequal Number of Observations 544
Example: Merge Data Sets with Duplicate Values of Common Variables 546
Example: Merge Data Sets with Different Values for the Common Variables 548

Examples: Combine Data One-to-One 550
Example: Combine Data Sets That Contain an Equal Number of Observations . . . 550

Example: Merge Data Using a Hash Table 552
Example: Use a Hash Table to Merge Data Sets One-to-Many or Many-to-Many 552

Examples: Update Data 556
Example: Update Data Using the UPDATE Statement 556
Example: Update Data Sets with Duplicate Values of the BY Variable 558
Example: Update a Data Set with Missing and Different Values for the BY Variables 560

Example: Modify Data 563
Example: Modify a Data Set by Adding an Observation 563

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 323.

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 501 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 495 and "Example: Creating Unique BY Values When Data Contains Duplicate BY Values " on page 499 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

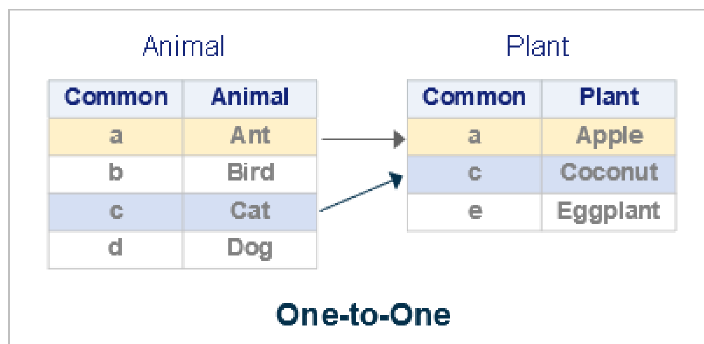
rows in input data sets that are not selected for output because the values do not match the specified criteria. The matching criteria can be based on row number (or position within the data set) or it can be based on the values of variables that you specify in a BY statement.

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

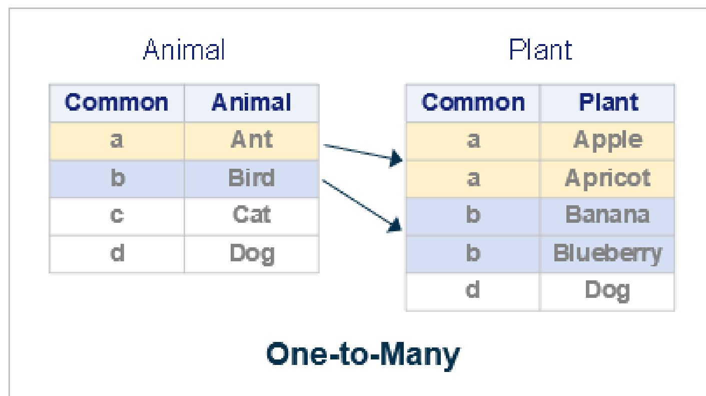
In a one-to-one relationship, a single row in one data set is related to only one row in a second data set, and a single row in the second data set is related to only one row in the first data set. The relationship is based on the values of one or more selected variables. A one-to-one relationship implies 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

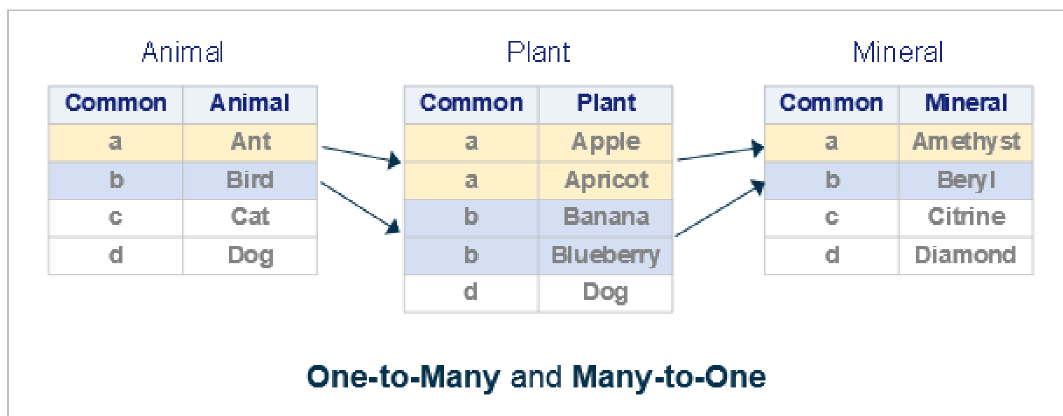
A one-to-many relationship between input data sets implies that one data set has at most one row with a specific value of the selected variable, but the other input data set can have more than one, or duplicates, of each value. When you work with multiple selected variables, this relationship implies 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

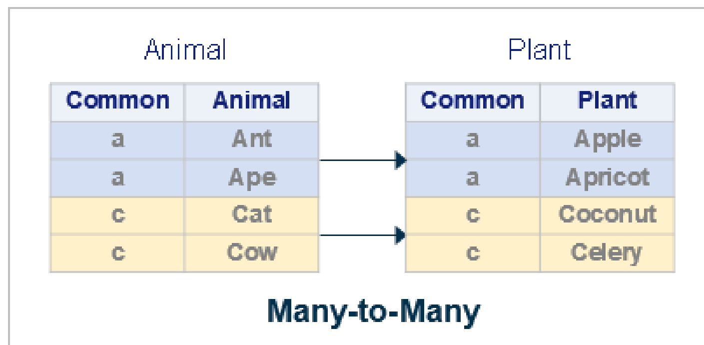
A one-to-many or many-to-one relationship between input data sets implies that one data set has at most one row with a specific value of the selected variable, but the other input data set can have more than one occurrence of each value. When you work with multiple selected variables, this relationship 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*

Type	Method	Statement or Procedure	Simplified Example Code

Concatenate	Concatenating Using the SET Statement	SET statement	`data concat;`
			`set data1 data2;`
			`run;`

	Concatenating using PROC SQL	SQL Procedure	`proc sql;`
			`select * from data1`
			`union all corresponding`
			`select * from data2;`
			`quit;`

	Appending using PROC APPEND	APPEND procedure	`proc append base=data1 data=data2;` `run;`

Interleave	Interleaving	SET statement with BY statement	`data interleav;`
			`set data1 data2; by year;`
			`run;`

Type	Method	Statement or Procedure	Simplified Example Code

Merge	Match-Merging	MERGE statement with BY statement	`data matchmerged;`
			`merge data1 data2; by year;`
			`run;`

	One-to-One Merging	MERGE statement	`data merged;`
			`merge data1 data2;`
			`run;`

	One-to-One Reading	SET statement	`data merged;`
			`set data1; set data2;`
			`run;`

	Hash Table Merging	HASH object with FIND method	`declare hash h(dataset:'data1');`
			`rc = h.definekey('key');`
			`rc = h.definedata('data1');`
			`h.definedone();`
			`set data2;`
			`h.find();`
			`run;`

	SQL procedure	PROC SQL Statement	`proc sql;`
			`select`
			`* from data1, data2; by year;`
			`where data1.key=data2.key;`
			`run;`

Update	Updating	UPDATE statement with BY statement	`data master;`
			`update master trans; by year;`
			`run;`

	Modifying	MODIFY statement with BY statement	`data master;`
			`modify master trans; by year;`
			`run;`

	Hash Table Updating	HASH object with REPLACE method	`declare hash h(dataset:'data1');`
			`rc = h.definekey('key');`
			`rc = h.definedata('data1');`
			`h.definedone();`
			`set data2;`
			`rc = h.find();`
			`if rc = 0 then h.replace();`
			`run;`

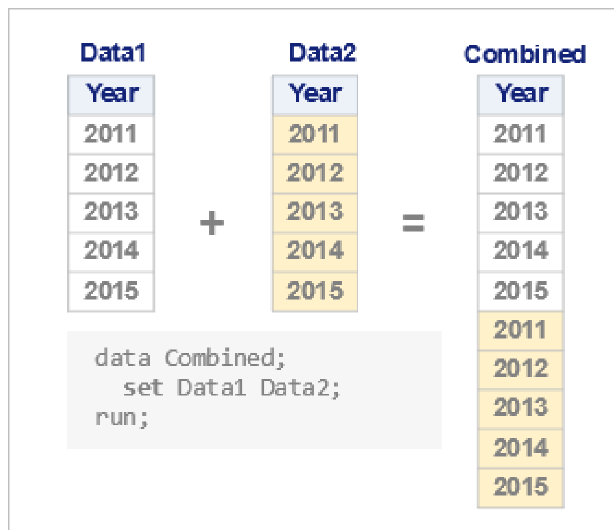
Concatenating

Definition *Concatenating Two Data Sets*

Concatenating combines two or more SAS data sets, one after the other, into a single, new output data set.

Syntax

SET <*data-set-1*> <*data-set-2*><… *data-set-n*>;



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 510
- "Concatenating SAS Data Sets" in *SAS DATA Step Statements: Reference*

Interleaving

Definition

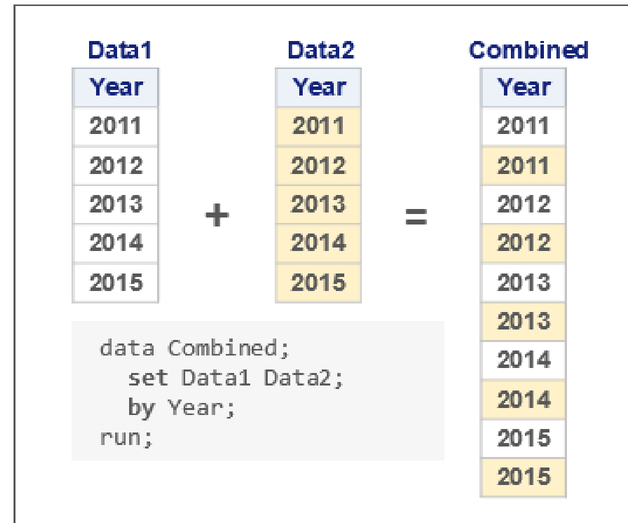
Interleaving combines two or more SAS data sets into a single data set by interspersing their rows with one another based on the values of common BY variables. In some programming languages, the term *merge* means to interleave rows. However, rows that are interleaved in SAS data sets are not merged; they are copied from the original data sets in the order of the values of the BY variable.

Syntax

Interleaving Two Data Sets

SET <*data-set-1*> <*data-set-2*><…*data-set-n*>;

BY *variable-1* <…*variable-n*>;



In the figure, rows from `Data2` are interspersed between rows from `Data1`.

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 518
- “Interleaving SAS Data Sets” in *SAS DATA Step Statements: Reference*

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*>;

Match–Merging Two Data Sets

Data1

Year	X
2011	x1
2012	x2
2013	x3
2014	x4
2015	x5

+

Data2

Year	Y
2011	y1
2012	y2
2013	y3
2014	y4
2015	y5

=

Combined

Year	X	Y
2011	x1	y1
2012	x2	y2
2013	x3	y3
2014	x4	y4
2015	x5	y5

```
data Combined;
 merge Data1 Data2;
 by Year;
run;
```

In the figure, rows from `Data2` are merged with rows from `Data1` based on the values of the BY variable, `Year`.

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 489

See Also

- "Examples: Match-Merge Data" on page 525
- "Match-Merging" in *SAS DATA Step Statements: Reference*

One-to-One Merging

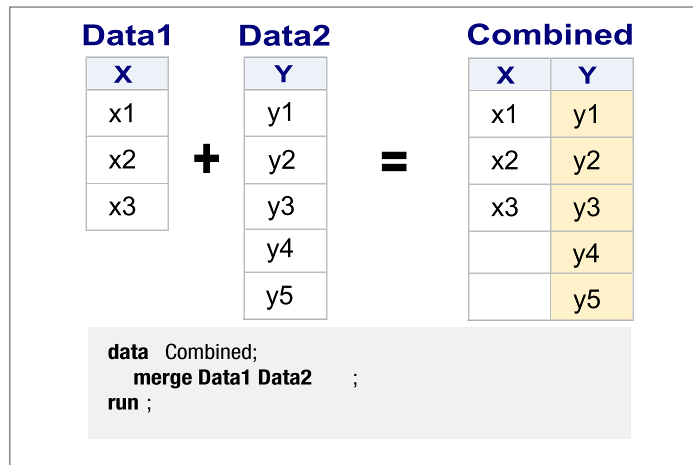
Definition

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.

One-to-One Merging of Two Data Sets

Syntax

MERGE *data-set-1* <*data-set-2*><…
data-set-n>;



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 542
- "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 491

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

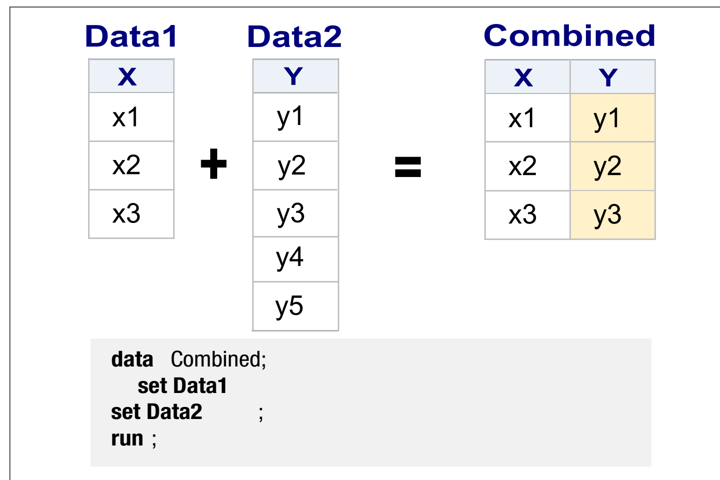
One-to-One Reading of Two Data Sets

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*>;



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 550
- “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 491

Hash Table Merging

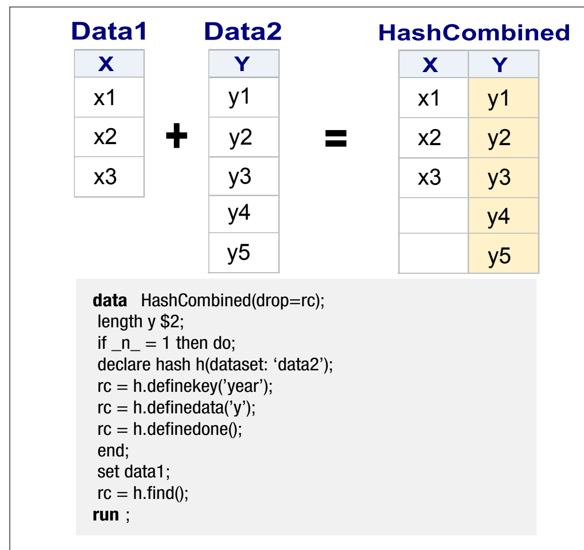
Definition

Hash Table Merging of Two Data Sets

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



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 552
- *SAS Component Objects: Reference*

Updating

Definition

Updating a Master Data Set from a Transaction Data Set

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*;

Master

Year	X	Y
2005	X1	Y1
2006	X1	Y1
2007	X1	Y1
2008	X1	Y1
2009	X1	Y1
2010	X1	Y1
2011	X1	Y1
2012	X1	Y1
2013	X1	Y1
2014	X1	Y1

```
data Combined;
 update Master Trans;
 by Year;
run;
```

+

Trans

Year	X	Y
2011	X2	
2012	X2	Y2
2013	X2	
2013		Y2
2015	X2	Y2

=

Combined

Year	X	Y
2005	X1	Y1
2006	X1	Y1
2007	X1	Y1
2008	X1	Y1
2009	X1	Y1
2010	X1	Y1
2011	X2	Y1
2012	X2	Y2
2013	X2	Y2
2014	X1	Y1
2015	X2	Y2

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 489

See Also

- “Example: Update Data Using the UPDATE Statement” on page 556
- Table 21.2 on page 489

- Updating a Table with Values from Another Table Using PROC SQL

Modifying

Definition

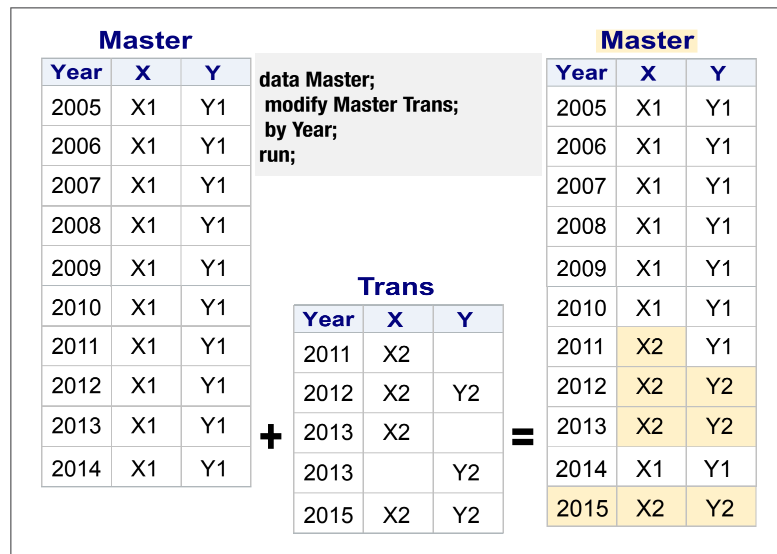
The MODIFY statement matches the values of one or more BY variables in a *master* data set against the same variables in a *transaction* data set. The existing master data set is modified. A new data set is not created.

Syntax

MODIFY *master-data-set transaction-data-set* <(*data-set-options*)>

BY *by-variable*;

Modifying a Master Data Set from a Transaction Data Set



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 489

See Also

- "Example: Modify a Data Set by Adding an Observation" on page 563

■ “Updating and Modifying Comparison” on page 489

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:

```
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:

```
data merged;
   merge Inventory Sales; by PartNumber;
run;
```

Includes both the matching and the non-matching rows in the output by default.

SQL Inner Join Inventory and Sales

Row	PartNumber	PartName	PartNumber	PartName	SalesPerson
1	BV1E	timer	BV1E	timer	JN
2	BV1E	timer	BV1E	timer	KM
3	K89R	seal	K89R	seal	SJ
4	LK43	filter	LK43	filter	KM
5	LK43	filter	LK43	filter	JN
6	M4J7	sander	M4J7	sander	KM
7	NCF3	valve	NCF3	valve	JN

Only matched observations are included

DATA Step Match-Merge Inventory and Sales

Obs	PartNumber	PartName	SalesPerson
1	BC85	clamp	
2	BV1E	timer	JN
3	BV1E	timer	KM
4	JD03	switch	
5	K89R	seal	SJ
6	KJ66	cutter	
7	LK43	filter	KM
8	LK43	filter	JN
9	M4J7	sander	KM
10	MN21	brace	
11	NCF3	valve	JN
12	UYN7	rod	

Matched and unmatched observations are included

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

Characteristic	Updating	Modifying

Syntax	UPDATE statement
	`data <output-data-set>;`
	`UPDATE master-data-set;`
	`trans-data-set`
	`BY variable-name;`
		Form 1: MODIFY statement (matching access)
		`data <output-data-set>;`
		`MODIFY master-data-set;`
		`trans-data-set`
		`BY variable-name;`

When to use	■ You have a master data set that needs to be changed or updated based on the values of a transaction data set. ■ You want to process most of the data in the master data set.	■ 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. ■ You want to process or change only a small portion of the master data set.
Number of data sets that can be processed	UPDATE and MODIFY when specified with the BY statement can combine rows from exactly two data sets.	UPDATE and MODIFY when specified with the BY statement can combine rows from exactly two data sets.
Disk space	"Updating" requires more disk space because it produces an updated copy of the data set.	"Modifying" saves disk space because it updates the existing table without creating a new table.
BY statement requirements	The BY statement is required.	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. *
Unique BY or key value requirements	■ Duplicate 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. ■ Unique values for BY variables are required in the master data set. If duplicate values for the BY variable exist in the master data	■ Duplicate values for BY variables are allowed in either the master data set or the transaction data set. ■ 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. ■ If duplicates exist in the transaction data set, the duplicates are applied one on top

Characteristic	Updating	Modifying
	set, then only the first observation with that value in the BY group is updated and SAS issues a warning.	of another unless you write an accumulation statement to add all of them to the master observation. Without the accumulation statement, the values in the duplicates overwrite each other so that only the value in the last transaction is the result in the master observation.
Sort or index requirements	■ Sorting or indexing is required on input data sets. ■ 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.	■ Sorting or indexing is not required for any data set. ■ 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. ■ If an index exists on the input data set, then the MODIFY statement maintains it on the updated data set.
Treatment of missing values in the input data sets or master data set	■ MODIFY and UPDATE do not overwrite values in the master data set with missing ones in the transaction data set by default. ■ 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 560	
Changes to variables permitted	Yes. You can add new variables, delete variables, change variables, and perform any other change that affects the descriptor information of the data set.	No. You cannot add new variables, delete variables, change variables, or perform any change that affects the descriptor information of the data set.
Error-checking capabilities	No. Updating does not need error checking because transactions without a corresponding master record are added to the data set by default.	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	No data loss occurs because the UPDATE statement works on a copy of the data.	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

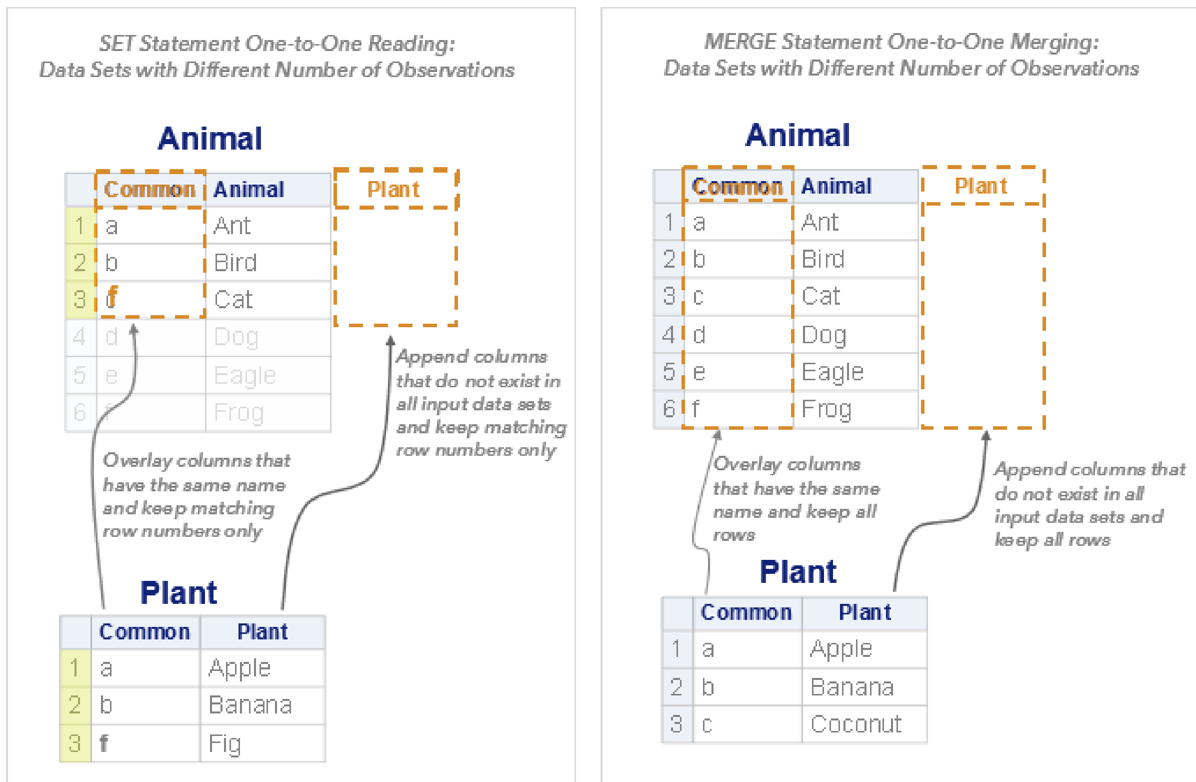
Table 21.3 One-to-One Reading versus One-to-One Merging

Characteristic	One-to-One Reading	One-to-One Merging

Syntax	SET statement (DATA step)
	`data <output-data-set>;`
	`   SET input-data-set-1;`
	`   SET input-data-set-2;`
		MERGE statement (DATA step)
		`data <output-data-set>;`
		`   MERGE input-data-set-1`
		`         input-data-set-2;`

How rows are matched	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.	
Treatment of "unmatched rows"	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.	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.
Treatment of common variables	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.	
Sort requirements	None. There are no sort requirements for either of these methods.	

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

Language Element	Purpose	Access Method		Use with BY statement
		Sequential	**Direct**	
BY	■ 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.	NA	NA	NA

Language Element	Purpose	Access Method		Use with BY statement
		Sequential	Direct	
MERGE statement	■ 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.	X		X
MODIFY	■ Modifying processes rows in a SAS data set by updating the existing master data set rather than creating a new data set (as opposed to updating, which creates a new data set). ■ Sorting is not required but is recommended for performance.	X	X	X
SET	■ The SET statement reads any row from one or more SAS data sets. ■ Interleaving 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. ■ Concatenating (appending) of data sets	X	X	X

Language Element	Purpose	Access Method		Use with BY statement
		Sequential	Direct	
	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. ■ One-to-one reading is achieved when multiple SET statements are used, one for each input data set, and no BY statement is used. ■ The KEY= option and the POINT= option can be specified in the SET statement. They enable you to directly select specific rows for output.			
UPDATE	■ "Updating" applies transactions to rows in a master data set. ■ Updating does not update rows by changing the existing master data set. It produces an updated copy of the existing master data set. ■ Updating requires that the data in both the master and transaction data sets are indexed or sorted by the values of the BY variables.	X		X
PROC APPEND	■ PROC APPEND adds rows from one SAS data set to the end of another SAS data set.	X		
PROC SQL	■ PROC SQL reads any row from one or more SAS data sets. ■ PROC SQL reads rows from up to 32 SAS data	X	X	X

Language Element	Purpose	Access Method		Use with BY statement
		Sequential	Direct	
	sets and joins them into single rows. ■ PROC SQL manipulates rows in an existing SAS data set without creating a new data set. ■ PROC SQL can produce a Cartesian product of the variables in the input data sets. The access method is chosen by the internal optimizer.			
Hash Table	■ 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.	X	X	

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.

Inventory		Sales			Sales2019	
Part Number	Part Name	Part Number	Part Name	Sales Person	Part Number	LastSoldDate

K89R	seal	NCF3	valve	JN	BC85	10/15/2019
M4J7	sander	BV1E	timer	JN	BV1E	10/23/2019
LK43	filter	LK43	filter	KM	KJ66	11/11/2019
MN21	brace	K89R	seal	SJ	LK43	09/12/2019
BC85	clamp	LK43	filter	JN	MN21	09/13/2019
NCF3	valve	M4J7	sander	KM	NCF3	07/24/2019
KJ66	cutter	BV1E	timer	KM	UYN7	12/11/2019
UYN7	rod					
JD03	switch					
BV1E	timer					

```
proc contents data=Inventory; run;            /* 1 */
proc contents data=Sales; run;
proc contents data=Sales2019; run;
quit;
proc sort data=inventory; by PartNumber; run;  /* 2 */
proc sort data=Sales; by PartNumber; run;
proc sort data=Sales2019; by PartNumber; run;

proc print data=inventory; run                /* 3 */;
proc print data=Sales; run;
proc print data=Sales2019; run;
```

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*

The CONTENTS Procedure

WORK.INVENTORY

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
2	partName	Char	8
1	partNumber	Char	8

The CONTENTS Procedure

WORK.SALES

Alphabetic List of Variables and Attributes

#	Variable	Type	Len
2	partName	Char	8
1	partNumber	Char	8
3	salesPerson	Char	8

The CONTENTS Procedure

WORK.SALES2019

Alphabetic List of Variables and Attributes

#	Variable	Type	Len	Format
2	lastSoldDate	Num	8	MMDDYY10.
1	partNumber	Char	8	

Figure 21.3 PROC PRINT Output for the Inventory, Sales, and Sales2019 Data Sets Showing One-to-Many and Many-to-One Relationships

WORK.INVENTORY

Obs	partNumber	partName
1	BC85	clamp
2	BV1E	timer
3	JD03	switch
4	K89R	seal
5	KJ66	cutter
6	LK43	filter
7	M4J7	sander
8	MN21	brace
9	NCF3	valve
10	UYN7	rod

WORK.SALES

Obs	partNumber	partName	salesPerson
1	BV1E	timer	JN
2	BV1E	timer	KM
3	K89R	seal	SJ
4	LK43	filter	KM
5	LK43	filter	JN
6	M4J7	sander	KM
7	NCF3	valve	JN

WORK.SALES2019

Obs	partNumber	lastSoldDate
1	BC85	10/15/2019
2	BV1E	10/23/2019
3	KJ66	11/11/2019
4	LK43	09/12/2019
5	MN21	09/13/2019
6	NCF3	07/24/2019
7	UYN7	12/11/2019

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 488 and "PROC SQL versus Match-Merging" on page 488 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 530, the MODIFY or UPDATE statements, or a hash-table merge on page 552.
- 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 501 "Example: Prepare Data in Which Common Variables Have Different Lengths" on page 504, and "Example: Prepare Data in Which Common Variables Have Different Formats and Labels" on page 506 for more information.

See Also

- "COMPARE Procedure" in *Base SAS Procedures Guide*
- "Example: Find the Common Variables in Multiple Input Data Sets" on page 498

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.

```
proc sql;
   title "Variables Common to Inventory, Sales, and Sales2019";
   create table commonvars as                                                /*
1 */
      select memname, upcase(name) as name
             from dictionary.columns
             where libname='WORK' and
                   memname in ('INVENTORY', 'SALES', 'SALES2019');
   select name                                                               /*
2 */
          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.

Variables Common to Inventory, Sales, and Sales2019

name
PARTNUMBER

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: 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 21.1 Output of PROC FREQ That Shows Duplicate Values for the Common Variable, PartNumber, Found in the Sales Data Set

Obs	partNumber	COUNT
1	BV1E	2
2	LK43	2

Output 21.2 PROC PRINT Output Showing a New Variable Containing Unique Values for the Common Variable PartNumber

Obs	uniqueID	partNumber	partName	salesPerson
1	BV1E.1	BV1E	timer	JN
2	BV1E.2	BV1E	timer	KM
3	K89R.3	K89R	seal	SJ
4	LK43.4	LK43	filter	KM
5	LK43.5	LK43	filter	JN
6	M4J7.6	M4J7	sander	KM
7	NCF3.7	NCF3	valve	JN

Key Ideas

- See “Comparing Methods” on page 488 for a comparison of different combining methods and for information about which methods require unique BY values.

See Also

- For more information about the _N_ automatic variable, see "Automatic Variables" on page 96
- For more information about the CATX function, see "CATX Function" in *SAS Functions and CALL Routines: Reference*
- "Data Relationships" on page 476
- "Overview of Combining Data" on page 474
- Table 21.2 on page 489
- For more information about automatic variables that are generated by PROC FREQ, see Output Data Sets in the FREQ procedure documentation.

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.

3 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

WORK.CARSSMALL

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
4	DriveTrain	Char	5
6	Length	Num	8
1	Make	Char	9
7	MakeModelDrive	Char	58
2	Model	Char	29
3	Type	Char	6
5	Weight	Char	4

WORK.CARS

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
1	Make	Char	13		
2	Model	Char	40		
4	Origin	Char	6		
3	Type	Char	8		
13	Weight	Num	8		Weight (LBS)
14	Wheelbase	Num	8		Wheelbase (IN)

Sort Information	
Sortedby	Make Type
Validated	YES
Character Set	ANSI

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;
```

1 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

Cars by Make Model

Obs	Make	Model	DriveTrain	Weight	MakeModelDrive
1	Acura	3.5 RL 4dr	Front	3880	
2	Acura	3.5 RL w/Navigation 4dr	Front	3893	
3	Acura	MDX	All	4451	
4	Acura	NSX coupe 2dr manual S	Rear	3153	
5	Acura	RSX Type S 2dr	Front	2778	
6	Acura	TL 4dr	Front	3575	
90	Chevrolet	Venture LS	Front	3699	
91	Chevrolet	Aveo 4dr	Front	2370	Chevrolet Aveo 4dr - Front wheel drive
92	Chevrolet	Aveo LS 4dr hatch	Front	2348	Chevrolet Aveo LS 4dr hatch - Front wheel drive
93	Chrysler	300M 4dr	Front	3581	
94	Chrysler	300M Special Edition 4dr	Front	3650	

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*
- "Converting Numeric Values to Character Value" 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.

```
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.

Quarter1

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	4

Quarter2

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	8

Quarter3

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	6

Quarter4

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	6

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.
```

Yearly

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	4

Yearly2

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	account	Num	8
1	mileage	Num	6

You can also use the ATTRIB statement to change the length attribute on the variable:

```
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 505 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;

proc sort data=class; by name; run;                 /* 2 */
proc sort data=classfit; by name; run;

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 21.3 *PROC CONTENTS Output for the Data Sets Class and Classfit Showing the Differences in Formats and Labels*

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Format
2	Age	Num	8	
3	Height	Num	8	COMMA8.
1	Name	Char	8	
4	Weight	Num	8	COMMA8.

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
2	Age	Num	8		
3	Height	Num	8		
1	Name	Char	8		
5	Predict	Num	8	COMMA8.1	
4	Weight	Num	8	COMMA8.2	Weight in Pounds

Output 21.4 PROC CONTENTS Output for the Default Behavior When Formats and Labels Are Different in the Merged Data Sets

Obs	Name	Age	Height	Weight in Pounds	Predict
1	Alice	13	57	84	.
2	Barbara	13	65	98	.
3	Carol	14	63	103	.
4	Jane	12	60	85	.
5	Janet	15	63	113	100.7
6	Joyce	11	51	51	.
7	Judy	14	64	90	.
8	Louise	12	56	77	.
9	Mary	15	67	112	116.3
10	Philip	16	72	112	137.7
11	Ronald	15	67	133	118.2
12	William	15	67	150	116.3

Output 21.5 PROC CONTENTS Output Showing How the ATTRIB Statement Changes the Format and Label on the Merged Output Data Set

Obs	Name	Age	Height	Weight	Predict
1	Alice	13	56.50	84.00	.
2	Barbara	13	65.30	98.00	.
3	Carol	14	62.80	102.50	.
4	Jane	12	59.80	84.50	.
5	Janet	15	62.50	112.50	100.66
6	Joyce	11	51.30	50.50	.
7	Judy	14	64.30	90.00	.
8	Louise	12	56.30	77.00	.
9	Mary	15	66.50	112.00	116.26
10	Philip	16	72.00	112.00	137.70
11	Ronald	15	67.00	133.00	118.21
12	William	15	66.50	150.00	116.26

PROC CONTENTS Output for Data Set Merged

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Label
2	Age	Num	8		
3	Height	Num	8	COMMA8.2	
1	Name	Char	8		
5	Predict	Num	8	COMMA8.2	
4	Weight	Num	8	COMMA8.2	Weight

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:

```
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.

animal			plant		
OBS	common	animal	OBS	common	plant
1	a	Ant	1	a	Apple
2	b	Bird	2	b	Banana
3	c	Cat	3	c	Coconut
4	d	Dog	4	d	Dewberry
5	e	Eagle	5	e	Eggplant
6	f	Frog	6	f	Fig

```
data concatenate;
   set animal plant;              /* 1 */
run;
proc print data=concatenate;      /* 2 */
run;
```

1 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.

2 Print the results (Output 21.6 on page 511).

Output 21.6 *PROC PRINT Output Showing Concatenated Data Sets Using the SET Statement*

Obs	common	animal	plant
1	a	Ant	
2	b	Bird	
3	c	Cat	
4	d	Dog	
5	e	Eagle	
6	f	Frog	
7	a		Apple
8	b		Banana
9	c		Coconut
10	d		Dewberry
11	e		Eggplant
12	f		Fig

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 479 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:

animal			plant		
OBS	common	animal	OBS	common	plant
1	a	Ant	1	a	Apple
2	b	Bird	2	b	Banana
3	c	Cat	3	c	Coconut
4	d	Dog	4	d	Dewberry
5	e	Eagle	5	e	Eggplant
6	f	Frog	6	f	Fig

```
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.

2 Include all variables from animal.

3 Use an OUTER UNION CORRESPONDING statement to include all rows from both tables and to overlay the common variables.

4 Include all variables from plant.

Output 21.7 *Concatenated animal and plant Data Sets Using PROC SQL*

common	animal	plant
a	Ant	
b	Bird	
c	Cat	
d	Dog	
e	Eagle	
f	Frog	
a		Apple
b		Banana
c		Coconut
d		Dewberry
e		Eggplant
f		Fig

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.

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

```
     Year1                        Year2

OBS     Date                  OBS     Date

 1      2009                  1       2010
 2      2010                  2       2011
 3      2011                  3       2012
 4      2012                  4       2013
                              5       2014
```

```
proc append base=Year1 data=Year2;
run;
proc print data=Year1;
   title "Year1";
run;
```

Output 21.8 Concatenated Tables (PROC APPEND)

Year1

Obs	date
1	2009
2	2010
3	2011
4	2012
5	2010
6	2011
7	2012
8	2013
9	2014

Appended rows from Year2

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:

```
data concat;
   set table1 table2;
```

```
run;
proc print data=concat;
   title "Concatenate Table1 and Table2";
run;
```

Concatenate Table1 and Table2

Obs	Name	Sex	Age	Height	Weight	predict	lowermean
1	Alfred	M	14	69.0	112.5	.	.
2	Carol	F	14	62.8	102.5	.	.
3	Henry	M	14	63.5	102.5	.	.
4	Judy	F	14	64.3	90.0	.	.
5	Alfred	M	14	69.0	112.5	126.006	116.942
6	Carol	F	14	62.8	102.5	101.832	96.375
7	Henry	M	14	63.5	102.5	104.562	98.982
8	Judy	F	14	64.3	90.0	107.681	101.842

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

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Carol	F	14	62.8	102.5
3	Henry	M	14	63.5	102.5
4	Judy	F	14	64.3	90.0
5	Alfred	M	14	69.0	112.5
6	Carol	F	14	62.8	102.5
7	Henry	M	14	63.5	102.5
8	Judy	F	14	64.3	90.0

The following message appears in the log:

***Output 21.9** 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:

```
      animal                     plant

OBS  common  animal       OBS  common  plant
 1     a     Ant           1     a     Apple
 2     b     Bird          2     b     Banana
 3     c     Cat           3     c     Coconut
 4     d     Dog           4     d     Dewberry
 5     e     Eagle         5     e     Eggplant
 6     f     Frog          6     f     Fig
```

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.

Output 21.10 *PROC PRINT Output for Interleaved Data Sets*

Obs	common	animal	plant
1	a	Ant	
2	a		Apple
3	b	Bird	
4	b		Banana
5	c	Cat	
6	c		Coconut
7	d	Dog	
8	d		Dewberry
9	e	Eagle	
10	e		Eggplant
11	f	Frog	
12	f		Fig

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 480 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:

animalDupes

OBS	common	animal
1	a	Ant
2	a	Ape
3	b	Bird
4	c	Cat
5	d	Dog
6	e	Eagle

plantDupes

OBS	common	plant
1	a	Apple
2	b	Banana
3	c	Coconut
4	c	Celery
5	d	Dewberry
6	e	Eggplant

```
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 21.11 *PROC PRINT Output for Interleaved Data Sets with Duplicate Values of the BY Variable*

Obs	common	animal	plant
1	a	Ant	
2	a	Ape	
3	a		Apple
4	b	Bird	
5	b		Banana
6	c	Cat	
7	c		Coconut
8	c		Celery
9	d	Dog	
10	d		Dewberry
11	e	Eagle	
12	e		Eggplant

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;
```

Output 21.12 PROC PRINT Output for Interleaved Data Sets with the SET Statement Order Changed

Obs	common	plant	animal
1	a	Apple	
2	a		Ant
3	a		Ape
4	b	Banana	
5	b		Bird
6	c	Coconut	
7	c	Celery	
8	c		Cat
9	d	Dewberry	
10	d		Dog
11	e	Eggplant	
12	e		Eagle

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 480
- “SET 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:

animalDupes

OBS	common	animal
1	a	Ant
2	a	Ape
3	b	Bird
4	c	Cat
5	d	Dog
6	e	Eagle

plantMissing2

OBS	common	plant
1	a	Apple
2	b	Banana
3	c	Coconut
4	e	Eggplant
5	f	Fig

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.

```
proc sort data=animalDupes; by common; run;   /* 1*/
proc sort data=plantMissing2; by common; run;

data interleave;                                     /* 2*/
   set animalDupes plantMissing2;
   by common;
run;

proc print data=interleave; run;
```

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.

Output 21.13 PROC PRINT Output for Interleaved Data Sets with Different Values for the BY Values

Obs	common	animal	plant
1	a	Ant	
2	a	Ape	
3	a		Apple
4	b	Bird	
5	b		Banana
6	c	Cat	
7	c		Coconut
8	d	Dog	
9	e	Eagle	
10	e		Eggplant
11	f		Fig

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 480 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 21.6 *Input Data Sets*

```
      animal                                plant

OBS   common   animal                 OBS   common   plant

 1      a      Ant                     1      a      Apple
 2      b      Bird                    2      b      Banana
 3      c      Cat                     3      c      Coconut
 4      d      Dog                     4      d      Dewberry
 5      e      Eagle                   5      e      Eggplant
 6      f      Frog                    6      f      Fig
```

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:

```
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 21.14 *PROC PRINT Output for Match-Merge Observations Based on a BY Variable*

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Bird	Banana
3	c	Cat	Coconut
4	d	Dog	Dewberry
5	e	Eagle	Eggplant
6	f	Frog	Fig

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 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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 21.7 Input Data Sets

```
     one                                      many
                                      ID city      state
 ID  state

                                      1  Phoenix   Ariz
 1   AZ                               2  Boston    Mass
 2   MA                               2  Foxboro   Mass
 3   WA                               3  Olympia   Mass
 4   WI                               3  Seattle   Wash
                                      3  Spokane   Wash
                                      4  Madison   Wis
                                      4  Milwaukee Wis
                                      4  Madison   Wis
                                      4  Hurley    Wis
```

Output 21.15 PROC PRINT Output Showing Input Data Sets That Contain a Common BY Variable and a Common Non-BY Variable

BY variable

One

ID	State
1	AZ
2	MA
3	WA
4	WI

Many

common (non-BY) variable

Duplicate values

ID	City	State
1	Phoenix	Ariz
2	Boston	Mass
2	Foxboro	Mass
3	Olympia	Mass
3	Seattle	Wash
3	Spokane	Wash
4	Madison	Wis
4	Milwauke	Wis
4	Madison	Wis
4	Hurley	Wis

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.

```
data three;
  merge many one;
  by ID;
run;
proc print data=three noobs; run;
title;
```

Output 21.16 PROC PRINT Output for Match Merging Observations with a Common Variable That is Not the BY Variable

ID	city	state
1	Phoenix	AZ
2	Boston	MA
2	Foxboro	Mass
3	Olympia	WA
3	Seattle	Wash
3	Spokane	Wash
4	Madison	WI
4	Milwauke	Wis
4	Madison	Wis
4	Hurley	Wis

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:

```
data solution;
  merge many(drop=state) one;
  by ID;
run;
proc print data=solution noobs; run;
```

Output 21.17 *PROC PRINT Output for Solution to One-to-many Merge with Common Variables That Are Not the BY Variables*

ID	city	state
1	Phoenix	AZ
2	Boston	MA
2	Foxboro	MA
3	Olympia	WA
3	Seattle	WA
3	Spokane	WA
4	Madison	WI
4	Milwauke	WI
4	Madison	WI
4	Hurley	WI

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).

See Also

- "Match-Merging" on page 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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:

animalDupes			plantDupes		
OBS	common	animal1	OBS	common	plant1
1	a	Ant	1	a	Apple
2	a	Ape	2	b	Banana
3	b	Bird	3	c	Coconut
4	c	Cat	4	c	Celery
5	d	Dog	5	d	Dewberry
6	e	Eagle	6	e	Eggplant

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=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 21.18 *Match-Merge Observations with Duplicate Values of the BY Variable*

Obs	common	animal	plant
1	a	Ant	Apple
2	a	Ape	Apple
3	b	Bird	Banana
4	c	Cat	Coconut
5	c	Cat	Celery
6	d	Dog	Dewberry
7	e	Eagle	Eggplant

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 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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:

```
    animalMissing                       plantMissing2

OBS   common   animal1              OBS   common   plant

1       a       Ant                 1       a       Apple
2       c       Cat                 2       b       Banana
3       d       Dog                 3       c       Coconut
4       e       Eagle               4       e       Eggplant
                                    5       f       Fig
```

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:

```
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 21.19 *PROC PRINT Output for Match-Merge with Unmatched Observations*

Obs	common	animal	plant
1	a	Ant	Apple
2	b		Banana
3	c	Cat	Coconut
4	d	Dog	
5	e	Eagle	Eggplant
6	f		Fig

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 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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 532 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:

animalMissing			plantMissing2		
OBS	common	animal	OBS	common	plant
1	a	Ant	1	a	Apple
2	c	Cat	2	b	Banana
3	d	Dog	3	c	Coconut
4	e	Eagle	4	e	Eggplant
			5	f	Fig

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;
```

Output 21.20 *Match-Merge with Unmatched Observations*

Obs	common	animal	plant
1	a	Ant	Apple
2	b		Banana
3	c	Cat	Coconut
4	d	Dog	
5	e	Eagle	Eggplant
6	f		Fig

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.

```
data matchmerge2;
   merge animalMissing(in=i) plantMissing2(in=j);
   by common;
   if (i=1) and (j=1);
run;
proc print data=matchmerge2; run;
```

Output 21.21 *PROC PRINT Output for Match-Merge Using the IN= Data Set Option to Remove Unmatched Observations*

Obs	common	animal	plant
1	a	Ant	Apple
2	c	Cat	Coconut
3	e	Eagle	Eggplant

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 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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`.

fruit			color		
OBS	ID	fruit	OBS	ID	color
1	a	apple	1	a	amber
2	c	apricot	2	b	brown
3	d	banana	3	b	blue
4	e	bluebarry	4	b	black
5	c	cantaloupe	5	b	beige
6	c	coconut	6	b	bronze
7	c	cherry	7	c	cocoa
8	c	crabapple	8	c	cream
9	c	cranberry			

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 21.22 *PROC PRINT Output for Merging Observations by a Common Variable When Duplicates Exist in More Than One Input Data Set*

Fruit

Obs	id	fruit
1	a	apple
2	a	apricot
3	b	banana
4	b	blueberry
5	c	cantaloupe
6	c	coconut
7	c	cherry
8	c	crabapple
9	c	cranberry

Color

Obs	id	color
1	a	amber
2	b	brown
3	b	blue
4	b	black
5	b	beige
6	b	bronze
7	c	cocoa
8	c	cream

Merged by ID

Obs	id	fruit	color
1	a	apple	amber
2	a	apricot	amber
3	b	banana	brown
4	b	blueberry	blue
5	b	blueberry	black
6	b	blueberry	beige
7	b	blueberry	bronze
8	c	cantaloupe	cocoa
9	c	coconut	cream
10	c	cherry	cream
11	c	crabapple	cream
12	c	cranberry	cream

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 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 481 data sets
- "MERGE Statement" in *SAS DATA Step Statements: Reference*
- "Data Relationships" on page 476
- "Comparing DATA Step Match-Merges with PROC SQL Joins" in *SAS SQL Procedure User's Guide*

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`.

```
    one                          two

OBS    ID      age      score   OBS    ID       name    age

 1      1       8         90     1      1       Sarah   11
 2      1       .        100     2      2       John    10
 3      1       .         95
 4      2       9         80
 5      2       .        100
```

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:

```
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;
```

One

Obs	id	age	score
1	1	8	90
2	1	.	100
3	1	.	95
4	2	9	80
5	2	.	100

Two

Obs	id	name	age
1	1	Sarah	11
2	2	John	10

Merged by ID

Obs	id	age	score	name
1	1	11	90	Sarah
2	1	.	100	Sarah
3	1	.	95	Sarah
4	2	10	80	John
5	2	.	100	John

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;
```

One

Obs	id	age	score
1	1	8	90
2	1	.	100
3	1	.	95
4	2	9	80
5	2	.	100

Two

Obs	id	name	age
1	1	Sarah	11
2	2	John	10

Merged by ID with Age Retained

Obs	id	age	score	name
1	1	11	90	Sarah
2	1	11	100	Sarah
3	1	11	95	Sarah
4	2	10	80	John
5	2	10	100	John

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 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 481 data sets
- “MERGE Statement” in *SAS DATA Step Statements: Reference*
- “Data Relationships” on page 476
- “Comparing DATA Step Match-Merges with PROC SQL Joins” in *SAS SQL Procedure User’s Guide*

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:

```
      animal                                  plantG

OBS   common   animal                   OBS   common   plant

 1      a      Ant                       1      a      Apple
 2      b      Bird                      2      b      Banana
 3      c      Cat                       3      c      Coconut
 4      d      Dog                       4      d      Dewberry
 5      e      Eagle                     5      e      Eggplant
 6      f      Frog                      6      g      Fig
```

The following program merges these data sets and prints the results:

```
data merged;
    merge animal plantG;
run;

proc print data=merged; run;
```

Output 21.23 *PROC PRINT Output for One-to-One Merge of Data Sets with an Equal Number of Observations*

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Bird	Banana
3	c	Cat	Coconut
4	d	Dog	Dewberry
5	e	Eagle	Eggplant
6	g	Frog	Fig

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 476

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:

animal

OBS	common	animal
1	a	Ant
2	b	Bird
3	c	Cat
4	d	Dog
5	e	Eagle
6	f	Frog

plantMissing

OBS	common	plant
1	a	Apple
2	b	Banana
3	c	Coconut

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:

```
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;
```

Output 21.24 PROC PRINT Output for Merging Data Sets with an Unequal Number of Observations

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Bird	Banana
3	c	Cat	Coconut
4	d	Dog	
5	e	Eagle	
6	f	Frog	

Compare the program above to the one-to-one reading of the same data sets using the SET statement:

```
data combine;
   set animal;
   set plantMissing;
run;
proc print data=combine; run;
```

Output 21.25 PROC PRINT Output for Combining Data Sets with an Unequal Number of Observations Using the SET Statement

Obs	Common	Animal	Plant
1	a	Ant	Apple
2	b	Bird	Banana
3	c	Cat	Coconut

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 476

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:

```
      animalDupes                          plantDupes

OBS   common   animal                 OBS   common   plant

 1      a      Ant                     1      a      Apple
 2      a      Ape                     2      b      Banana
 3      b      Bird                    3      c      Coconut
 4      c      Cat                     4      c      Celery
 5      d      Dog                     5      d      Dewberry
 6      e      Eagle                   6      e      Eggplant
```

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.

```
/* 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;
```

Output 21.26 PROC PRINT Output for Undesirable Results When Merging Data Sets That Have Duplicate Values of Common Variables

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Ape	Banana
3	c	Bird	Coconut
4	c	Cat	Celery
5	d	Dog	Dewberry
6	e	Eagle	Eggplant

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 476

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:

animalMissing

OBS	common	animal
1	a	Ant
2	c	Cat
3	d	Dog
4	e	Eagle

plantMissing2

OBS	common	plant
1	a	Apple
2	b	Banana
3	c	Coconut
4	e	Eggplant
5	f	Fig

The following program produces the data set `merged` and prints the results:

```
data merged;
   merge animalMissing plantMissing2;
run;
proc print data=merged; run;
```

Output 21.27 *PROC PRINT Output Showing Undesirable Results with a One-to-One Merge of Data Sets with Different Values for the Common Variable*

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Cat	Banana
3	c	Dog	Coconut
4	e	Eagle	Eggplant
5	f		Fig

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 476

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:

```
      animal                                        plantG

OBS  common  animal                           OBS  common  plant

 1     a     Ant                               1     a     Apple
 2     b     Bird                              2     b     Banana
 3     c     Cat                               3     c     Coconut
 4     d     Dog                               4     d     Dewberry
 5     e     Eagle                             5     e     Eggplant
 6     f     Frog                              6     g     Fig
```

The following program combines the data sets and prints the results:

```
data combine;
   set animal;
   set plantG;
run;

proc print data=combine; run;
```

Output 21.28 *PROC PRINT Output for Combine Data Sets That Contain an Equal Number of Observations Using the SET Statement*

Obs	common	animal	plant
1	a	Ant	Apple
2	b	Bird	Banana
3	c	Cat	Coconut
4	d	Dog	Dewberry
5	e	Eagle	Eggplant
6	g	Frog	Fig

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

Product_Id   Product_Name                        Supplier_ID
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
210200100009 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
```

```
                       supplier

Supplier_ID Supplier_Name                Supplier_Address    Country
50          Scandinavian Clothing A/S    Kr. Augusts Gate 13 NO
316         Prime Sports Ltd             9 Carlisle Place    GB
755         Top Sports                   Jernbanegade 45     DK
772         AllSeasons Outdoor Clothing  553 Cliffview Dr    US
798         Sportico                     C. Barquillo 1      ES
1280        British Sports Ltd           85 Station Street   GB
1303        Eclipse Inc                  1218 Carriole Ct    US
1684        Magnifico Sports             Rua Costa Pinto 2   PT
1747        Pro Sportswear Inc           2434 Edgebrook Dr   US
3298        A Team Sports                2687 Julie Ann Ct   US
3808        Carolina Sports              3860 Grand Ave      US
```

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.

Product Information

Obs	Product_Id	Supplier_ID	Supplier_Name	Supplier_Address	Country
1	240200100101	3808	Carolina Sports	3860 Grand Ave	US
2	210200100017	3298	A Team Sports	2687 Julie Ann Ct	US
3	240400200022	1280	British Sports Ltd	85 Station Street	GB
4	230100100017	50	Scandinavian Clothing A/S	Kr. Augusts Gate 13	NO
5	210200300006	1303	Eclipse Inc	1218 Carriole Ct	US
6	210200500002	772	AllSeasons Outdoor Clothing	553 Cliffview Dr	US
7	210200700016	798	Sportico	C. Barquillo 1	ES
8	210200100009	3298	A Team Sports	2687 Julie Ann Ct	US
9	220100100019	1303	Eclipse Inc	1218 Carriole Ct	US
10	220100100025	1303	Eclipse Inc	1218 Carriole Ct	US
11	220100300001	772	AllSeasons Outdoor Clothing	553 Cliffview Dr	US
12	220200200036	1747	Pro Sportswear Inc	2434 Edgebrook Dr	US
13	230100100015	50	Scandinavian Clothing A/S	Kr. Augusts Gate 13	NO
14	230100500004	316	Prime Sports Ltd	9 Carlisle Place	GB
15	210200500006	772	AllSeasons Outdoor Clothing	553 Cliffview Dr	US
16	230100500006	316	Prime Sports Ltd	9 Carlisle Place	GB
17	224040020000	3808	Carolina Sports	3860 Grand Ave	US
18	220200200035	1747	Pro Sportswear Inc	2434 Edgebrook Dr	US
19	240400200066	1280	British Sports Ltd	85 Station Street	GB
20	240200100081	3808	Carolina Sports	3860 Grand Ave	US

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
- Using the SAS® Hash Object with Duplicate Key Entries

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.

```
          master                              plantNew

OBS  common  animal  plant              OBS  common  plant

 1     a     Ant     Apple               1     a     Apricot
 2     b     Bird    Banana              2     b     Barley
 3     c     Cat     Coconut             3     c     Cactus
 4     d     Dog     Dewberry            4     d     Date
 5     e     Eagle   Eggplant            5     e     Escarole
 6     f     Frog    Fig                 6     f     Fennel
```

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:

```
data master2;
   update master plantNew;
   by common;
run;
proc print data=master2; run;
```

Output 21.29 *PROC PRINT Output for Using the UPDATE Statement to Update Data*

Obs	common	animal	plant
1	a	Ant	Apricot
2	b	Bird	Barley
3	c	Cat	Cactus
4	d	Dog	Date
5	e	Eagle	Escarole
6	f	Frog	Fennel

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 485
- Table 21.2 on page 489
- "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 615

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:

```
          master                          plantNewDupes

Obs  common  animal  plant          Obs common      plant

 1     a      Ant     Apple           1      a      Apricot
 2     b      Bird    Banana          2      b      Barley
 3     c      Cat     Coconut         3      c      Cactus
 4     d      Dog     Dewberry        4      d      Date
 5     e      Eagle   Eggplant        5      d      Dill
 6     f      Frog    Fig             6      e      Escarole
                                      7      f      Fennel
```

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:

```
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.

Output 21.30 *PROC PRINT Output for Update Data Sets with Duplicate Values of the BY Variable*

Obs	common	animal	plant
1	a	Ant	Apricot
2	b	Bird	Barley
3	c	Cat	Cactus
4	d	Dog	Dill
5	e	Eagle	Escarole
6	f	Frog	Fennel

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 485
- Table 21.2 on page 489
- "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 615

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:

```
          master                              minerals

OBS  common  animal  plant             OBS  common  plant    mineral

 1     a     Ant      .                 1     a     Apricot   Amethyst
 2     c     Cat      Coconut           2     b     Barley    Beryl
 3     d     Dog      Dewberry          3     c     Cactus    .
```

```
4     e     Eagle     Eggplant     4     e     .          .
5     f     Frog      Fig          5     f     Fennel     .
                                   6     g     Grape      Garnet
```

The following program updates the `master` data set based on the values in the `minerals` data set and prints the results:

```
data master;
   update master minerals;
   by common;
run;

proc print data=master; run;
```

Output 21.31 *PROC PRINT Output for Using UPDATE for Processing Unmatched Observations, Missing Values, and New Variables*

Obs	common	animal	plant	mineral
1	a	Ant	Apricot	Amethyst
2	b		Barley	Beryl
3	c	Cat	Cactus	
4	d	Dog	Dewberry	
5	e	Eagle	Eggplant	
6	f	Frog	Fennel	
7	g		Grape	Garnet

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:

```
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.

Output 21.32 PROC PRINT Output for the Updated Master Data Set with Values Updated to Missing

Updated Data Set master
With Values Updated to Missing

Obs	common	animal	plant	mineral
1	a	Ant	Apricot	Amethyst
2	b		Barley	Beryl
3	c	Cat	Cactus	
4	d	Dog	Dewberry	
5	e	Eagle		
6	f	Frog	Fennel	
7	g		Grape	Garnet

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 485
- Table 21.2 on page 489
- "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 615

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:

```
                         Inventory


 partNumber   partName          stock      price     receivedDate
```

K89R	seal	34	245.00	07jul2015
M4J7	sander	98	45.88	20jun2015
LK43	filter	121	10.99	19may2016
MN21	brace	43	27.87	10aug2016
BC85	clamp	80	9.55	16aug2016
NCF3	valve	198	24.50	20mar2016
KJ66	cutter	6	19.77	18jun2016
UYN7	rod	211	11.55	09sep2016
JD03	switch	383	13.99	09jan2017
BV1E	timer	26	34.50	03aug2017

InventoryAdd

partNumber	partName	newStock	newPrice
AA11	hammer	55	32.26
BB22	wrench	21	17.35
BV1E	timer	30	36.50
CC33	socket	7	22.19
K89R	seal	6	247.50
KJ66	cutter	10	24.50

To begin this example, first sort and print the `Inventory` and `InventoryAdd` data sets for comparison.

```
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 21.33 PROC PRINT Output for the Master Inventory Data Set Sorted by partNumber

Inventory

Obs	partNumber	partName	stock	price	receivedDate
1	BC85	clamp	80	$9.55	08/16/2017
2	BV1E	timer	26	$34.50	08/03/2018
3	JD03	switch	383	$13.99	01/09/2018
4	K89R	seal	34	$245.00	07/07/2018
5	KJ66	cutter	6	$19.77	06/18/2018
6	LK43	filter	121	$10.99	05/19/2018
7	M4J7	sander	98	$45.88	06/20/2018
8	MN21	brace	43	$27.87	08/10/2018
9	NCF3	valve	198	$24.50	03/20/2017
10	UYN7	rod	211	$11.55	09/09/2017

Output 21.34 PROC PRINT Output for the Transaction Data Set InventoryAdd Sorted by partNumber

InventoryAdd

Obs	partNumber	partName	newStock	newPrice
1	AA11	hammer	55	$32.26
2	BB22	wrench	21	$17.35
3	BV1E	timer	30	$36.50
4	CC33	socket	7	$22.19
5	K89R	seal	6	$247.50
6	KJ66	cutter	10	$24.50

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 21.35 PROC PRINT Output for the Modified Inventory Master Data Set Sorted by partNumber

Modified Inventory Data Set Sorted by partNumber

Obs	partNumber	partName	stock	price	receivedDate
1	AA11	hammer	55	$32.26	01/25/2019
2	BB22	wrench	21	$17.35	01/25/2019
3	BC85	clamp	80	$9.55	08/16/2017
4	BV1E	timer	56	$36.50	01/25/2019
5	CC33	socket	7	$22.19	01/25/2019
6	JD03	switch	383	$13.99	01/09/2018
7	K89R	seal	40	$247.50	01/25/2019
8	KJ66	cutter	16	$24.50	01/25/2019
9	LK43	filter	121	$10.99	05/19/2018
10	M4J7	sander	98	$45.88	06/20/2018
11	MN21	brace	43	$27.87	08/10/2018
12	NCF3	valve	198	$24.50	03/20/2017
13	UYN7	rod	211	$11.55	09/09/2017

New observation Update to existing observation

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 489 and Table 21.4 on page 492.
- 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 487
- Table 21.2 on page 489
- “%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*

22

Using Indexes

Indexes in SAS 571

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 22.1 *Indexes in SAS Engines*

Engine	Documentation	Usage Notes
V9 engine	"Indexes" in *SAS V9 LIBNAME Engine: Reference*	Examples in the linked document show how to create SAS indexes. You can use an index to optimize WHERE or BY processing.
SPD Engine	"Features That Boost Processing Performance" in *SAS Scalable Performance Data Engine: Reference*	You can use the same language elements as with the V9 engine to create or use SAS indexes. 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.

Engine	Documentation	Usage Notes
CAS engine	"Indexing" in *SAS Cloud Analytic Services: Fundamentals*	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.
SAS/ACCESS engines	SAS/ACCESS software documentation on support.sas.com	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.

23

Using Arrays

Definitions for Array Processing **574**

Rules for Referencing Arrays **575**
Rules for Referencing Arrays 575

A Conceptual View of Arrays **576**
One-Dimensional Array 576
Two-Dimensional Array 576

Syntax for Defining and Referencing an Array **577**

Processing Simple Arrays **578**
Grouping Variables in a Simple Array 578
Using a DO Loop to Repeat an Action 579
Using a DO Loop to Process Selected Elements in an Array 580
Selecting the Current Variable 580
Defining the Number of Elements in an Array 581
Rules for Referencing Arrays 582

Variations on Basic Array Processing **583**
Determining the Number of Elements in an Array Efficiently 583
DO WHILE and DO UNTIL Expressions 583
Using Variable Lists to Define an Array Quickly 584

Multidimensional Arrays: Creating and Processing **584**
Grouping Variables in a Multidimensional Array 584
Using Nested DO Loops 585

Specifying Array Bounds **587**
Identifying Upper and Lower Bounds 587
Determining Array Bounds: LBOUND and HBOUND Functions 588
When to Use the HBOUND Function Instead of the DIM Function 588
Specifying Bounds in a Two-Dimensional Array 589

Examples **590**
Example: Using Character Variables in an Array 590
Example: Assigning Initial Values to the Elements of an Array 591
Example: Creating an Array for Temporary Use in the Current DATA Step 592
Example: Performing an Action on All Numeric Variables 593

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;

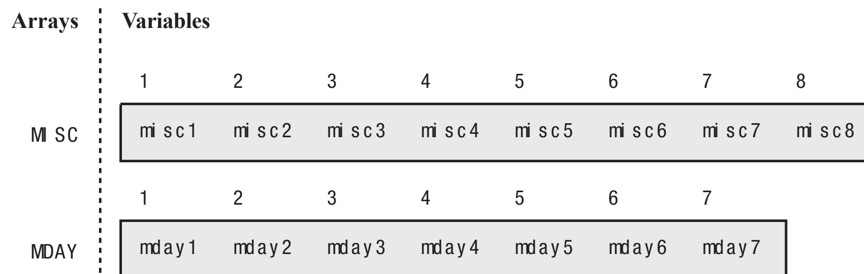
data two;
   array state{*} &list;
   … more SAS statements …
run;
```

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



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.

Figure 23.2 *Example of a Two-Dimensional Array*

First Dimension		**Second Dimension**							
Expense Categories		Days of the Week							Total
		1	2	3	4	5	6	7	8
Hotel	1	hotel1	hotel2	hotel3	hotel4	hotel5	hotel6	hotel7	hotel8
Phone	2	phone1	phone2	phone3	phone4	phone5	phone6	phone7	phone8
Pers. Auto	3	peraut1	peraut2	peraut3	peraut4	peraut5	peraut6	peraut7	peraut8
Rental Car	4	carrnt1	carrnt2	carrnt3	carrnt4	carrnt5	carrnt6	carrnt7	carrnt8
Airfare	5	airlin1	airlin2	airlin3	airlin4	airlin5	airlin6	airlin7	airlin8
Dues	6	dues1	dues2	dues3	dues4	dues5	dues6	dues7	dues8
Registration Fees	7	regfee1	regfee2	regfee3	regfee4	regfee5	regfee6	regfee7	regfee8
Other	8	other1	other2	other3	other4	other5	other6	other7	other8
Tips (non-meal)	9	tips1	tips2	tips3	tips4	tips5	tips6	tips7	tips8
Meals	10	meals1	meals2	meals3	meals4	meals5	meals6	meals7	meals8

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:

ARRAY *array-name* {*number-of-elements*} <$> <*length*> <*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 23.1 Array Reference Assignments for Array Books

Variable	Array Reference
Reference	books{1}
Usage	books{2}
Introduction	books{3}

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:

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:

```
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:

```
array days{7} D1-D7;
```

The following DO statements process selected elements of the array Days:

Table 23.2 DO Statement Processing

DO Statement	Description
`do i=2 to 4;`	processes elements 2 through 4
`do i=1 to 7 by 2;`	processes elements 1, 3, 5, and 7
`do i=3,5;`	processes elements 3 and 5

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;
   datalines;
45 63 113
.  75 150
62 .   98
;

proc print data=changed;
   title 'Number of Books Sold';
run;
```

The following output shows the CHANGED data set.

Output 23.1 *Using an Array Statement to Process Missing Data Values*

Number of Books Sold

Obs	Reference	Usage	Introduction
1	45	63	113
2	0	75	150
3	62	0	98

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:

```
%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:

DIM*n*(*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:

- `do i=1 to dim(days);`
- `do i=1 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.

```
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.

```
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):

```
array temprg{2,5} 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:

```
c1t1 c1t2 c1t3 c1t4 c1t5
c2t1 c2t2 c2t3 c2t4 c2t5
```

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 23.3 *Array References for Array TEMPRG*

Variable	Array Reference
c1t1	temprg{1,1}
c1t2	temprg{1,2}
c2t2	temprg{2,2}
c2t5	temprg{2,5}

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.

```
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*

Temperature Measures for Two Cities

Obs	c1t1	c1t2	c1t3	c1t4	c1t5	c2t1	c2t2	c2t3	c2t4	c2t5	i	j
1	90	65	75	78	89	74	87	90	98	36	3	6
2	76	82	98	94	68	101	87	59	36	76	3	6

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:

LBOUND*n*(*array-name*)

HBOUND*n*(*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=lbound{second} to hbound{second};
   if second{i}=9 then 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:

```
array years{72:76} first second third fourth 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:

```
do i=lbound(years) to hbound(years);
   if years{i}=99 then 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.

```
X60   X61   X62   X63   X64   X65   X66   X67   X68   X69
X70   X71   X72   X73   X74   X75   X76   X77   X78   X79
X80   X81   X82   X83   X84   X85   X86   X87   X88   X89
X90   X91   X92   X93   X94   X95   X96   X97   X98   X99
```

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.

```
options linesize=80 pagesize=60;

data text;
   array names{*} $ n1-n5;              /* 1 */
   array capitals{*} $ c1-c5;
   input names{*};                      /* 2 */
      do i=1 to 5;
         capitals{i}=upcase(names{i}); /* 3 */
      end;
   datalines;
smithers michaels gonzalez hurth frank
;

proc print data=text;
   title 'Names Changed from Lowercase to Uppercase';
run;
```

1 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

Obs	n1	n2	n3	n4	n5	c1	c2	c3	c4	c5	i
1	smithers	michaels	gonzalez	hurth	frank	SMITHERS	MICHAELS	GONZALEZ	HURTH	FRANK	6

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.

```
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};
            output;                    /* 4 */
         end;
   end;
   datalines;
1234  99 60 82
5678  80 85 75
;

proc print noobs data=score1;
   title 'Data Set SCORE1';
run;
```

1 Assign the initial values 90, 80, and 70 to the Test array and assign the variables s1, s2, and s3 to the Score array.

2 The INPUT statement reads a value for the variable named ID and then reads values for all the variables in the Score array.

3 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.

4 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

t1	t2	t3	s1	s2	s3	id	NewScore
90	80	70	99	60	82	1234	99
90	80	70	99	60	82	1234	82
90	80	70	80	85	75	5678	85
90	80	70	80	85	75	5678	75

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*

Data Set SCORE2

s1	s2	s3	id	NewScore
99	60	82	1234	99
99	60	82	1234	82
80	85	75	5678	85
80	85	75	5678	75

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*

Data Set CONVERT

Obs	Value1	Value2	Value3	Value4
1	33	168	174	183
2	66	153	171	183
3	66	147	159	174

24

Debugging Errors

Definitions of Error Types in SAS **595**
Summary of Types of Errors That SAS Recognizes 595
Syntax Errors 596
Semantic Errors 598
Execution-Time Errors 600
Data Errors 604
Macro-related Errors 606

Error Processing in SAS **606**
Syntax Check Mode 606
Checkpoint Mode and Restart Mode 607
Using System Options to Control Error Handling 612
Using Return Codes 614
Other Error-Checking Options 614

Error Checking When Using Indexes to Randomly Access or Update Data **615**
The Importance of Error Checking 615
Error-Checking Tools 615
Example 1: Routing Execution When an Unexpected Condition Occurs 617
Example 2: Using Error Checking on All Statements That Use KEY= 620

Examples **625**
Example: Use the Debugger Tool to Debug Logic Errors in the DATA Step 625
Example: Route Execution When an Unexpected Condition Occurs 626
Example: Use Error Checking on All Statements That Use KEY= 628
Example: Process Multiple Errors 633

Definitions of Error Types 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*

Type of Error	When This Error Occurs	When the Error Is Detected
syntax	when programming statements do not conform to the rules of the SAS language	compile time
semantic	when the language element is correct, but the element might not be valid for a particular usage	compile or execution time
execution-time	when SAS attempts to execute a program and execution fails	execution time
data	when data values are invalid	execution time
macro-related	when you use the macro facility incorrectly	macro compile time or execution time, DATA, or PROC step compile time or execution time

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.

```
date temp;  /* 1  */
   x=1;
run;

proc print data=temp;
run;
```

1 `data` is misspelled as `date`.

Example Code 24.1 *SAS Log: Syntax Error (Misspelled Key Word)*

```
39   date temp;
     ----
     14
WARNING 14-169: Assuming the symbol DATA was misspelled as date.

40      x=1;
41   run;
NOTE: The data set WORK.TEMP has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

42
43   proc print data=temp;
44   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

45   proc printto; 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;
```

Example Code 24.2 *SAS Log: Syntax Error (Missing Semicolon)*

```
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:

```
/* '; * "; */;
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 606.)

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.

```
data _null_;
   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)*

```
81   data _null_;
82      array all{*} x1-x5;
ERROR: invalid reference to the array all.
83      all=3;
84      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

90   ;
91
92   run;
93   proc printto; 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.

```
data test;
   set somelib.old;
run;
```

Example Code 24.4 *SAS Log: Semantic Error (Libref Not Previously Assigned)*

```
101  data test;
ERROR: Libname SomeLib is not assigned.
102     set somelib.old;
103  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)

```
115  data inventory;
116     input Item $ 1-14 TotalCost 15-20
117           UnitsOnHand 21-23;
118     UnitCost=TotalCost/UnitsOnHand;
119     datalines;
NOTE: Division by zero detected at line 118 column 22.
RULE:      ----+----1----+----2----+----3----+----4----+----5---
121        Nylon cord    35    0
Item=Nylon cord TotalCost=35 UnitsOnHand=0 UnitCost=. _ERROR_=1
_N_=2
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 118:22
NOTE: The data set WORK.INVENTORY has 3 observations and 4
      variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
      cpu time            0.00 seconds

123  ;
124
125  proc print data=inventory;
126     format TotalCost dollar8.2 UnitCost dollar8.2;
127  run;

NOTE: Writing HTML Body file: sashtml1.htm
NOTE: There were 3 observations read from the data set
      WORK.INVENTORY.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.56 seconds
      cpu time            0.01 seconds
```

Output 24.1 SAS Output: Execution-Time Error (division by 0)

The SAS System

Obs	Item	TotalCost	UnitsOnHand	UnitCost
1	Hammers	$440.00	55	$8.00
2	Nylon cord	$35.00	0	.
3	Ceiling fans	$1155.00	30	$38.50

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.

```
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)*

```
163  data test;
164     array all{*} x1-x3;
165     input I measure;
166     if measure > 0 then
167        all{I} = measure;
168     datalines;
ERROR: Array subscript out of range at line 167 column 7.
RULE:      ----+----1----+----2----+----3----+----4----+----5---
170        . 3
x1=. x2=. x3=. I=. measure=3 _ERROR_=1 _N_=2
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 1 observations and 5
         variables.
WARNING: Data set WORK.TEST was not replaced because this step
         was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

172  ;
173
174  proc print data=test;
175  run;
NOTE: No variables in data set WORK.TEST.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds

176  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.

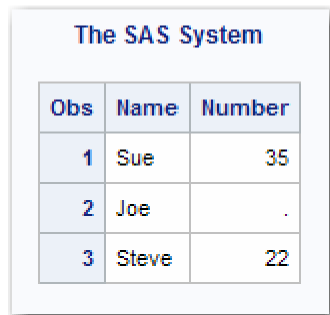
Example Code 24.7 *SAS Log: Data Error*

```
234  data age;
235     input Name $ Number;
236     datalines;
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           0.07 seconds
      cpu time            0.04 seconds
```

Output 24.2 *SAS Output: Data Error*



The SAS System

Obs	Name	Number
1	Sue	35
2	Joe	.
3	Steve	22

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: SYNTTAXCHECK 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.

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;`.

```
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 information, see "CHECKPOINT EXECUTE_ALWAYS Statement" in *SAS Global Statements: Reference*.

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 -errorabbend
```

  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 you 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:

- the _IORC_ automatic variable that SAS creates (and the associated IORCMSG function) when you use the MODIFY statement or the KEY= data set option in the SET statement
- the ERRORS= system option to limit the number of identical errors that SAS writes to the log
- the SYSRC and SYSMSG functions to return information when a data set or external-files access function encounters an error condition
- the SYSRC automatic macro variable to receive return codes

- the SYSERR automatic macro variable to detect major system errors, such as out of memory or failure of the component system
- log control options:

  MSGLEVEL=
  : controls the level of detail in messages that are written to the SAS log.

  PRINTMSGLIST
  : controls the printing of extended lists of messages to the SAS log.

  SOURCE
  : controls whether SAS writes source statements to the SAS log.

  SOURCE2
  : controls whether SAS writes source statements included by %INCLUDE to the SAS log.

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*

Mnemonic Value	Meaning of Return Code	When Return Code Occurs
_DSENMR	The Transaction data set observation does not exist in the Master data set.	MODIFY with BY is used and no match occurs.
_DSEMTR	Multiple Transaction data set observations with the same BY variable value do not exist in the Master data set.	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.
_DSENOM	No matching observation was found in the Master data set.	SET or MODIFY with KEY= finds no match.
_SENOCHN	The output operation was unsuccessful.	the KEY= option in a MODIFY statement contains duplicate values.
_SOK	The I/O operation was successful.	a match is found.

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:

```
         Master                         Transaction

OBS    PartNumber    Quantity       OBS    PartNumber    AddQuantity

 1         1            10           1         4             14
 2         2            20           2         6             16
 3         3            30           3         2             12
 4         4            40
 5         5            50
```

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;
```

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 values of PartNumber.

4 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;  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_;
         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:

```
          Master

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:

```
Master                                  ORDER

  OBS     PartNumber     Quantity              OBS     PartNumber

   1          1             10                  1          2
   2          2             20                  2          4
   3          3             30                  3          1
   4          4             40                  4          3
   5          5             50                  5          8
                                                6          5
                                                7          6
              Description

  OBS     PartNumber     PartDescription

   1          4             Nuts
   2          3             Bolts
   3          2             Screws
   4          6             Washers
```

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;
   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.

```
               Combine

OBS    PartNumber    PartDescription    Quantity
 1          2         Screws               20
```

```
2           4           Nuts              40
3           1           Nuts              10
4           3           Bolts             30
5           8           No description    30
6           5           No description    50
7           6           No description    50
```

Revised Program

To create an accurate output data set, this example performs error checking on both SET statements that use the KEY= option:

```
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;
         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;
      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.

```
                Combine

OBS    PartNumber    PartDescription    Quantity

 1          2        Screws               20
 2          4        Nuts                 40
 3          1        No description       10
 4          3        Bolts                30
 5          5        No description       50
 6          6        Washers               0
```

Examples

Example: 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 (/).

```
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.

See Also

- Using the DATA Step Debugger: Examples
- DEBUG option
- DATA statement
- Base SAS Utilities: Reference

Example: 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:

```
          Master                         Transaction

OBS   PartNumber   Quantity      OBS    PartNumber    AddQuantity

 1         1          10          1          4            14
 2         2          20          2          6            16
 3         3          30          3          2            12
```

```
4          4           40
5          5           50
```

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;
```

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 values of PartNumber.

4 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_;
         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.

Example: Use Error Checking on All Statements That Use KEY=

Overview

It is 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:

```
Master                                    ORDER

  OBS     PartNumber     Quantity               OBS     PartNumber

   1          1             10                   1          2
   2          2             20                   2          4
   3          3             30                   3          1
   4          4             40                   4          3
   5          5             50                   5          8
                                                 6          5
                                                 7          6
              Description

  OBS     PartNumber     PartDescription

   1          4             Nuts
   2          3             Bolts
   3          2             Screws
   4          6             Washers
```

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;
   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.

```
                Combine

OBS    PartNumber    PartDescription   Quantity
 1          2          Screws             20
 2          4          Nuts               40
 3          1          Nuts               10
 4          3          Bolts              30
 5          8          No description     30
 6          5          No description     50
 7          6          No description     50
```

Revised Program

To create an accurate output data set, this example performs error checking on both SET statements that use the KEY= option:

```
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;
         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;
      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.

```
                Combine

OBS    PartNumber    PartDescription   Quantity

 1         2         Screws               20
 2         4         Nuts                 40
 3         1         No description       10
 4         3         Bolts                30
 5         5         No description       50
 6         6         Washers               0
```

Example: Process Multiple Errors

Example Code

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.

```
data temporary;
   Item1=4;
run;

proc print data=temporary;
```

```
      var Item1 Item2 Item3;
   run;
```

Example Code 24.8 SAS Log: Multiple Program Errors

```
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.

25

Optimizing System Performance

Definitions for Optimizing System Performance 636

Collecting and Interpreting Performance Statistics 636
Using the FULLSTIMER and STIMER System Options 636
Interpreting FULLSTIMER and STIMER Statistics 637

Techniques for Optimizing I/O 638
Overview of Techniques for Optimizing I/O 638
Using WHERE Processing 639
Using DROP and KEEP Statements 639
Using LENGTH Statements 640
Using the OBS= and FIRSTOBS= Data Set Options 640
Creating SAS Data Sets 640
Using Indexes 640
Accessing Data through SAS Views 641
Using Engines Efficiently 641
Setting System Options to Improve I/O Performance 642
Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views 644
Using the SASFILE Statement 645
Using the DATASETS Procedure to Modify Attributes 645
Storing Variables as Characters 645

Techniques for Optimizing Memory Usage 646
System Options 646
Using the BY Statement with PROC MEANS 646

Techniques for Optimizing CPU Performance 647
Reducing CPU Time By Using More Memory or Reducing I/O 647
Storing a Compiled Program for Computation-Intensive DATA Steps 647
Reducing Search Time for SAS Executable Files 647
Specifying Variable Lengths 648
Using Parallel Processing 648
Reducing CPU Time By Modifying Program Compilation Optimization 648

Calculating Data Set Size 649

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 on your operating environment, so statistics that you see might differ from the ones shown.

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.

```
libname auto 'SAS-library';
data seatbelt;
   set auto.survey;
   if seatbelt='yes';
run;

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:

```
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 “WHERE Expressions” on page 428.

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 644.

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:

```
/* 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 649 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 "STRIPESIZE= 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 set 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 644.

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 648. 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.

26

Using Parallel Processing

Overview 651
What Is Threading Technology in SAS? 652
How Is Threading Controlled in SAS? 653
Threading in Base SAS 654
SAS/ACCESS Engines 657
SAS Scalable Performance Data Server 657
SAS Intelligence Platform 658
SAS High-Performance Analytics Portfolio of Products 659
SAS Grid Manager 660
SAS In-Database Technology 661
SAS In-Memory Analytics Technology 661
SAS High-Performance Analytics Product Integration 663
SAS Viya 665

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 threaded 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 Scalable Performance Data Server is a multi-user parallel-processing data server with a comprehensive security infrastructure, backup and restore utilities, and administrative and tuning options. SPD Server supports native SQL, FedSQL, and DS2 languages. Options specific to the SPD Server control threaded processing. The NOTHREADS and CPUCOUNT options have no effect. See the *SAS*

Scalable Performance Data Server: Administrator's Guide and *SAS Scalable Performance Data Server: User's Guide* for information.

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 659.)

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, NOTHREADS 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, th 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 *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 NTHREADSoption 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 installed. 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 . . . 669

Chapter 28
Printing . . . 697

27

Output

Definitions for SAS Output 670

Default Output Destination 671

Change the Output Destination 672

Customize Output 674
- Making Output Descriptive 674
- Using ODS to Customize the Style and Structure of Output 676
- Reformatting Values in Output 676
- Printing Missing Values 677

The SAS Log 677
- Structure of the Log 677
- The SAS Log in Batch, Line, or Objectserver Modes 679
- Writing to the Log in All Modes 681
- Customizing the Log 682

Examples: Manage Output Destinations 686
- Example: Create Default HTML Output 686
- Example: Use ODS Statements to Change the Output Destination 687

Examples: Rolling Over the SAS Log 688
- Example : Use Directives to Name the SAS Log 688
- Example: Automatically Roll Over the SAS Log When Directives Change 689
- Example: Roll Over the SAS Log by SAS Session 690
- Example: Roll Over the SAS Log by the Log Size 691

Examples: Suppress Output to the SAS Log 692
- Example: Suppress the Printing of the Source Program to the SAS Log Using the NOSOURCE System Option 692
- Example: Suppress the Printing of Variable Data to the SAS Log Using the NOLIST Option 694
- Example: Suppress the Printing of All Notes to the SAS Log Using the NONOTES System Option 695

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 681.

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 682 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

SAS Version	Mode of Running SAS	Viewer	ODS Destination
SAS 9.3 and later	SAS Windowing Environment	SAS Results Viewer or browser window	HTML
	SAS Studio	Results tab or browser window	HTML5
	Enterprise Guide	Project pane, process flow, or browser window	HTML5
	batch mode	Depends on operating environment	

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

Method to Use	Output Result
PRINTTO procedure	routes DATA step, log, or procedure output from the system default destinations to the

Method to Use	Output Result
	destination that you choose. The PRINTTO procedure defines non-ODS destinations.
FILENAME statement	associates a fileref with an external file or output device and enables you to specify file and device attributes
FILE command: Windows	stores the contents of the LOG or OUTPUT windows in files that you specify, when the command is issued from within the windowing environment.
ODS LISTING statement	opens, manages, or closes the LISTING destination.
ODS OUTPUT statement	produces a SAS data set from an output object and manages the selection and exclusion lists for the OUTPUT destination.
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

Method to Use	Output Result
	ALTLOG=, ALTPRINT=, LOG=, and PRINT= options.

For conceptual information about global ODS statements, see the following resources:

- "Destination Category Table" in *SAS Output Delivery System: User's Guide*.
- "Types of ODS Statements" in *SAS Output Delivery System: User's Guide*.

Operating Environment Information:

For information about changing the default output location for the z/OS and UNIX operating environments, see the following resources:

- z/OS: "Directing SAS Log and SAS Procedure Output" in *SAS Companion for z/OS*. and "Changing the Default Destination" in *SAS Companion for z/OS*
- 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 27.3 *Methods for Customizing Output*

SAS Language Element	Function
CENTER | NOCENTER system option	controls whether output is centered. By default, SAS centers titles and procedure output on the page and on the personal computer display.
DATE | NODATE system option	controls printing of date and time values. When this option is enabled, SAS prints on the

SAS Language Element	Function
	top of each page of output the date and time the SAS job started.
FOOTNOTE statement	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.
FORMCHAR= system option	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.
FORMDLIM= system option	specifies a character to delimit page breaks in SAS output for the LISTING destination.
LABEL statement	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 *Base SAS Procedures Guide* for information about using the LABEL statement with a specific procedure.
LINESIZE= and PAGESIZE= system options	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.
NUMBER | NONUMBER and PAGENO= system options	controls page numbering. The NUMBER system option controls whether the page number is printed on the first title line of each page of printed output. You can also specify a beginning page number for the next page of

SAS Language Element	Function
	output produced by SAS by using the PAGENO= system option.
TITLE statement	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.

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 modify existing style and table definitions if you do not want to create the definitions from scratch.

For information about the Output Delivery System, see *SAS Output Delivery System: User's Guide*.

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
1    options pagesize=24
2    linesize=64 pageno=1 nodate;  5
3    data logsample;  6
5       infile
5  ! '\\myserver\my-directory-path\sampledata.dat';  7
6       input LastName $ ID $ Gender $ Birth : date7. score1
6  ! score2 score3 score4 score5 score6 score7 score8;
7       format Birth mmddyy8.;
8    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

Note: Copyright information, licensing and site information, and the number of observations and variables in the data set can be suppressed using the NOTES | NONOTES System Option

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 688
- "Example: Automatically Roll Over the SAS Log When Directives Change" on page 689

- "Example: Roll Over the SAS Log by SAS Session" on page 690
- "Example: Roll Over the SAS Log by the Log Size" on page 691

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*

Statements	Descriptions	Examples
DATA Statement with /NESTING Option	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.	"DATA Statement" in *SAS DATA Step Statements: Reference*
ERROR Statement	Sets _ERROR_ to 1. A message written to the SAS log is optional.	"ERROR Statement" in *SAS DATA Step Statements: Reference*
LIST Statement	Writes to the SAS log the input data record for the observation that is being processed.	"LIST Statement" in *SAS DATA Step Statements: Reference*
PUT Statement	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.	"PUT Statement" in *SAS DATA Step Statements: Reference*
%PUT Statement	Writes text or macro variable information to the SAS log.	"%PUT Macro Statement" in *SAS Macro Language: Reference*
PUTLOG Statement	Writes a message to the SAS log.	"PUTLOG Statement" in *SAS DATA Step Statements: Reference*

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

SAS System Options	Descriptions
CPUID | NOCPUID	Specifies whether the CPU identification number is written to the SAS log.
ECHO: Windows and UNIX	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.
ECHOAUTO | NOECHOAUTO	Specifies whether autoexec code in an input file is written to the log.
ERRORS=	Specifies the maximum number of observations for which SAS issues complete error messages.
FULLSTATS: z/OS	Writes expanded statistics to the SAS log. The FULLSTATS system option is valid only under z/OS.
FULLSTIMER: Windows FULLSTIMER: UNIX	Writes a subset of system performance statistics to the SAS log.

ISPNOTES: z/OS	Specifies whether ISPF error messages are written to the SAS log. The ISPNOTES system option is valid only under the z/OS operating environment.
HOSTINFOLONG	Writes additional operating environment information to the SAS log when SAS starts.
LOGPARM "OPEN=APPEND | REPLACE | REPLACEHOLD"	When a log file already exists and SAS is opening the log, the LOGPARM option specifies whether to append to the existing log or to replace the existing log. The REPLACEHOLD option specifies to replace logs that are more than one day old.
MEMRPT: z/OS	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.
MLOGIC	Writes macro execution trace information to the SAS log.
MLOGICNEST	Specifies whether to display the macro nesting information in the MLOGIC output in the SAS log.
MPRINT | NOMPRINT	Specifies whether SAS statements that are generated by macro execution are written to the SAS log.
MPRINTNEST	Specifies whether to display the macro nesting information in the MPRINT output in the SAS log.
MSGLEVEL=	Specifies the level of detail in messages that are written to the SAS log.
NEWS=	Specifies whether news information that is maintained at your site is written to the SAS log.
NOTE | NONOTES	Specifies whether notes (messages beginning with NOTE) are written to the SAS log. NONOTES does not suppress error or warning messages.
OPLIST: Windows OPLIST: z/OS OPLIST: UNIX	Specifies whether the settings of the SAS system options are written to the SAS log.
OVP | NOOVP	Specifies whether overprinting of error messages to make them bold, is enabled.
PAGEBREAKINITIAL	Specifies whether the SAS log and the listing file begin on a new page.

PRINTMSGLIST | NOPRINTMSGLIST	Specifies whether extended lists of messages are written to the SAS log.
RTRACE: Windows RTRACE: UNIX	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.
SOURCE | NOSOURCE	Specifies whether SAS writes source statements to the SAS log.
SOURCE2 | NOSOURCE2	Specifies whether SAS writes secondary source statements from files included by %INCLUDE statements to the SAS log.
SYMBOLGEN | NOSYMBOLGEN	Specifies whether the results of resolving macro variable references are written to the SAS log
VERBOSE: Windows VERBOSE: z/OS VERBOSE: UNIX	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.

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*

DETAILS system option	Specifies whether to include additional information when files are listed in a SAS library.
DMSLOGSIZE= system option	Specifies the maximum number of rows to display in the SAS log window.
DTRESET system option	Specifies whether to update the date and time in the SAS log and in the listing file.
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*

Student Weight

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
8	Janet	F	15	62.5	112.5
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
16	Robert	M	12	64.8	128.0
17	Ronald	M	15	67.0	133.0
19	William	M	15	66.5	112.0

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*

Students

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5
8	Janet	F	15	62.5	112.5
14	Mary	F	15	66.5	112.0
15	Philip	M	16	72.0	150.0
16	Robert	M	12	64.8	128.0
17	Ronald	M	15	67.0	133.0
19	William	M	15	66.5	112.0

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

- "Understanding ODS Destinations" in *SAS Output Delivery System: User's Guide*

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 -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

Rollover Time	Log Name
SAS initialization	londonThames0206.log
First rollover	londonThames0207.log
Last log of the day	londonThames0223.log
First log past midnight	londonThames0300.log

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 -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 -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 *logname*.lck, where *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 *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:

```
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*

```
73         data mytable;
74         password="ABC123";
75         x = 1;
76         run;
```

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*

```
114  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           0.03 seconds
         cpu time            0.00 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 uses 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*

```
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.

```
options NONOTES;
data example;
   password="ABC123";
run;
```

Example Code 27.7 *Log output showing how notes are suppressed in the SAS log*

```
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*

28

Printing

Universal Printing 697
Definition of Universal Printing 697

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 9.4 Universal Printing for details.

PART 6

Managing Files

Chapter 29
Protecting Files 701

Chapter 30
Repairing SAS Files 719

Chapter 31
Compressing SAS Data Sets 733

Chapter 32
Moving SAS Files 735

Chapter 33
Cross-Environment Data Access 737

Chapter 34
Managing SAS Catalogs 757

29

Protecting Files

Definition of a Password **702**

Assigning Passwords **703**
- Syntax 703
- Assigning a Password with a DATA Step 703
- Assigning a Password to an Existing Data Set 704
- Assigning a Password with a Procedure 704
- Assigning a Password with the SAS Windowing Environment 705
- Assigning a Password outside of SAS 705

Removing or Changing Passwords **705**

Using Password-Protected SAS Files in DATA and PROC Steps **705**

How SAS Handles Incorrect Passwords **706**

Assigning Complete Protection with the PW= Data Set Option **707**

Encoded Passwords **708**

Using Passwords with Views **708**
- Levels of Protection 708
- PROC SQL Views 710
- SAS/ACCESS Views 710
- DATA Step Views 710

SAS Data File Encryption **711**
- About Encryption on SAS Data Files 711
- SAS Proprietary Encryption 712
- AES Encryption 713
- AES Encryption and Referential Integrity Constraints 715
- Passwords and Encryption with Generation Data Sets, Audit Trails, Indexes, and Copies 715

Blotting Passwords and Encryption Key Values **716**
- Check the SAS Log 716
- Examples of Passwords and Encryption Keys That Are Not Blotted 716
- Using Macros 717
- Length of Passwords 718

Metadata-Bound Libraries **718**

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 707.

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 708.

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.

- ```
     /* 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;
  ```
- ```
     /* Append ANIMALS to the write-protected data set ZOO */
  proc append base=mylib.zoo(write=yellow) data=mylib.animals;
  run;
  ```
- ```
     /* Delete alter-protected data set MYLIB.BOTANY */
  proc datasets library=mylib;
     delete botany(alter=red);
  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:

```
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:

```
   /* 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
… more data lines …
;

proc print data=mylib.states(read=orange);
run;
```

- PW= can be an alias for other password options:

```
   /* 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
… more data lines …
;

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:

```
   /*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 someone from updating the underlying data, assign Write protection to the view. For more information, see the SAS/ACCESS documentation for your DBMS.

DATA Step Views

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:

```
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 712 is implemented with the ENCRYPT=YES data set option.
- AES (Advanced Encryption Standard) encryption on page 713 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*

Features	ENCRYPT=YES	ENCRYPT=AES	ENCRYPT=AES2
License required	No	No	No
Encryption level	Medium	High	Highest
Algorithm supported	SAS Proprietary (within Base SAS software)	AES	AES2

Features	ENCRYPT=YES	ENCRYPT=AES	ENCRYPT=AES2
Installation required	No (part of Base SAS software)	No SAS/SECURE (delivered with Base SAS software)	No SAS/SECURE (delivered with Base SAS software)
Operating environments supported	UNIX Windows z/OS	UNIX Windows z/OS	UNIX Windows z/OS
SAS version support	8 and later	9.4 and later	9.4m5 and later

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:

```
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:

```
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.

See Also

"AUTHLIB Procedure" in *Base SAS Procedures Guide*

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:

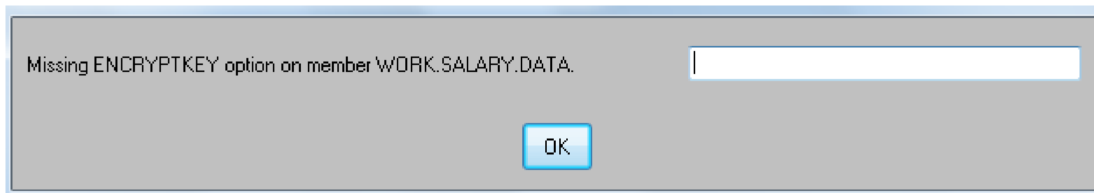
```
proc print data=salary(encryptkey=green);
```

```
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 712 and "AES Encryption" on page 713.

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:

```
proc print data=lubrary.abc(READ=secret);
run;
```

or

```
proc print data=library.abc(ERAD=secret);
run;
```

- 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.

```
libname mylib 'c:\';
   data mylub.abc(
   read=secret
   );
   x=1;
   run;
```

The following output is written to the SAS log:

```
636  libname mylib 'c:\';
NOTE: Libref MYLIB was successfully assigned as follows:
      Engine:        V9
      Physical Name: c:\
637     data mylub.abc(
638     read=secret
639     );
ERROR: Libref MYLUB is not assigned.
640     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:

```
options pdfpassword=(open=a owner=b );
proc options option=pdfpassword;
run;
```

The following is written to the SAS log:

```
634  options pdfpassword=XXXXXXX 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*.

30

Repairing SAS Files

Repairing Damage to SAS Data Sets and Catalogs 719
Overview 719
Understanding Damage That Can Be Repaired with SAS 721
Understanding the Repair Tools 721
Recommended Tasks for Repairing SAS Data Sets and Catalogs with SAS 722

Examples 724
Example: Determine the Default Repair Action for Your SAS Session 724
Example: Change the DLDMGACTION= System Option 725
Example: Repair a SAS Data Set with the REPAIR Statement 726
Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option 727
Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement 729
Example: Determine How Many Times a Data Set Was Repaired 730

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 721.

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

SAS File Type or Damage Type	Solution
DATA step views and PROC SQL views	Re-create the file.
Stored compiled DATA step programs	
Truncated SAS data sets, or SAS data sets whose audit file is damaged or deleted	Recover from a backup device.[1]
SAS data sets and catalogs that are on a damaged storage device	
Catalogs that cannot be repaired by the DLDMGACTION= system option or data set option, or by using the PROC DATASETS REPAIR statement	
Complex indexes	Repair the data set without indexes, and then re-create the index using the PROC DATASETS "INDEX CREATE Statement" in *Base SAS Procedures Guide*. Or, recover the data set from a backup device.
Indexes copied by using a DATA step	
Indexes that were deleted when using the FORCE option in the SORT procedure.	

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 730.

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

Task	Code	Description	Example
Determine the DLDMGACTION= system option setting for your SAS session	`proc options option=dldmgaction value; run;`	Prints the setting of the DLDMGACTION= system option to the SAS log.	"Example: Determine the Default Repair Action for Your SAS Session" on page 724.
Change how SAS responds when it detects damaged files (optional)	`options dldmgaction=value;`	Enables you to change or configure a repair response for the SAS session.	"Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement" on page 729

Task	Code	Description	Example
Actively repair damaged files yourself	`(dldmgaction=value)`	Enables you to specify a repair action for a specific SAS data set or catalog.	"Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option" on page 727.

	`proc datasets;`
	`  repair filename;`
	`run;`
		Attempts to repair a damaged SAS data set or catalog.	"Example: Repair a SAS Data Set with the REPAIR Statement" on page 726

	`proc datasets;`
	`  rebuild filename;`
	`run;`
		Specifies whether to restore or delete indexes and integrity constraints that were disabled by the DLDMGACTION=NOINDEX option.	"Example: Rebuild a Damaged SAS Data Set with the REBUILD Statement" on page 729

View a data set's repair history	`proc contents`
	`  data=filename;`
	`run;`
		Lists information about a SAS data set. Check the "Number of Data Set Repairs" and "Last Repaired" entries in the Engine/Host Dependent Information section of the output.	"Example: Determine How Many Times a Data Set Was Repaired" on page 730

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.

```
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*

```
Option Value Information For SAS Option DLDMGACTION
    Value: REPAIR
    Scope: Default
    How option value set: Shipped Default
```

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 721. 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 726.
- "Example: Override the Default Repair Action with the DLDMGACTION= Data Set Option" on page 727.

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.

```
options dldmgaction=prompt;

proc options option=dldmgaction value;
run;
```

Output 30.1 *Log Output from PROC OPTIONS*

```
73          options dldmgaction=prompt;
74
75          proc options option=dldmgaction value;
76           run;

    SAS (r) Proprietary Software Release 9.4  TS1M6

Option Value Information For SAS Option DLDMGACTION
    Value: PROMPT
    Scope: Default
    How option value set: OPTIONS Statement
```

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 729

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 727.

```
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 DLDMGACTION=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 730.

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.

Output 30.2 *The Engine/Host Dependent Information Section of the CONTENTS Procedure Output*

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	2
First Data Page	1
Max Obs per Page	2715
Obs in First Data Page	200
Index File Page Size	4096
Number of Index File Pages	3
Number of Data Set Repairs	2
Last Repair	13:44 Tuesday, November 13, 2018
ExtendObsCounter	YES
Filename	C:\dldmgactiontest\myfile.sas7bdat
Release Created	9.0401M6
Host Created	X64_10PRO
Owner Name	Computer_Name/User_ID
File Size	192KB
File Size (bytes)	196608

Key Ideas

- The CONTENTS procedure has two fields that report information about repairs:
  - Number of Data Set Repairs
  - Last Repair

See Also

"CONTENTS Procedure"

31

Compressing SAS Data Sets

Compression in SAS 733

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*

Engine	Documentation	Usage Notes
V9 engine	"Compression" in *SAS V9 LIBNAME Engine: Reference*	Supports COMPRESS=NO | CHAR | BINARY.

Engine	Documentation	Usage Notes
SPD Engine	"Updates to a Compressed SPD Engine Data Set" in *SAS Scalable Performance Data Engine: Reference*	Supports COMPRESS=NO | CHAR | BINARY. Additional language elements for the SPD Engine enable more control over compression.
CAS engine	"Data Compression" in *SAS Cloud Analytic Services: User's Guide*	Supports COMPRESS=YES | NO.

32

Moving SAS Files

Moving Files between Operating Environments 735

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*.

33

Cross-Environment Data Access

Definitions for Cross-Environment Data Access (CEDA) 737

Advantages of CEDA 738

SAS File Processing with CEDA 739
- Conditions That Invoke CEDA 739
- Restrictions for CEDA 740
- Compatible Data Representations 741
- Compatible Encodings 743
- How Output Processing Affects Encoding and Data Representation 743

Alternatives to Using CEDA 744

Examples: CEDA 745
- Example: Understand Log Messages about CEDA 745
- Example: Understand Log Messages about Truncation 746
- Example: Specify a Data Representation to Avoid CEDA 748
- Example: Specify an Encoding to Avoid CEDA 750
- Example: Specify a Data Representation and UTF-8 Encoding 752
- Example: Determine the Encoding and Data Representation of a SAS Session or a Data Set 754

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 741.)

- 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 741.

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 33.1 *SAS File Processing Provided by CEDA*

SAS File Type	Engine	Supported Processing
SAS data set	V9, SASESOCK, SPDE, TAPE	input and output[1]
PROC SQL view (DATA step view is not supported)	V9	input

SAS File Type	Engine	Supported Processing
SAS/ACCESS view for Oracle or SAP	V9	input
MDDB file[2]	V9	input

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 743.

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 107.
- 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 33.2 Compatibility across Environments

Data Representation Value and Environment Name	Notes
ALPHA_TRU64 (Tru64 UNIX) LINUX_IA64 (Linux for Itanium-based systems) LINUX_X86_64 (Linux for x64) SOLARIS_X86_64 (Solaris for x64)	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.

Data Representation Value and Environment Name	Notes
LINUX_POWER_64 (Linux on the Power Architecture)	Linux on the Power Architecture is added in SAS Viya 3.5 and is not supported in SAS 9.
ALPHA_VMS_32 (OpenVMS Alpha) ALPHA_VMS_64 (OpenVMS Alpha) VMS_IA64 (OpenVMS on HP Integrity)	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*.
HP_IA64 (HP-UX for the Itanium Processor Family Architecture) HP_UX_64 (HP-UX for PA-RISC, 64-bit) RS_6000_AIX_64 (AIX) SOLARIS_64 (Solaris for SPARC)	All of the environments in this group are compatible.
HP_UX_32 (HP-UX for PA-RISC) MIPS_ABI (MIPS ABI) RS_6000_AIX_32 (AIX) SOLARIS_32 (Solaris for SPARC)	All of the environments in this group are compatible.
LINUX_32 (Linux for Intel architecture) INTEL_ABI (ABI for Intel architecture)	All of the environments in this group are compatible.
MVS_32 (31-bit SAS on z/OS)	No other environments are compatible.
MVS_64_BFP (64-bit SAS on z/OS)	No other environments are compatible.
OS2 (OS/2 for Intel)	No other environments are compatible.
VAX_VMS (OpenVMS VAX)	No other environments are compatible.
WINDOWS_32 (32-bit SAS on Microsoft Windows) WINDOWS_64 (64-bit SAS on Microsoft Windows, for both Itanium-based systems and x64)	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.

Compatible Encodings

Compatible encodings do not require CEDA processing for transcoding. See "Encoding Combinations That Do Not Need CEDA Processing for Transcoding" in *SAS National Language Support (NLS): Reference Guide*.

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 748.

```
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.

```
proc sql;
   insert into myfiles.mytest
      values ('other data string');
quit;
```

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 740
- “Transcoding Considerations” in *SAS National Language Support (NLS): Reference Guide*
- “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.

```
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.

```
libname myfiles v9 'c:\examples';
proc print data=myfiles.mytrunctest;
run;
```

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*

Obs	a
1	サンプルテキス

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 740
- "Transcoding Considerations" in *SAS National Language Support (NLS): Reference Guide*

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.

```
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*

Data Set Name	MYFILES.MYTEST	Observations	1
Member Type	DATA	Variables	1
Engine	V9	Indexes	0
Created	08/27/2019 10:22:01	Observation Length	18
Last Modified	08/27/2019 10:22:01	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	latin1 Western (ISO)		

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 741
- "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*

Data Set Name	MYLNX.MYTEST	Observations	1
Member Type	DATA	Variables	1
Engine	V9	Indexes	0
Created	08/27/2019 15:31:24	Observation Length	18
Last Modified	08/27/2019 15:31:24	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

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

- "ENCODING= Data Set Option" in *SAS National Language Support (NLS): Reference Guide*
- "INENCODING=, OUTENCODING= LIBNAME Statement Options" in *SAS V9 LIBNAME Engine: Reference*
- "Encoding for NLS" in *SAS National Language Support (NLS): Reference Guide*

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.

```
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*

Data Set Name	MYFILES.TEST	Observations	1
Member Type	DATA	Variables	1
Engine	V9	Indexes	0
Created	09/03/2019 14:51:14	Observation Length	8
Last Modified	09/03/2019 14:51:14	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
Encoding	utf-8 Unicode (UTF-8)		

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

- "Default Values for DFLANG, DATESTYLE, and PAPERSIZE System Options Based on the LOCALE= System Option" in *SAS National Language Support (NLS): Reference Guide*
- "ENCODING System Option: UNIX, Windows, and z/OS" in *SAS National Language Support (NLS): Reference Guide*
- "ENCODING= Data Set Option" in *SAS National Language Support (NLS): Reference Guide*
- "INENCODING=, OUTENCODING= LIBNAME Statement Options" in *SAS V9 LIBNAME Engine: Reference*
- "Compatibility and Migration" in *SAS V9 LIBNAME Engine: Reference*

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:

```
%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:

```
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.

```
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

Data Set Name	WORK.SESSIONTEST	Observations	1
Member Type	DATA	Variables	1
Engine	V9	Indexes	0
Created	09/03/2019 14:17:06	Observation Length	8
Last Modified	09/03/2019 14:17:06	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_64		
Encoding	wlatin1 Western (Windows)		

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 741.
- 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*

34

Managing SAS Catalogs

Definition of a SAS Catalog 757

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, informats, 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 761

Chapter 36
The SAS Windowing Environment 781

Chapter 37
Cloud Analytic Services 831

Chapter 38
Industry Protocols Used in SAS 835

35

The SAS Registry

Introduction to the SAS Registry **761**
What Is the SAS Registry? 761
Who Should Use the SAS Registry? 762
Where the SAS Registry Is Stored 762
How Do I Display the SAS Registry? 763
Definitions for the SAS Registry 763

Managing the SAS Registry **765**
Primary Concerns about Managing the SAS Registry 765
Backing Up the Sasuser Registry 765
Recovering from Registry Failure 768
Using the SAS Registry to Control Color 769
Using the Registry Editor 770

Configuring Your Registry **776**
Configuring Universal Printing 776
Configuring SAS Explorer 776
Configuring Libraries and File Shortcuts with the SAS Registry 777
Fixing Library Reference (Libref) Problems with the SAS Registry 779

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:

```
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'

Contents of 'HTML'

Name	Data
coco	"D2,71,1E"
coco1	D2,73,1E

.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[1] 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

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.

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:

- 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.

7 Right-click the **`Regstry.Itemstor`** file.

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 **`regstry.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 *regstry.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 regstry.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 '.'
```

4 Run the REGISTRY procedure and redefine the Sasuser registry:

```
proc registry setsasuser="here.temp";
run;
```

5 Start the Registry Editor with the REGEDIT command. Select **Solutions** ⇨ **Accessories** ⇨ **Registry Editor** ⇨ **View All**.

6 Edit any damaged fields under the HKEY_USER_ROOT key.

7 Close your SAS session and rename the modified registry back to the original name.

8 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.

3 Select **Edit** ⇨ **New Binary Value**. A pop-up menu appears.

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.

```
filename mycolors temp;
data _null_;
   file "mycolors";
   put "[colornames\html]";
   put ' "rojo"=hex:ff,00,00';
   put ' "verde"=hex:00,ff,00';
   put ' "azul"=hex:00,00,ff';
   put ' "blanco"=hex:ff,ff,ff';
   put ' "negro"=hex:00,00,00';
   put ' "anaranjado"=hex:ff,a5,00';
run;

proc registry import="mycolornames";
run;
```

After you add these colors to the registry, you can use these color names anywhere that you use the color names supplied by SAS. For example, you could use the color name in the GOPTIONS statement as shown in the following code:

```
goptions cback=anaranjado;
proc gtestit;
run;
```

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**.

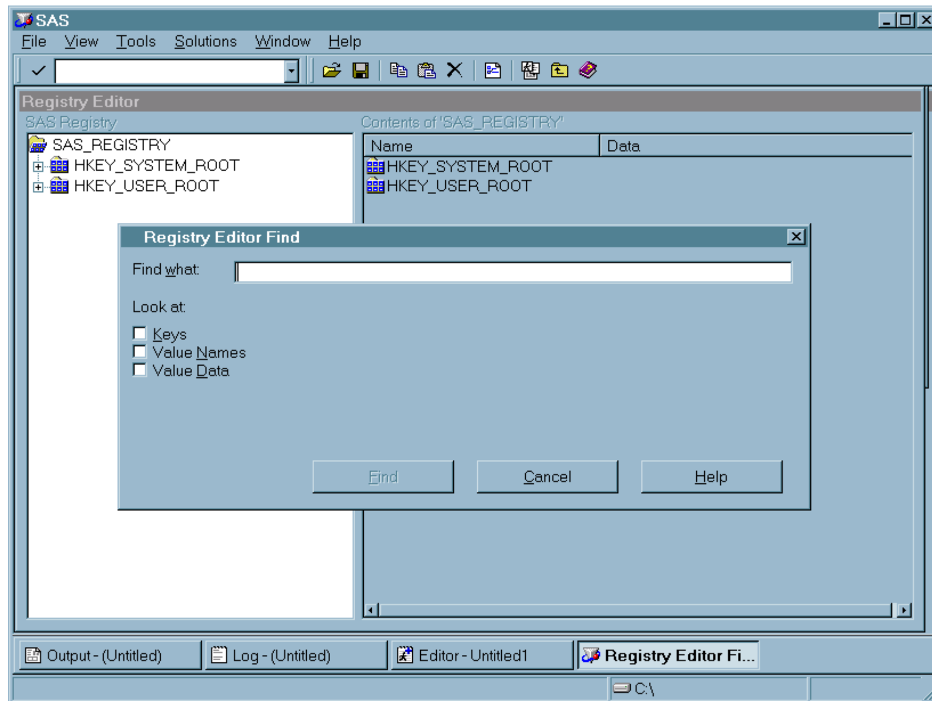
Finding Specific Data in the Registry

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



Changing a Value in the SAS Registry

CAUTION

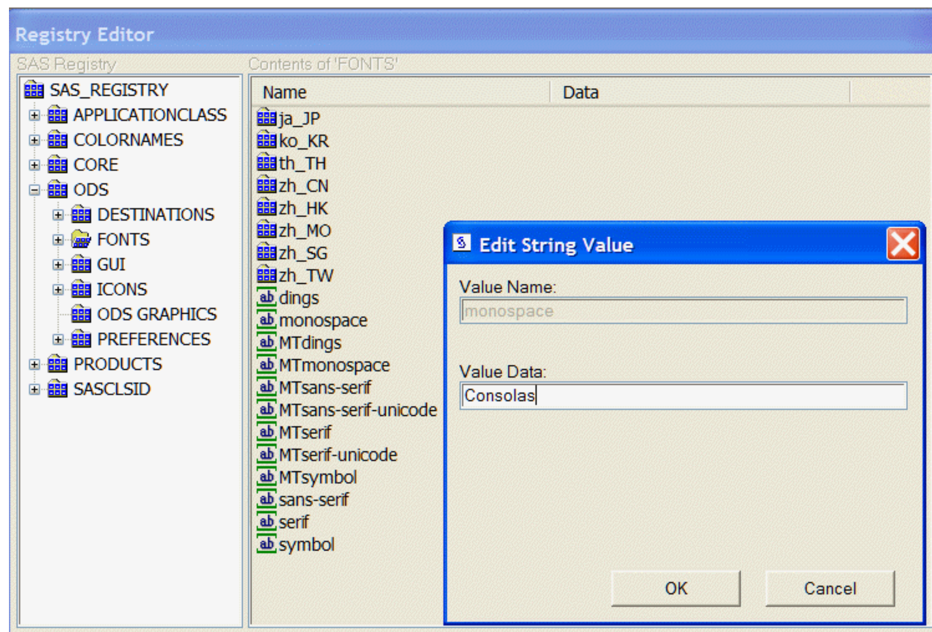
Before modifying registry values, always back up the regstry.sas7bitm file from Sasuser.

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.

Figure 35.3 Example Window for Changing a Value in the SAS Registry



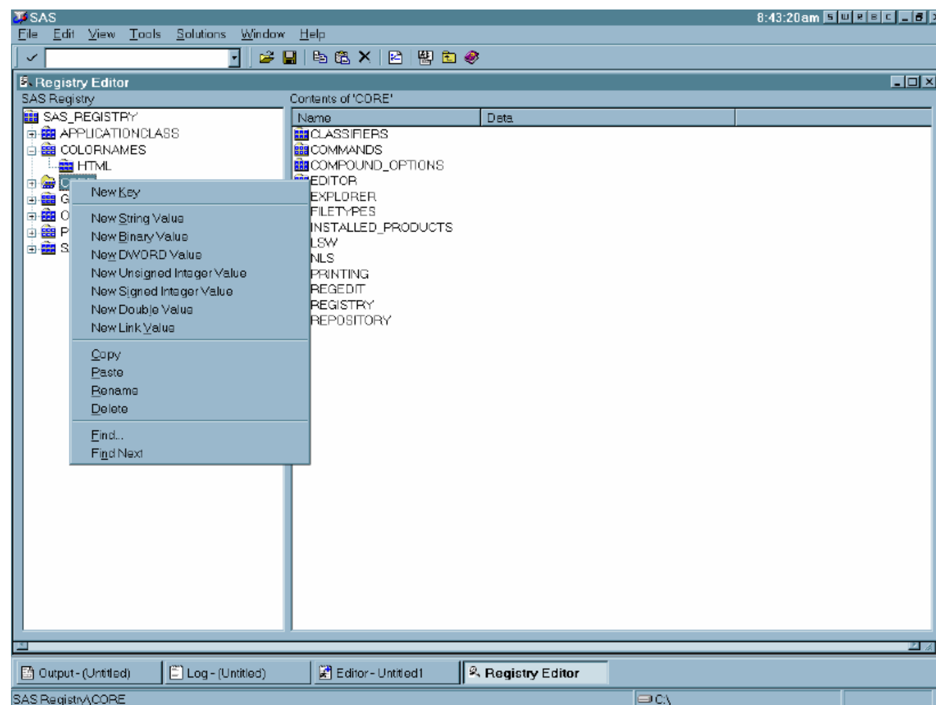
Adding a New Value or Key to the SAS Registry

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.

Figure 35.4 Registry Editor with Pop-up Menu for Adding New Keys and Values



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

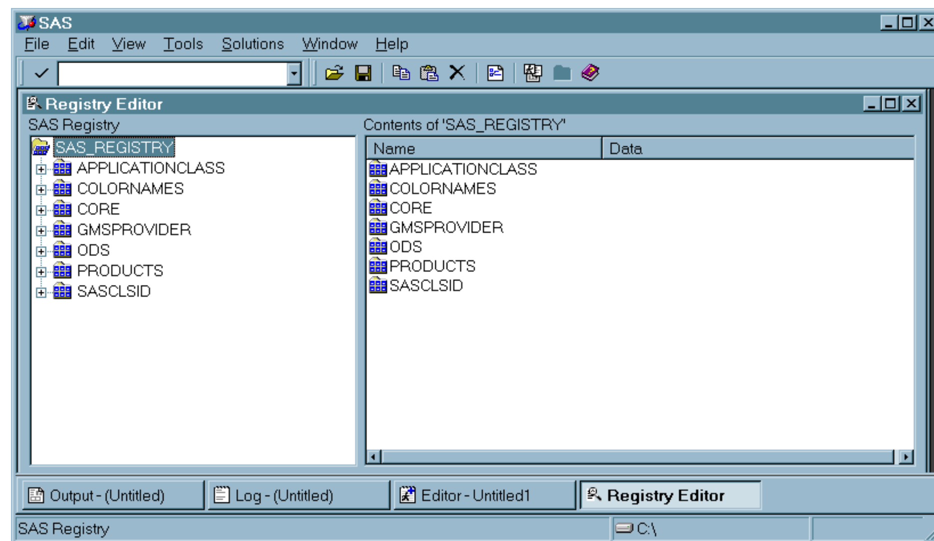
After you open the Registry Editor, you can change your view from the default. The default view shows the registry contents without regard to the storage location. The other registry view displays both Sasuser and Sashelp items in separate trees in the Registry Editor's left pane.

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.

Figure 35.5 The Registry Editor in View Overlay Mode



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

Registry Key	What portion of the Explorer it configures
CORE\EXPLORER\CONFIGURATION	the portions of the Explorer get initialized at startup.
CORE\EXPLORER\MENUS	the pop-up menus that are displayed in the Explorer.
CORE\EXPLORER\KEYEVENTS	the valid key events for the 3270 interface. This key is used only on the mainframe platforms.
CORE\EXPLORER\ICONS	Which icons to display in the Explorer. If the icon value is −1, this causes the icon to be hidden in the Explorer.
CORE\EXPLORER\NEW	This subkey controls what types of objects are available from the **File** ⇨ **New** menu in the Explorer.

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:

```
proc registry import="yourfile.sasxreg" usesashelp;
run;
```

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 `dmlibassign` 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:

```
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`

   or

   `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.

36

The SAS Windowing Environment

What Is the SAS Windowing Environment? 782

Main Windows in the SAS Windowing Environment 782
Overview of SAS Windows 782
SAS Explorer Window 784
Enhanced Editor Window 785
Log Window 786
Results Window 787
Output Window 788

Navigating in the SAS Windowing Environment 792
Overview of SAS Navigation 792
Menus in SAS 792
Toolbars in SAS 796
The Command Line 797

Getting Help in SAS 797
Type Help in the Command Line 797
Open the Help Menu from the Toolbar 798
Click Help in Individual SAS Windows 799

List of SAS Windows and Window Commands 799

Introduction to Managing Your Data in the SAS Windowing Environment 804

Managing Data with SAS Explorer 804
Introduction to Managing Data with SAS Explorer 804
Viewing Libraries and Data Sets 805
Assign File Shortcuts 806
Rename a SAS Data Set 807
Copy or Duplicate a SAS Data Set 807
Sorting Data Sets in a Library 808
View the Properties of a SAS Data Set 808

Working with VIEWTABLE 809
Overview of VIEWTABLE 809
Opening a SAS Data Set in a VIEWTABLE Window 810
Displaying Table Headers as Names or Labels 811
Customizing SAS Explorer for Opening the VIEWTABLE Window 812
Order of Precedence for How Column Headings Are Displayed 814

Mapping the VIEWTABLE Command to a Function Key 815
Temporarily Change Column Headings 815
Move Columns in a Table 817
Sort by Values of a Column 817
Edit Cell Values 819

Subsetting Data By Using the WHERE Expression **820**
Subset Rows of a Table 820
Clear the WHERE Expression 823

Exporting a Subset of Data **824**
Overview of Exporting Data 824
Export Data 824

Importing Data into a Table **827**
Overview of Importing Data 827
Import a Standard File 827
Import a Nonstandard File 829

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.

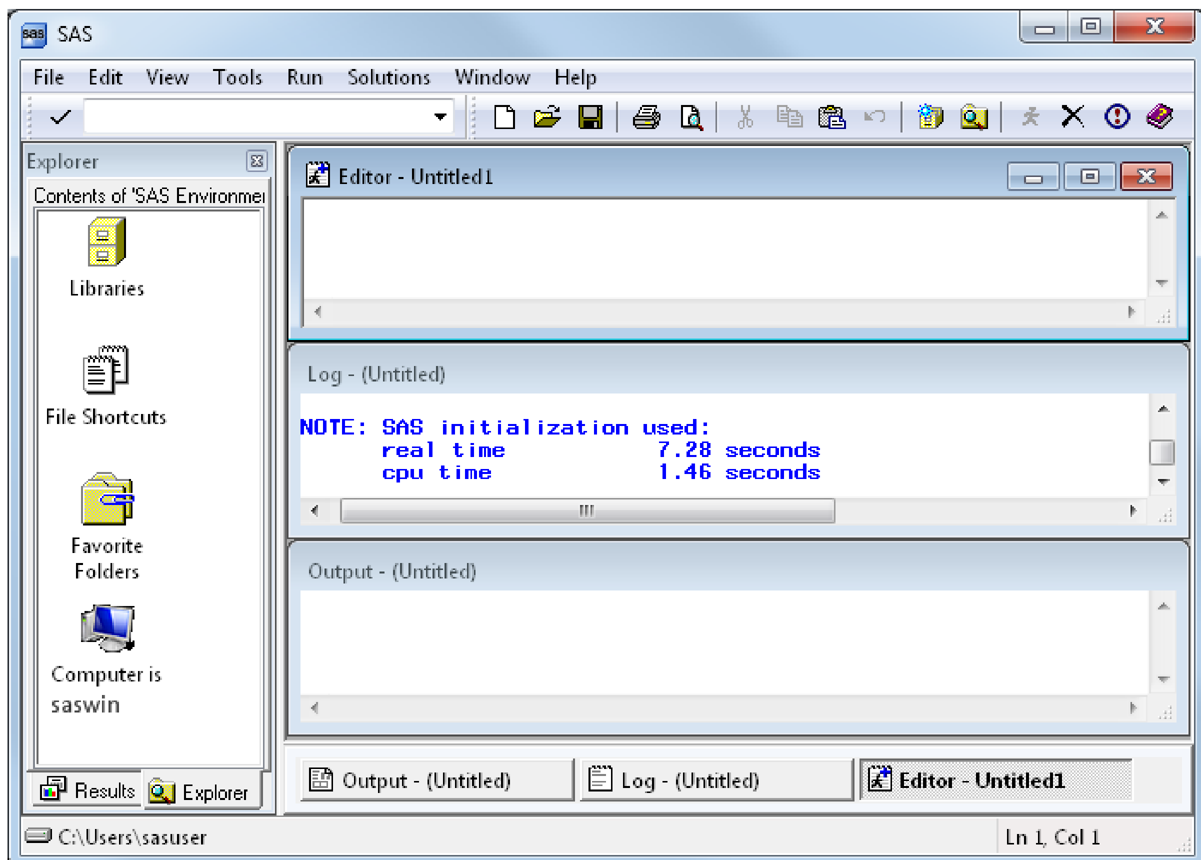
When you first invoke SAS, the Enhanced Editor, Log, Output, and Explorer windows are displayed. When you execute a SAS program, the default output (HTML) is displayed in the Results window. If you use a PUT statement in your program, then output is written to the SAS Log by default.

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

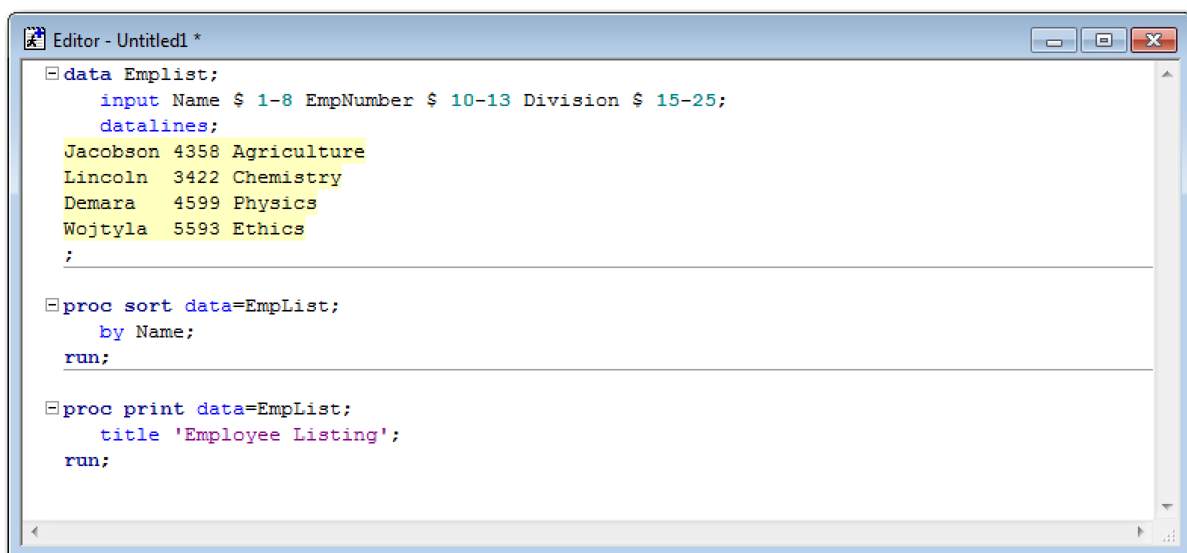
To open the Enhanced Editor window select **View** ⇨ **Enhanced Editor** from the main menu.

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:

Figure 36.2 Example of the Enhanced Editor Window



Note: In the Microsoft Windows operating environment, the Enhanced Editor window appears by default instead of the Program Editor Window. To open the

Program Editor window, follow the same steps for opening the Enhanced Editor window, except select **View** ⇨ **Program Editor** from the main menu. Alternatively, you can enter PROGRAM or PGM in the command line and press Enter.

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

```
Log - (Untitled)
1     data Emplist;
2        input Name $ 1-8 EmpNumber $ 10-13 Division $ 15-25;
3        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


8     ;
9
10    proc sort data=EmpList;
11       by Name;
12    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


13
14    proc print data=EmpList;
NOTE: Writing HTML Body file: sashtml.htm
15       title 'Employee Listing';
16    run;

NOTE: There were 4 observations read from the data set WORK.EMPLIST.
NOTE: PROCEDURE PRINT used (Total process time):
      real time           3.43 seconds
      cpu time            0.85 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:

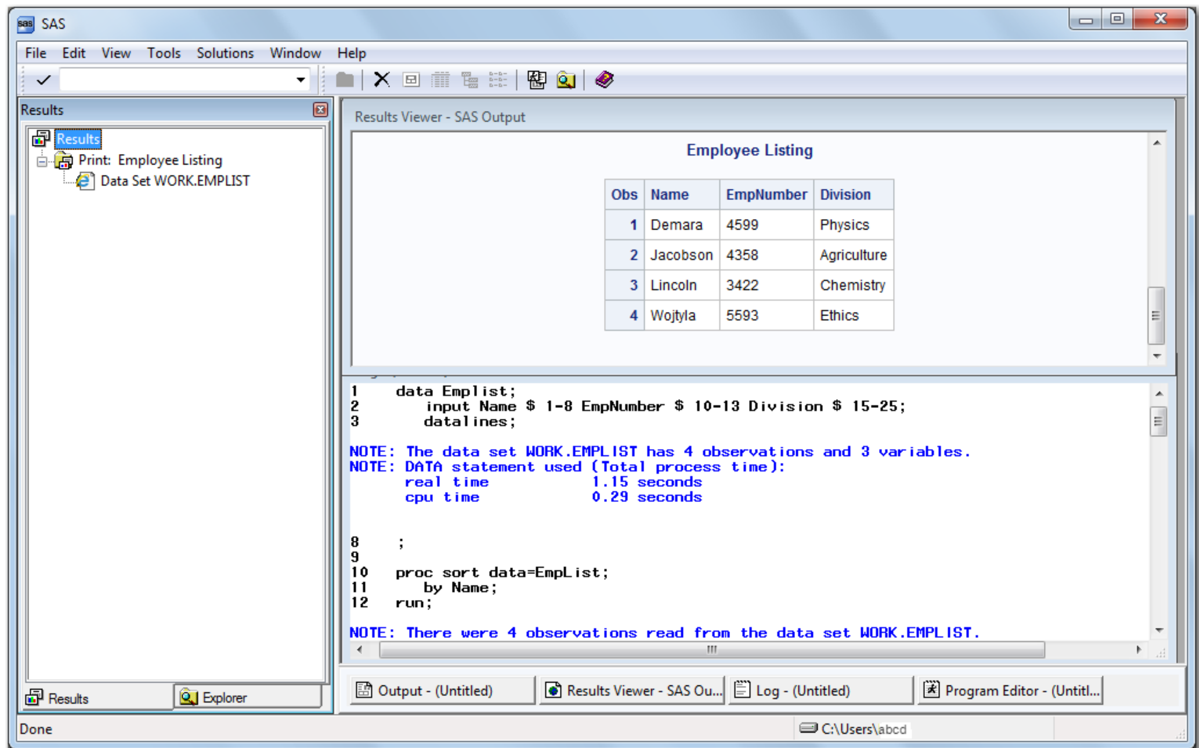
Command:
: Enter ODSRESULTS in the command line and press Enter.

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:

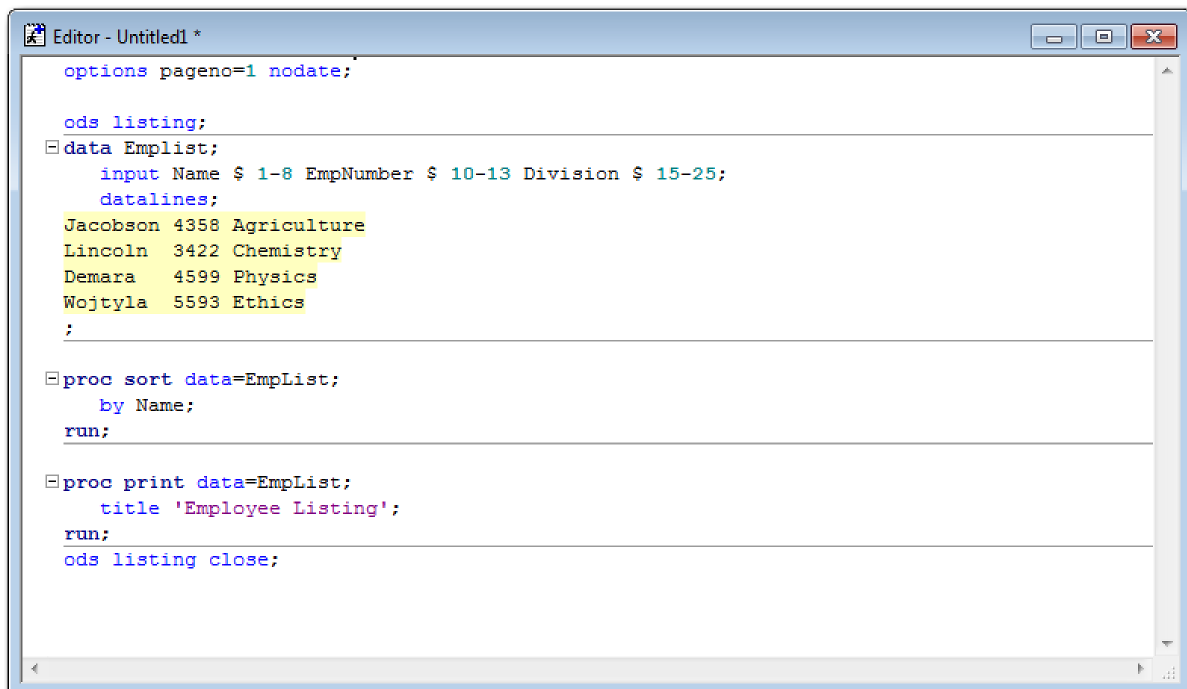
Select **View** ⇨ **Output**.

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 pageno=1 nodate;

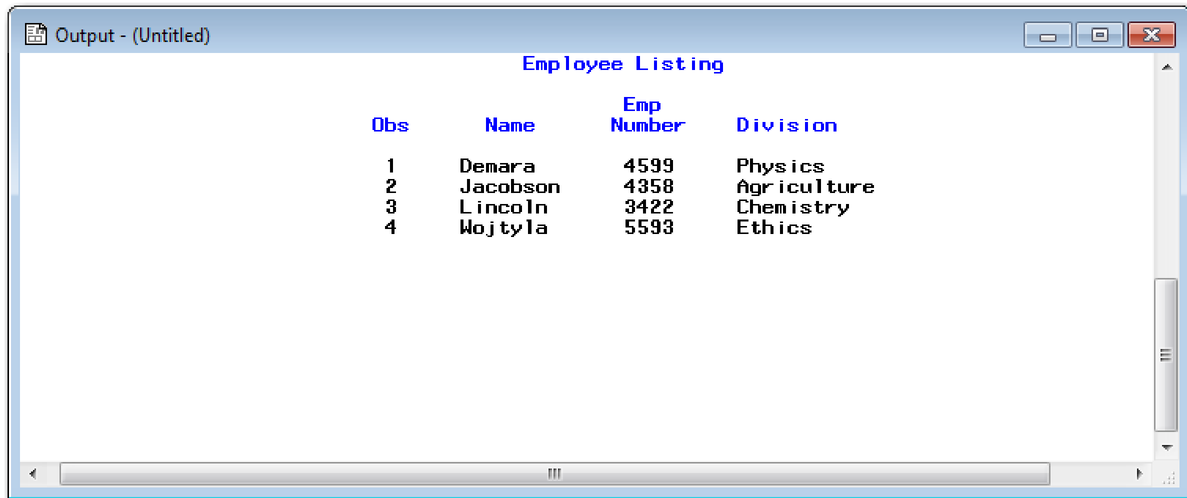
ods listing;
data Emplist;
   input Name $ 1-8 EmpNumber $ 10-13 Division $ 15-25;
   datalines;
Jacobson 4358 Agriculture
Lincoln  3422 Chemistry
Demara   4599 Physics
Wojtyla  5593 Ethics
;

proc sort data=EmpList;
   by Name;
run;

proc print data=EmpList;
   title 'Employee Listing';
run;
ods listing close;
```

SAS creates the following LISTING output:

Figure 36.6 Example of Listing Output in the Output Window

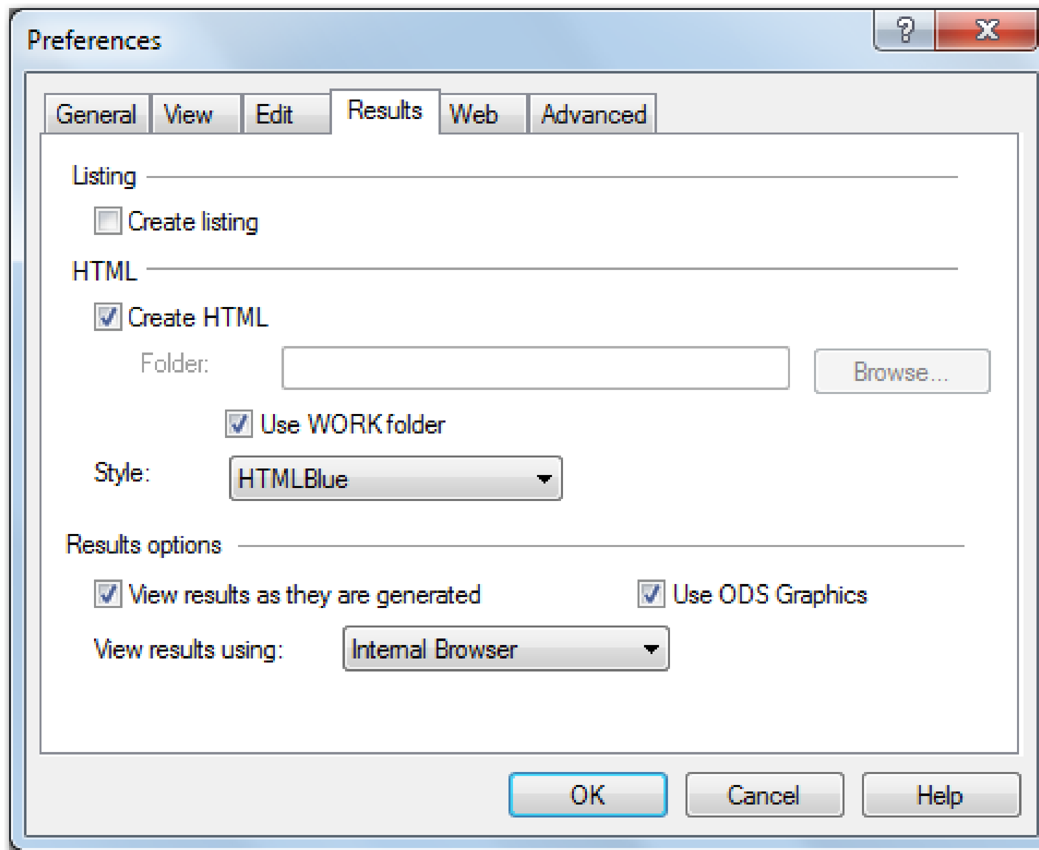


Using the Preferences Dialog Box to Select Output Types

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:

Figure 36.7 Example of the Preferences Dialog Box



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

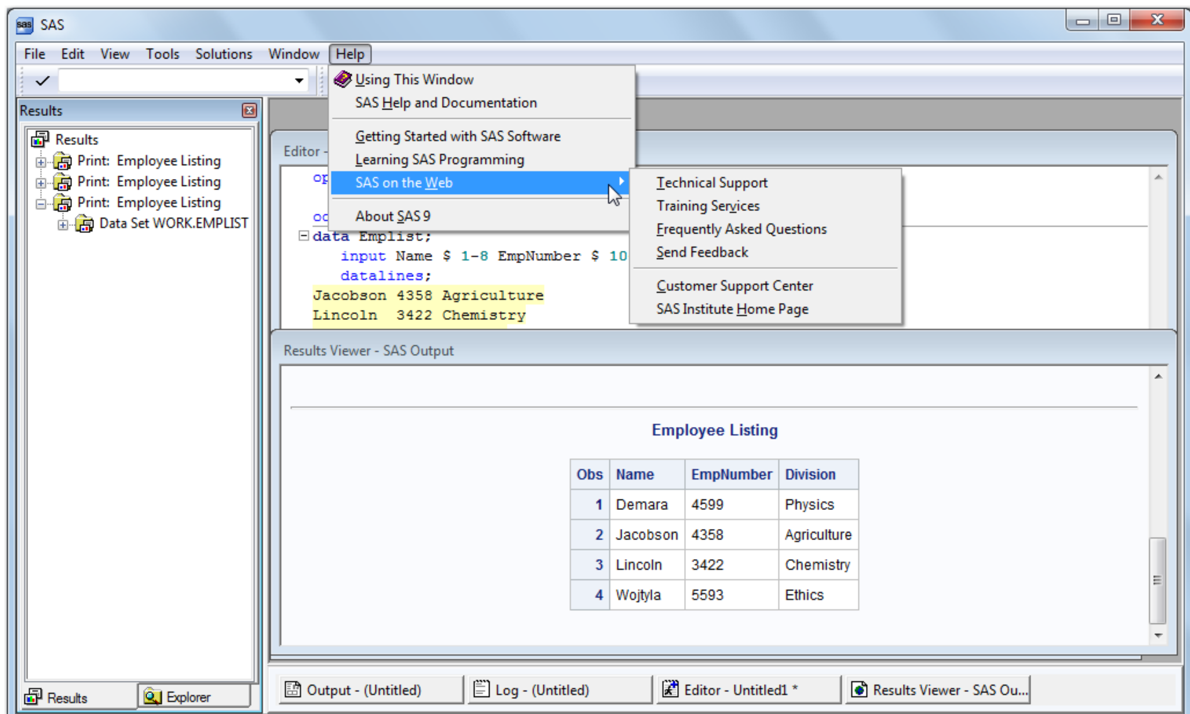
SAS windows have several features that work in a similar manner across all operating environments: menus, toolbars, and online Help. You can customize many of these features by selecting **Tools** ⇨ **Customize** from the menu. For specific information about these features, see the documentation for 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:

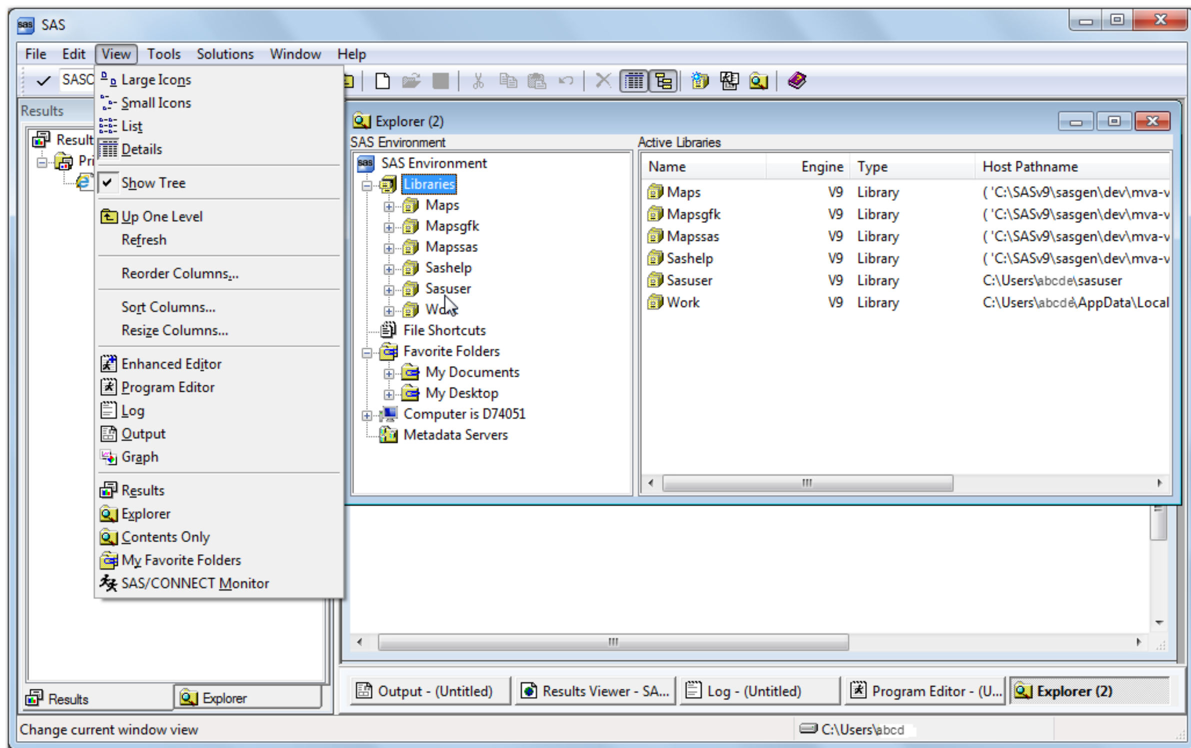
Figure 36.8 The Help Menu



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:

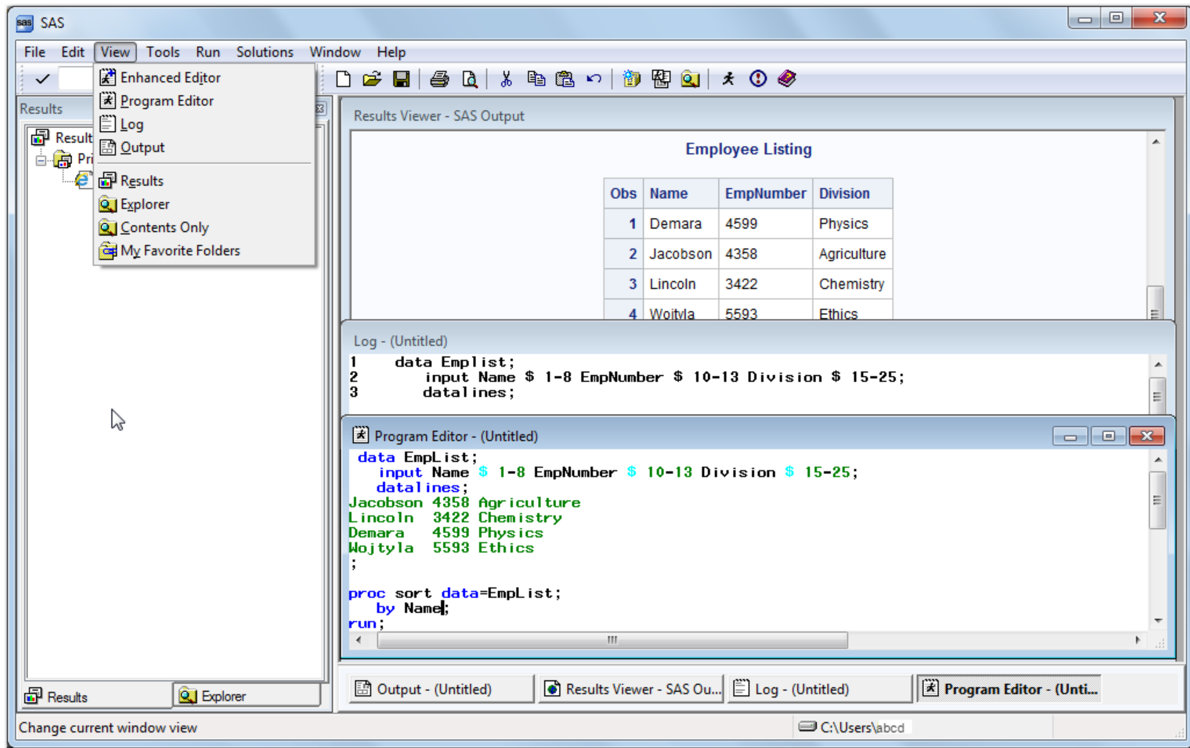
Figure 36.9 View Options When the Explorer Window Is Active



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.

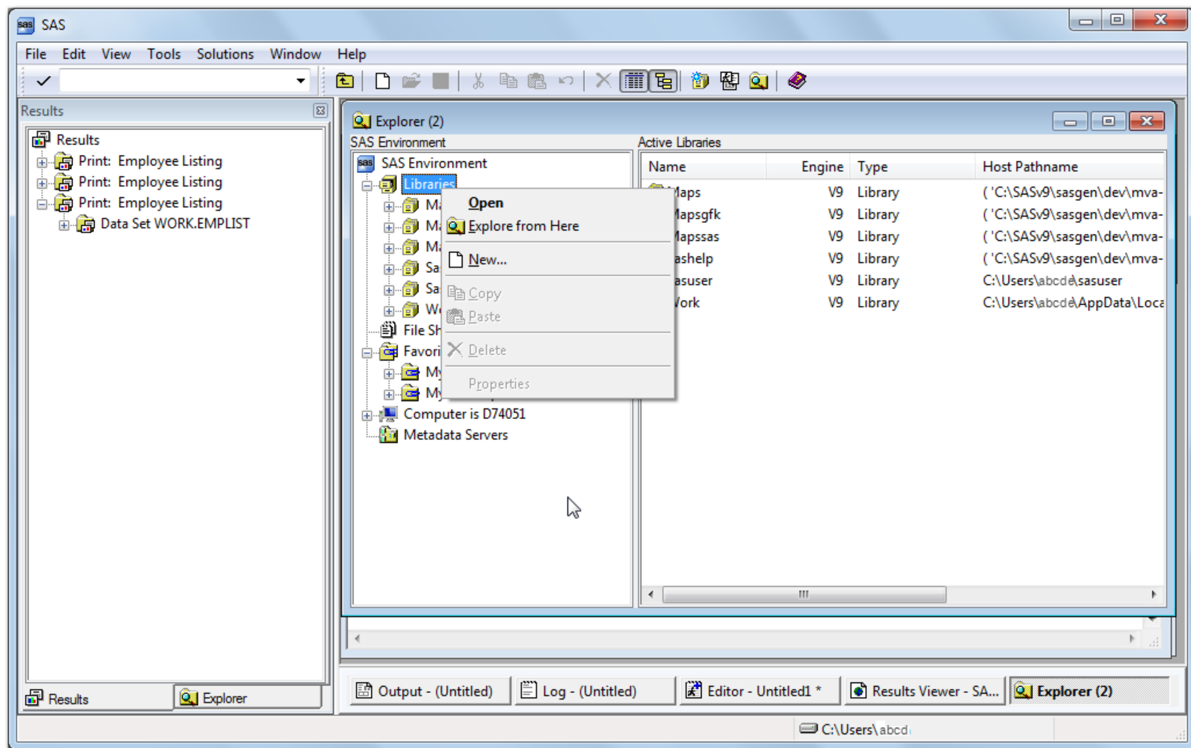
The following display shows the **View** menu when the Program Editor window is active:

Figure 36.10 View Options 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:

Figure 36.11 Another Example of a Menu



The menu remains visible until you make a selection from the menu or until you click an area outside of the menu area.

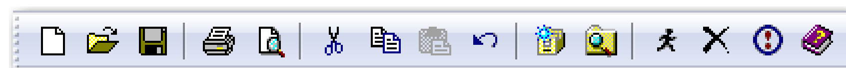
Toolbars in SAS

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



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

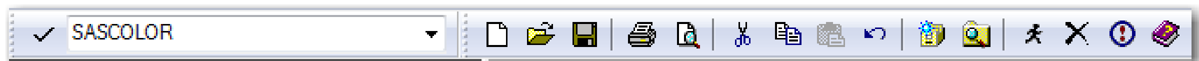
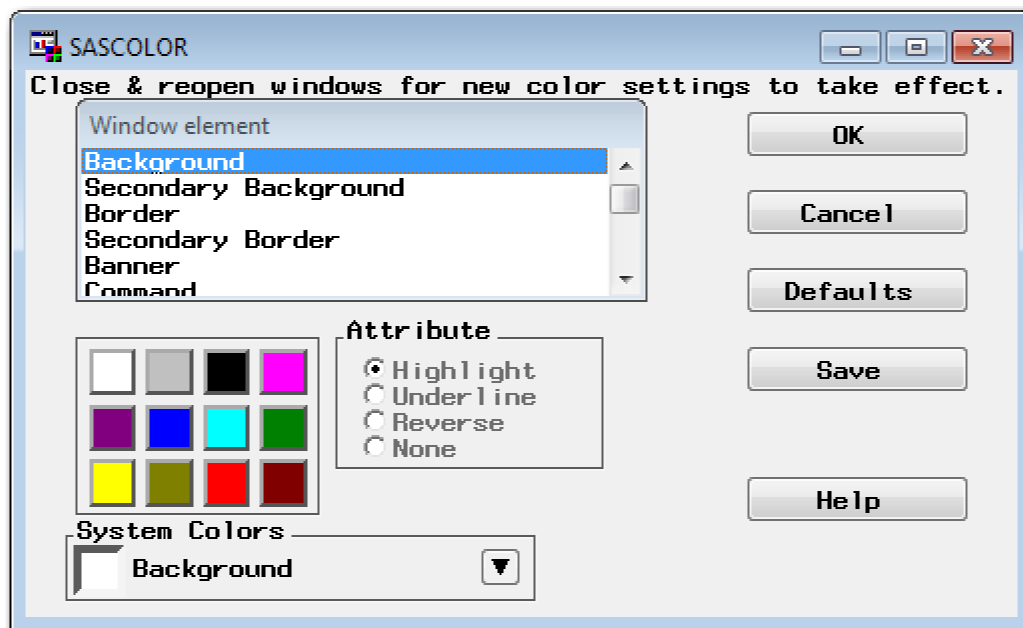


Figure 36.14 The SASCOLOR Window



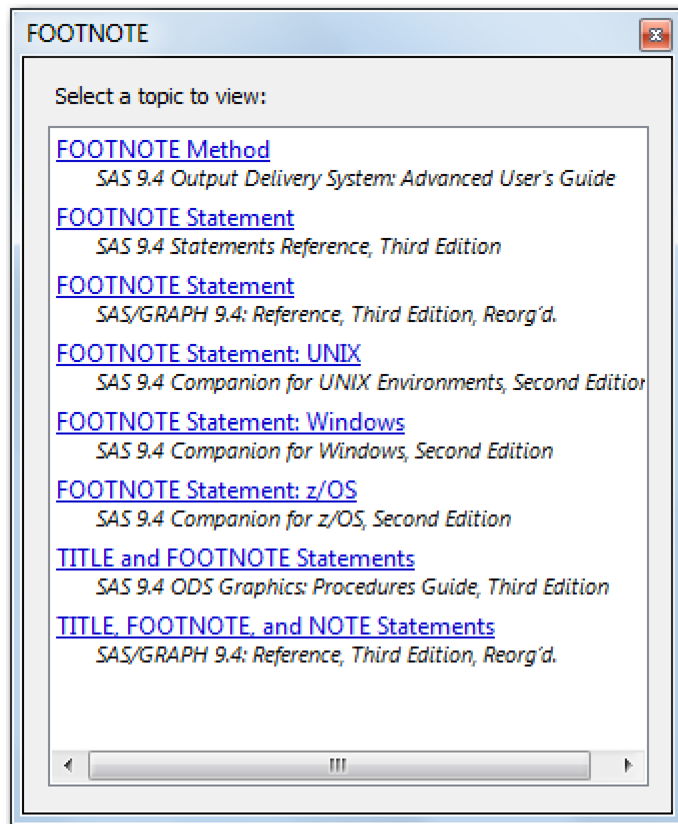
Getting Help in SAS

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:

Figure 36.15 Results of Using Help in the Command Line of a SAS Session



Related items are displayed, along with the documents that contain the information. Click a topic to view Help for that item.

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:

- Contact Technical Support.
- Find information about Training Services.
- Read Frequently Asked Questions (FAQs).
- Send feedback.
- Access the Customer Support Center.
- Browse the SAS Institute home page.

About SAS®9
: provides version and release information about SAS.

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.

Note: You can use the DM statement in SAS to submit window commands as SAS statements when running SAS in SAS Display Manager (DM).

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

Window Name	Description	Window Commands
Documents	Displays your ODS documents in a hierarchical tree structure.	ODSDOCUMENTS
Edit Scheme	Enables you to change the default colors in edit windows.	SYNCONFIG
Explorer	Provides a central access point to data such as catalogs, libraries, data sets, and host files.	ACCESS, BUILD, CATALOG, DIR, EXPLORER, FILENAME, LIBNAME, V6CAT, V6DIR, V6FILENAME, V6LIBNAME
Explorer Options	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.	DMEXPOPTS
File Shortcut Assignment	Assigns a file shortcut to a file using a graphical user interface.	DMFILEASSIGN
Find	Enables you to search for an expression in a SAS library.	EXPFIND
Select Font (operating-environment specific)	Enables you to select a font, font style, and font size.	DLGFONT
FOOTNOTES	Enables you to enter, browse, and modify footnotes for output.	FOOTNOTES
FSBROWSE	Enables you to select a data set for browsing.	FSBROWSE
FSEDIT	Enables you to select a data set to be processed by the FSEDIT procedure.	FSEDIT

Window Name	Description	Window Commands
FSFORM	Enables you to customize a form for sending output to the printer.	FSFORM *formname*
FSLETTER	Enables you to edit or create catalog entries.	FSLETTER
FSLIST	Enables you to browse external files in a SAS session.	FSLIST
FSVIEW	Enables you to browse, edit, or create a SAS data set, displaying the data set as a table with rows and columns.	FSVIEW
HELP	Displays help information about SAS.	HELP
KEYS	Enables you to browse, alter, and save function key settings.	KEYS
Log	Displays messages and SAS statements for the current SAS session.	LOG
Metabase	Accesses the SAS/EIS Metabase Facility to register data, to copy data registrations, and to create, delete, or edit repository files.	METABASE
Metadata Browser	Opens the Metadata Server Configuration dialog box.	METABROWSE (not available on z/OS)
Metafind	Enables you to search for metadata objects in repositories by using Uniform Resource Identifiers (URIs).	METAFIND
Metadata Server Connections	Enables you to import, export, add, remove,	METACON

Window Name	Description	Window Commands
	reorder, and test metadata server connections.	
New Library	Enables you to create a new SAS library and assign a libref.	DMLIBASSIGN
NOTEPAD	Enables you to create and store notepads of text.	NOTEPAD, NOTE, FILEPAD *filename*
Options (SAS system options)	Enables you to view and change some SAS system options.	OPTIONS
Output	Displays procedure output in listing format.	OUTPUT, OUT, LISTING, LIST, LST
Page Setup	Enables you to specify page setup options that apply to Universal Printing jobs.	DMPAGESETUP
Password	Enables you to edit, assign, or clear passwords for a particular data set.	SETPASSWORD (followed by a two-level data set name)
Preferences (operating-environment specific)	Enables you to set or edit SAS system preferences.	DLGPREF
Print	Enables you to print the content of an active SAS window through Universal Printing.	DMPRINT
Print Setup	Enables you to change your default printer, create or edit a printer definition, or delete a printer definition for Universal Printing.	DMPRTSETUP
Program Editor	Enables you to enter, edit, and submit SAS statements and save source files.	PROGRAM, PGM
Properties	Shows details that are associated with the current data set.	VAR *libref.SAS-data-set*, V6VAR *libref.SAS-data-set*

Window Name	Description	Window Commands
SAS Registry Editor	Enables you to edit the SAS registry and to customize aspects of the SAS windowing environment.	REGEDIT
Results	Lists the procedure output that is produced by SAS.	ODSRESULTS
SAS/ACCESS		ACCESS
SAS/AF	Displays windowing applications that are created by SAS/AF software.	AF, AFA
SAS/ASSIST	Displays the primary menu of SAS/ASSIST software, which simplifies the use of SAS.	ASSIST
SASCOLOR	Enables you to change default colors for the different window elements in your SAS windows.	SASCOLOR
SQL QUERY	Enables you to build, run, and save queries without being familiar with Structured Query Language (SQL).	QUERY
SAS System Options	Enables you to change SAS system option settings.	OPTIONS
Templates	Enables you to browse and edit template source code.	ODSTEMPLATES
TITLES	Enables you to enter, browse, and modify titles for output.	TITLES
VIEWTABLE	Enables you to browse, edit, or create tables (data sets).	VIEWTABLE, VT

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.

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.

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.

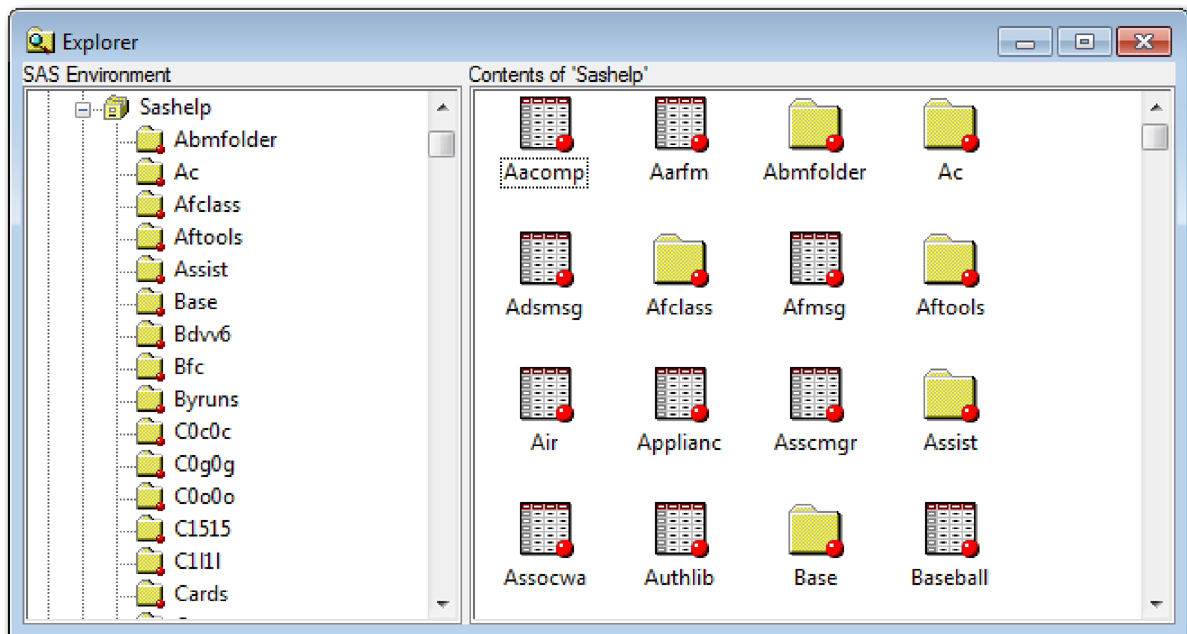
Viewing Libraries and Data Sets

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.

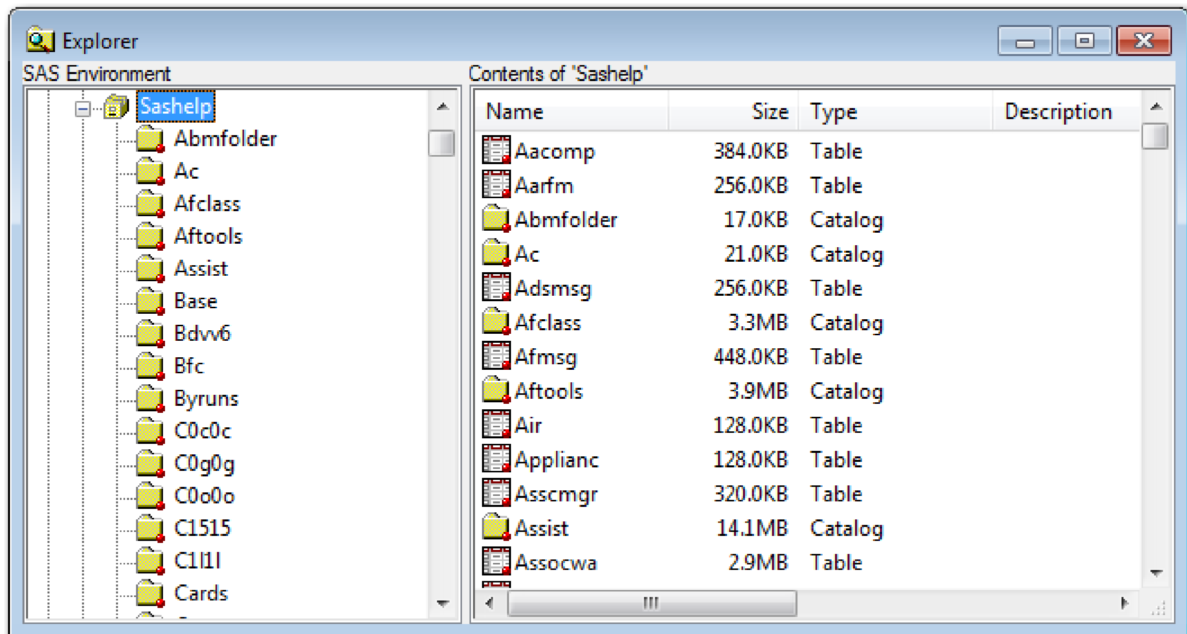
The following example uses large icons to show the contents of Sashelp:

Figure 36.16 Sashelp Library Represented by Large Icons



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:

Figure 36.17 Detailed View of the Sashelp Library



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.

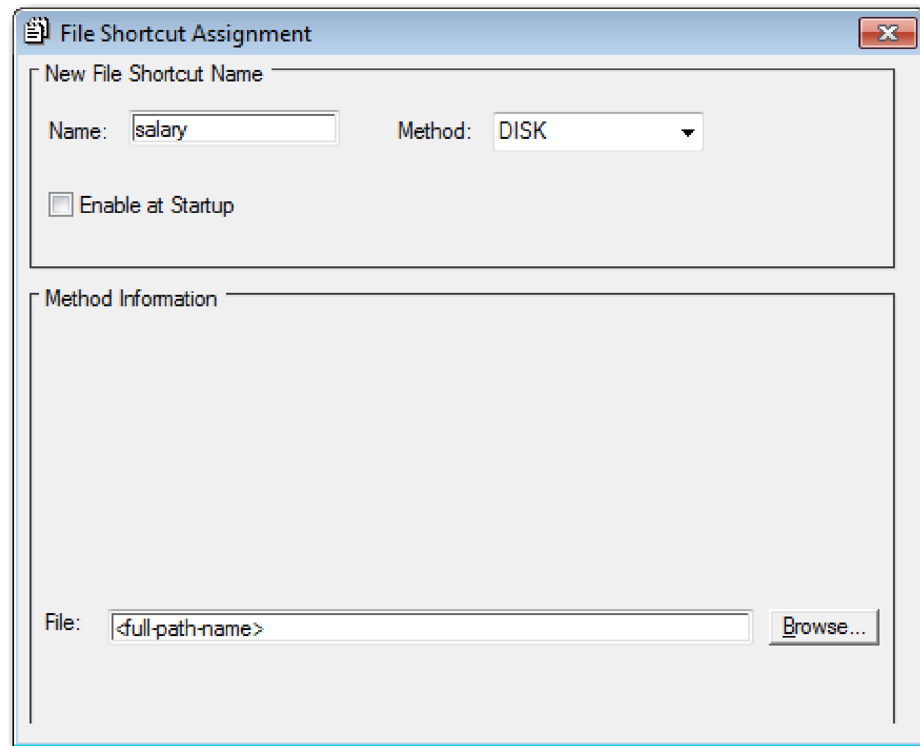
To assign a fileref to a file, follow these steps:

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:

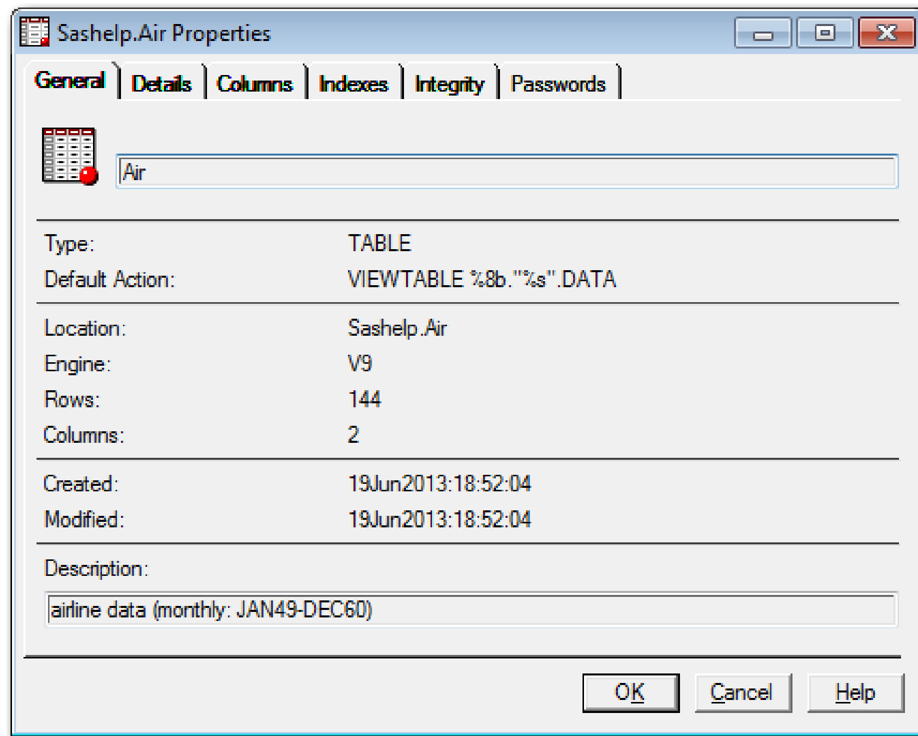
1. Open SAS Explorer and select a library.

   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

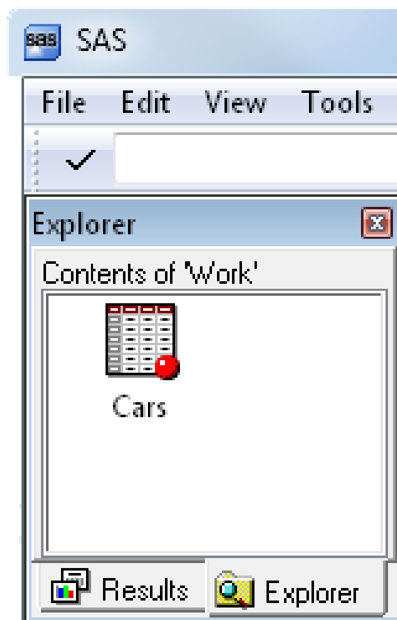
Using the SAS Explorer Window

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.

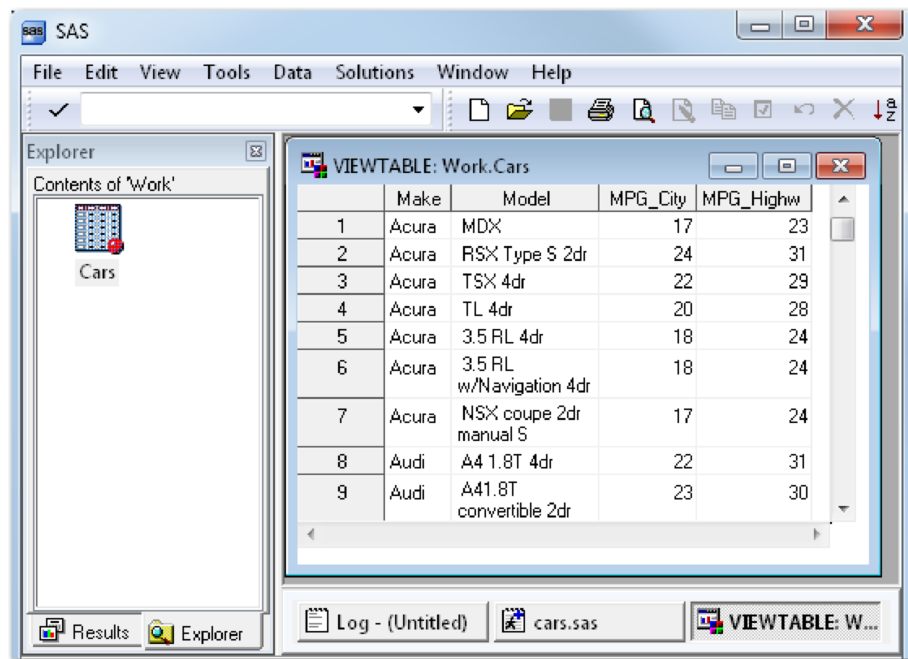
Here are the steps for using the SAS Explorer window to open a SAS data set in a VIEWTABLE window:

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.



4 Use the scroll bar on the VIEWTABLE window to view all of the data.

Using the VIEWTABLE Command

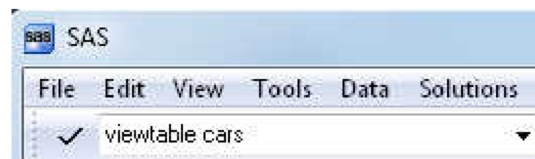
You can also open a data set in a VIEWTABLE window by using the VIEWTABLE command in the SAS Display Manager command line.

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:

```
viewtable cars
```



Displaying Table Headers as Names or Labels

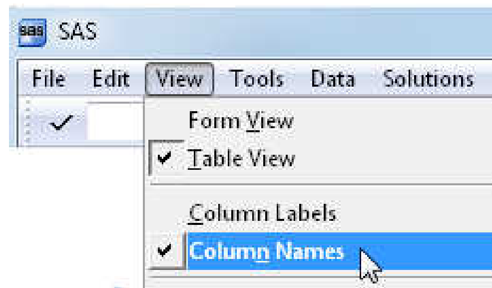
When you open a data set that contains labels in a VIEWTABLE window, SAS automatically displays the table headers as variable labels rather than the variable names. You can change the way SAS displays table headers by using the VIEWTABLE pop-up menu or by using the VIEWTABLE command.

- 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.

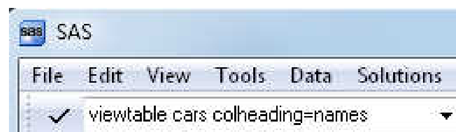
- Using the VIEWTABLE command to change the way table headers are displayed when a table is opened:

  1 Specify the COLHEADING= option on the VIEWTABLE command in the SAS command line using the following syntax.

```
VIEWTABLE data-set-name  -<COLHEADING>=NAMES | LABELS>
```

  2 Here is an example:

```
viewtable cars colheading=names
```

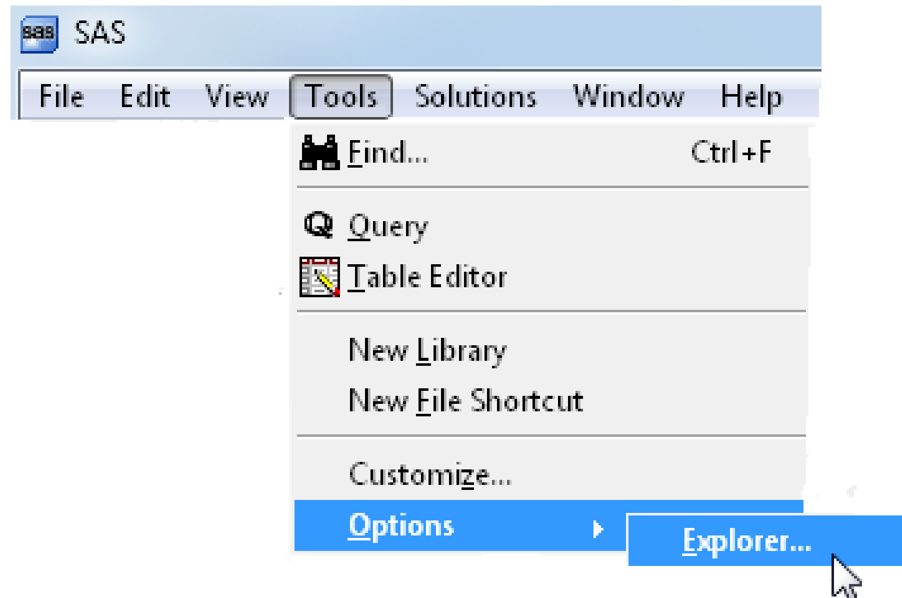


Customizing SAS Explorer for Opening the VIEWTABLE Window

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.



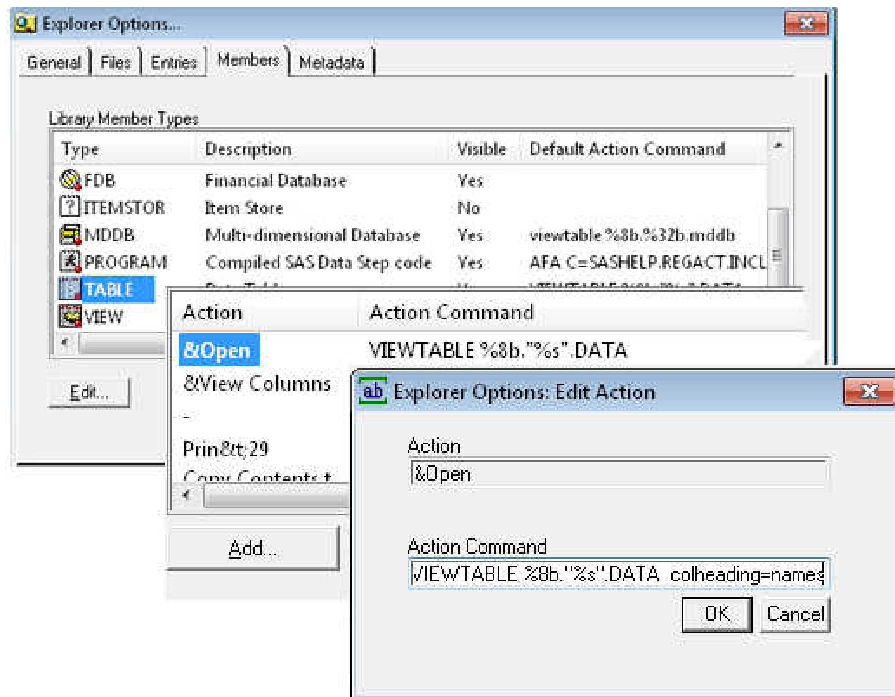
2 Select the **Members** tab.

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:

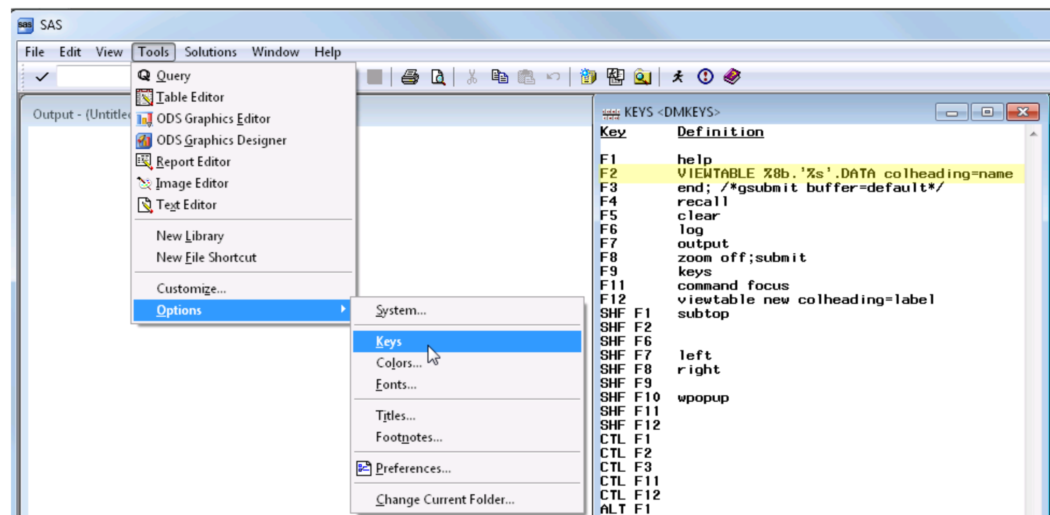
1 Select **Tools** ⇨ **Options** ⇨ **Keys** from the SAS menu. The Keys window will appear.

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. '%s'.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).

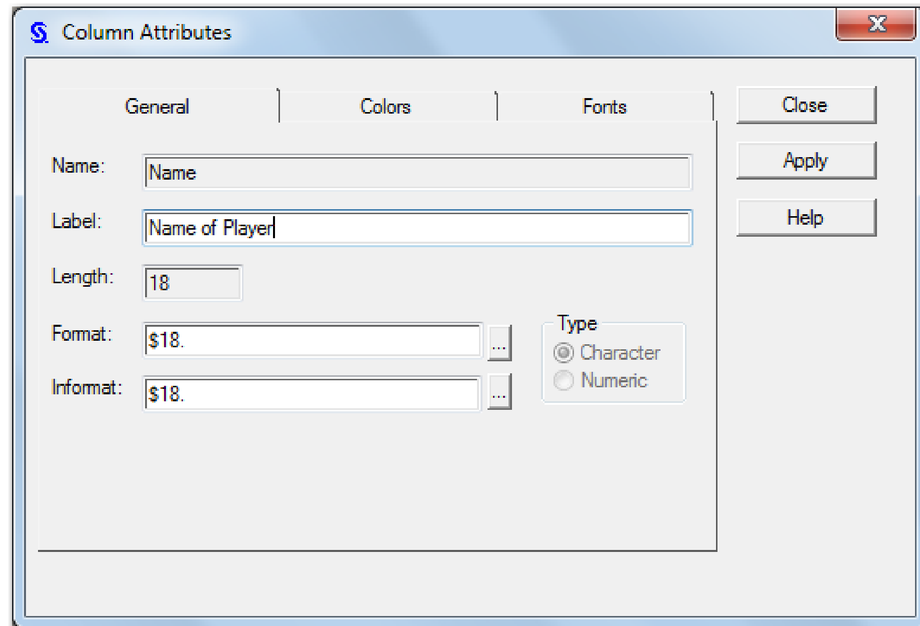
Temporarily Change Column Headings

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.

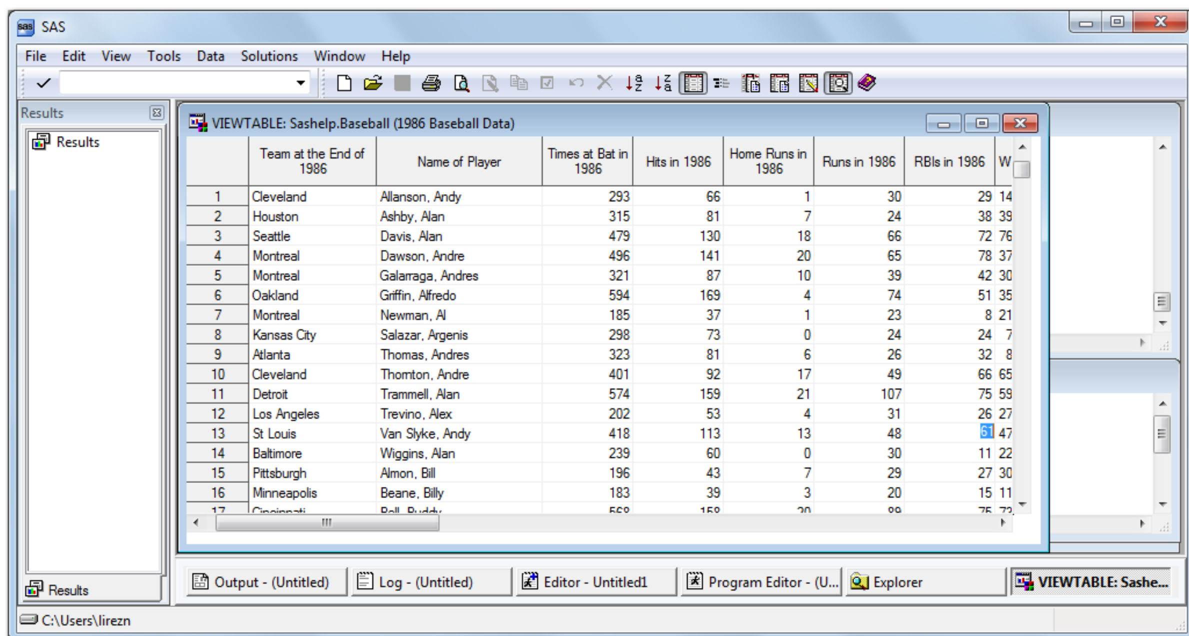
2 In the **Label** field of the Column Attributes window, enter the new name of the column heading and then click **Apply**.

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**.



	Team at the End of 1986	Name of Player	Times at Bat in 1986	Hits in 1986	Home Runs in 1986	Runs in 1986	RBIs in 1986	W
1	Cleveland	Allanson, Andy	293	66	1	30	29	14
2	Houston	Ashby, Alan	315	81	7	24	38	39
3	Seattle	Davis, Alan	479	130	18	66	72	76
4	Montreal	Dawson, Andre	496	141	20	65	78	37
5	Montreal	Galarraga, Andres	321	87	10	39	42	30
6	Oakland	Griffin, Alfredo	594	169	4	74	51	35
7	Montreal	Newman, Al	185	37	1	23	8	21
8	Kansas City	Salazar, Argenis	298	73	0	24	24	7
9	Atlanta	Thomas, Andres	323	81	6	26	32	8
10	Cleveland	Thornton, Andre	401	92	17	49	66	65
11	Detroit	Trammell, Alan	574	159	21	107	75	59
12	Los Angeles	Trevino, Alex	202	53	4	31	26	27
13	St Louis	Van Slyke, Andy	418	113	13	48	61	47
14	Baltimore	Wiggins, Alan	239	60	0	30	11	22
15	Pittsburgh	Almon, Bill	196	43	7	29	27	30
16	Minneapolis	Beane, Billy	183	39	3	20	15	11

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.

VIEWTABLE: Sashelp.Baseball (1986 Baseball Data)

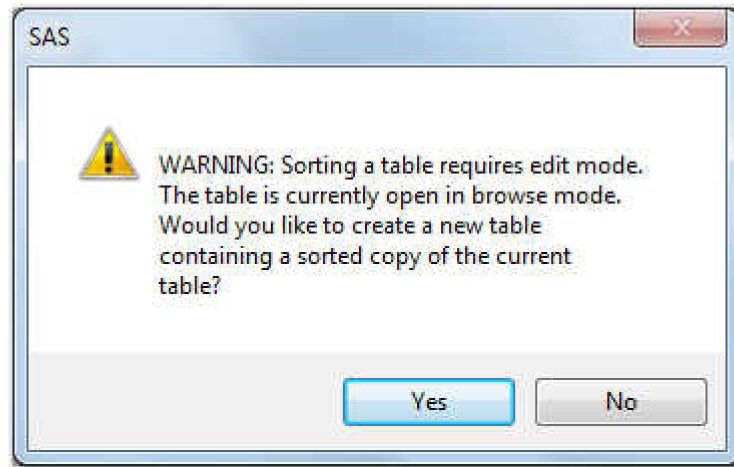
	Team at the End of 1986	Name	Times at Bat in 1986	Hits in 1986	Home Runs in 1986	Runs in 1986	RBIs in 1986	W
1	Cleveland	Allanson, Andy	293	66	1	30	29	14
2	Houston	Ashby, Alan	315	81	7	24	38	39
3	Seattle	Davis, Alan	479	130	18	66	72	76
4	Montreal	Dawson, Andre	496	141	20	65	78	37
5	Montreal	Galarraga, Andres	321	87	10	39	42	30
6	Oakland	Griffin, Alfredo	594	169	4	74	51	35
7	Montreal	Newman, Al	185	37	1	23	8	21
8	Kansas City	Salazar, Argenis	298	73	0	24	24	7
9	Atlanta	Thomas, Andres	323	81	6	26	32	8
10	Cleveland	Thornton, Andre	401	92	17	49	66	65
11	Detroit	Trammell, Alan	574	159	21	107	75	59
12	Los Angeles	Trevino, Alex	202	53	4	31	26	27
13	St Louis	Van Slyke, Andy	418	113	13	48	61	47
14	Baltimore	Wiggins, Alan	239	60	0	30	11	22
15	Pittsburgh	Almon, Bill	196	43	7	29	27	30
16	Minneapolis	Beane, Billy	183	39	3	20	15	11

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:

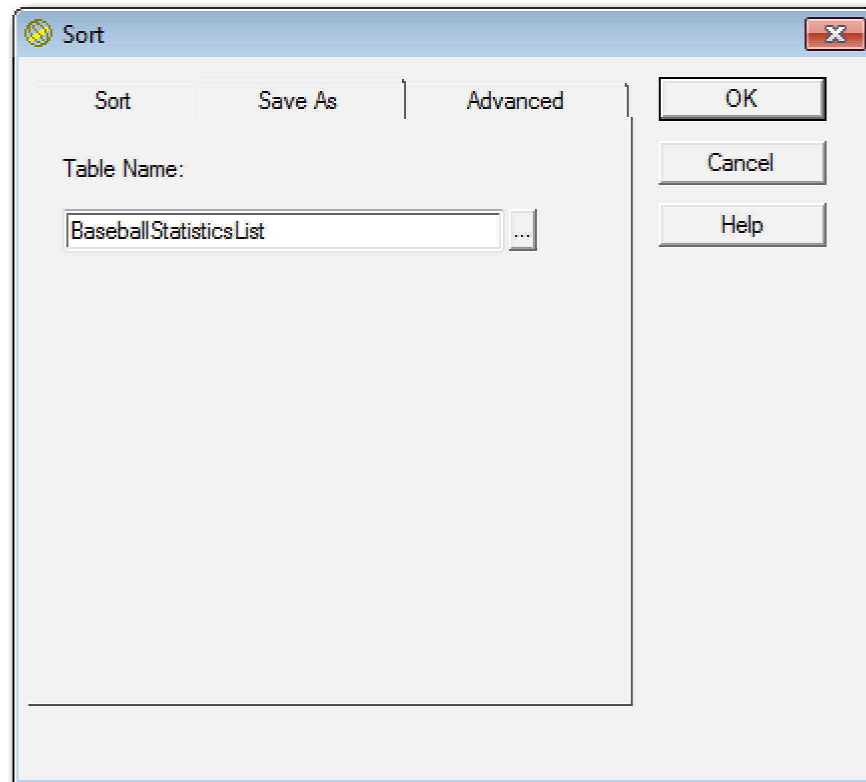
1. Right-click the heading of the column on which you want to sort, and select **Sort** from the menu.
2. Select **Ascending** or **Descending** from the menu.
3. When the following warning message appears, click **Yes** to create a sorted copy of the table.



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**.

VIEWTABLE: BaseballStatisticsList (1986 Baseball Data)

	Team at the End of 1986	Player's Name	Times at Bat in 1986	Hits in 1986	Home Runs in 1986	Runs in 1986	RBIs in 1986	Walks in 1986	Years Major
1	Atlanta	Thomas, Andres	323	81	6	26	32	8	2
2	Atlanta	Homer, Bob	517	141	27	70	87	52	9
3	Atlanta	Sample, Billy	200	57	6	23	14	14	9
4	Atlanta	Murphy, Dale	614	163	29	89	83	75	11
5	Atlanta	Hubbard, Glenn	408	94	4	42	36	66	9
6	Atlanta	Oberkfell, Ken	503	136	5	62	48	83	10
7	Atlanta	Moreno, Omar	359	84	4	46	27	21	12
8	Atlanta	Virgil, Ozzie	359	80	15	45	48	63	7
9	Atlanta	Ramirez, Rafael	496	119	8	57	33	21	7
10	Atlanta	Harper, Terry	265	68	8	26	30	29	7
11	Atlanta	Simmons, Ted	127	32	4	14	25	12	19
12	Baltimore	Wiggins, Alan	239	60	0	30	11	22	6
13	Baltimore	Ripken, Cal	627	177	25	98	81	70	6
14	Baltimore	Murray, Eddie	495	151	17	61	84	78	10
15	Baltimore	Lynn, Fred	397	114	23	67	67	53	13
16	Baltimore	Rayford, Floyd	210	37	8	15	19	15	6
17	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

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.

VIEWTABLE: BaseballStatisticsList (1986 Baseball Data)

	Team at the End of 1986	Player's Name	Times at Bat in 1986	Hits in 1986	Home Runs in 1986	Runs in 1986	RBIs in 1986	Walks in 1986	Years Major
1	Atlanta	Thomas, Andres	323	81	6	26	32	8	2
2	Atlanta	Homer, Bob	517	141	27	70	87	52	9
3	Atlanta	Sample, Billy	200	57	6	23	14	14	9
4	Atlanta	Murphy, Dale	614	163	29	89	83	75	11
5	Atlanta	Hubbard, Glenn	408	94	4	42	36	66	9
6	Atlanta	Oberkfell, Ken	503	136	5	62	48	83	10
7	Atlanta	Moreno, Omar	359	84	4	46	27	21	12
8	Atlanta	Virgil, Ozzie	359	80	15	45	48	63	7
9	Atlanta	Ramirez, Rafael	496	119	8	57	33	21	7
10	Atlanta	Harper, Terry	265	68	8	26	30	29	7
11	Atlanta	Simmons, Ted	127	32	4	14	25	12	19
12	Baltimore	Wiggins, Alan	239	60	0	30	11	22	6
13	Baltimore	Ripken, Cal	627	177	25	98	81	70	6
14	Baltimore	Murray, Eddie	495	151	17	61	84	78	10
15	Baltimore	Lynn, Fred	397	114	23	67	67	53	13
16	Baltimore	Rayford, Floyd	210	37	8	15	19	15	6
17	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

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**.

VIEWTABLE: BaseballStatisticsList (1986 Baseball Data)

	Team at the End of 1986	Player's Name	Times at Bat in 1986	Hits in 1986	Home Runs in 1986	Runs in 1986	RBIs in 1986	Walks in 1986	Years Major
1	Atlanta	Thomas, Andres	323	81	6	26	32	8	2
2	Atlanta	Horner, Bob	517	141	27	70	87	52	9
3	Atlanta	Sample, Billy	200	57	6	23	14	14	9
4	Atlanta	Murphy, Dale	614	163	29	89	83	75	11
5	Atlanta	Hubbard, Glenn	500	94	4	42	36	66	9
6	Atlanta	Oberkfell, Ken	503	136	5	62	48	83	10
7	Atlanta	Moreno, Omar	359	84	4	46	27	21	12
8	Atlanta	Virgil, Ozzie	359	80	15	45	48	63	7
9	Atlanta	Ramirez, Rafael	496	119	8	57	33	21	7
10	Atlanta	Harper, Terry	265	68	8	26	30	29	7
11	Atlanta	Simmons, Ted	127	32	4	14	25	12	19
12	Baltimore	Wiggins, Alan	239	60	0	30	11	22	6
13	Baltimore	Ripken, Cal	627	177	25	98	81	70	6
14	Baltimore	Murray, Eddie	495	151	17	61	84	78	10
15	Baltimore	Lynn, Fred	397	114	23	67	67	53	13
16	Baltimore	Rayford, Floyd	210	37	8	15	19	15	6

4 Select **File** ⇨ **Close** from the **File** menu.

5 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** ⇨ **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.

VIEWTABLE: Sashelp.Cars (2004 Car Data)

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	Engine Size (L)	Cylinders	H
1	Acura	MDX	SUV	Asia	All	$36,945	$33,337	3.5	6	
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21,761	2	4	
3	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24,647	2.4	4	
4	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30,299	3.2	6	
5	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39,014	3.5	6	
6	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41,100	3.5	6	
7	Acura	NSX coupe 2dr manual S	Sports	Asia	Rear	$89,765	$79,978	3.2	6	
8	Audi	A4 1.8T 4dr	Sedan	Europe	Front	$25,940	$23,508	1.8	4	
9	Audi	A41.8T convertible 2dr	Sedan	Europe	Front	$35,940	$32,506	1.8	4	
10	Audi	A4 3.0 4dr	Sedan	Europe	Front	$31,840	$28,846	3	6	
11	Audi	A4 3.0 Quattro 4dr manual	Sedan	Europe	All	$33,430	$30,366	3	6	
12	Audi	A4 3.0 Quattro 4dr auto	Sedan	Europe	All	$34,480	$31,388	3	6	
13	Audi	A6 3.0 4dr	Sedan	Europe	Front	$36,640	$33,129	3	6	
14	Audi	A6 3.0 Quattro 4dr	Sedan	Europe	All	$39,640	$35,992	3	6	
15	Audi	A4 3.0 convertible 2dr	Sedan	Europe	Front	$42,490	$38,325	3	6	
16	Audi	A4 3.0 Quattro convertible 2dr	Sedan	Europe	All	$44,240	$40,075	3	6	
17	Audi	A6 2.7 Turbo Quattro 4dr	Sedan	Europe	All	$42,840	$38,840	2.7	6	

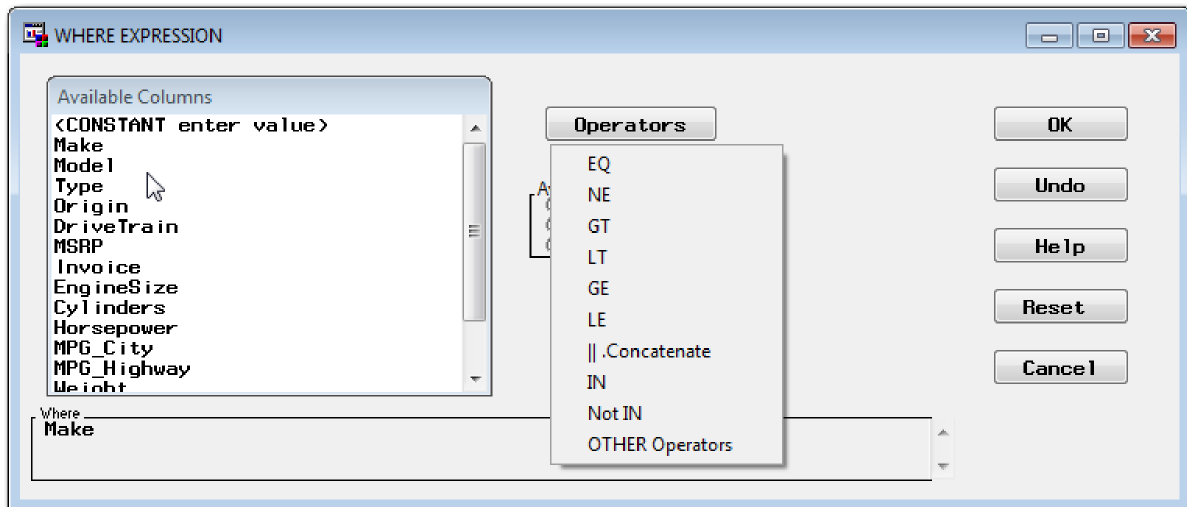
2 Right-click any table cell that is not a heading and select **Where** from the menu.

The WHERE EXPRESSION window appears.

```
WHERE EXPRESSION
Available Columns
<CONSTANT enter value>
Make
Model
Type
Origin
DriveTrain
MSRP
Invoice
EngineSize
Cylinders
Horsepower
MPG_City
MPG_Highway
Weight
Operators
Available Columns Sort
(•) None
( ) Ascending
( ) Descending
OK
Undo
Help
Reset
Cancel
Where
```

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.

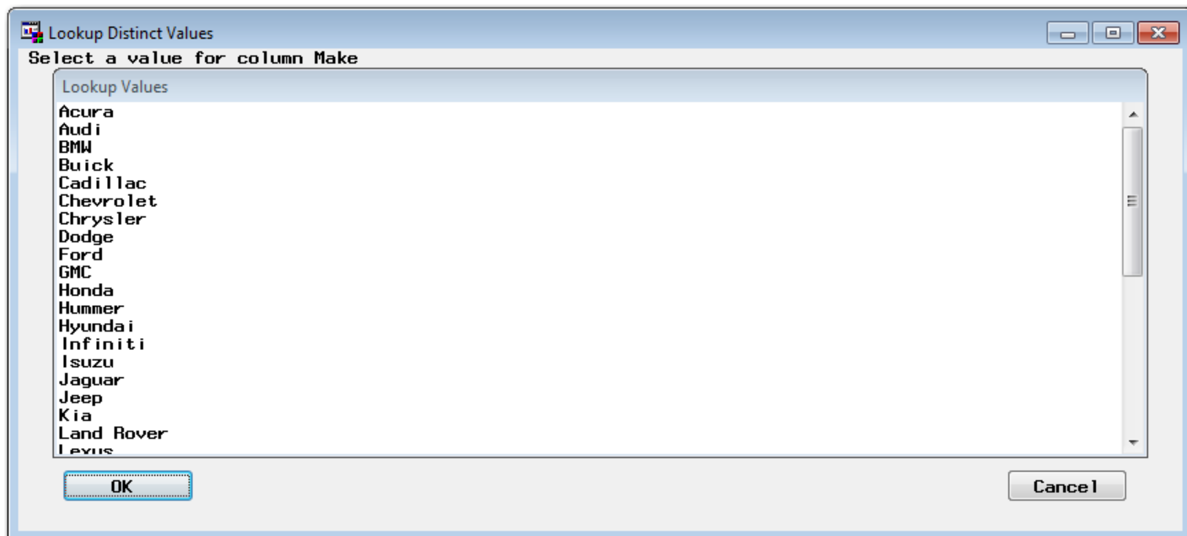


4 In the **Available Columns** list, select another value to complete the WHERE expression.

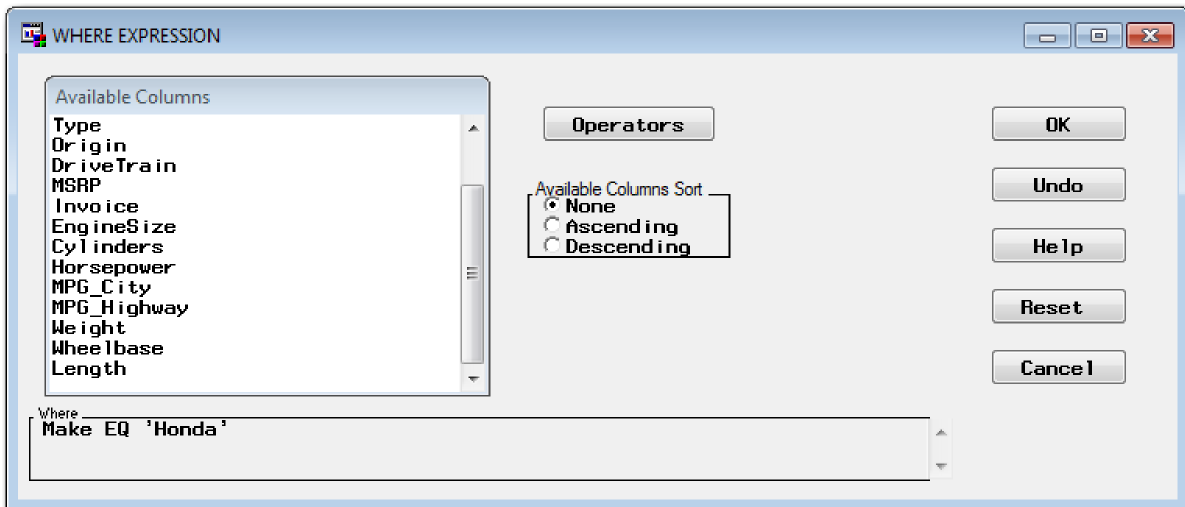
In this example, scroll to the bottom of the **Available Columns** window and select **<LOOKUP distinct values>**.

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.

VIEWTABLE: Sashelp.Cars (2004 Car Data)

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	Engine Size (L)	Cylinders	H
150	Honda	Civic Hybrid 4dr manual (gas/electric)	Hybrid	Asia	Front	$20,140	$18,451	1.4	4	
151	Honda	Insight 2dr (gas/electric)	Hybrid	Asia	Front	$19,110	$17,911	2	3	
152	Honda	Pilot LX	SUV	Asia	All	$27,560	$24,843	3.5	6	
153	Honda	CR-V LX	SUV	Asia	All	$19,860	$18,419	2.4	4	
154	Honda	Element LX	SUV	Asia	All	$18,690	$17,334	2.4	4	
155	Honda	Civic DX 2dr	Sedan	Asia	Front	$13,270	$12,175	1.7	4	
156	Honda	Civic HX 2dr	Sedan	Asia	Front	$14,170	$12,996	1.7	4	
157	Honda	Civic LX 4dr	Sedan	Asia	Front	$15,850	$14,531	1.7	4	
158	Honda	Accord LX 2dr	Sedan	Asia	Front	$19,860	$17,924	2.4	4	
159	Honda	Accord EX 2dr	Sedan	Asia	Front	$22,260	$20,080	2.4	4	
160	Honda	Civic EX 4dr	Sedan	Asia	Front	$17,750	$16,265	1.7	4	
161	Honda	Civic Si 2dr hatch	Sedan	Asia	Front	$19,490	$17,849	2	4	
162	Honda	Accord LX V6 4dr	Sedan	Asia	Front	$23,760	$21,428	3	6	
163	Honda	Accord EX V6 2dr	Sedan	Asia	Front	$26,960	$24,304	3	6	
164	Honda	Odyssey LX	Sedan	Asia	Front	$24,950	$22,498	3.5	6	
165	Honda	Odyssey EX	Sedan	Asia	Front	$27,450	$24,744	3.5	6	
166	Honda	S2000 convertible 2dr	Sports	Asia	Rear	$33,260	$29,965	2.2	4	

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

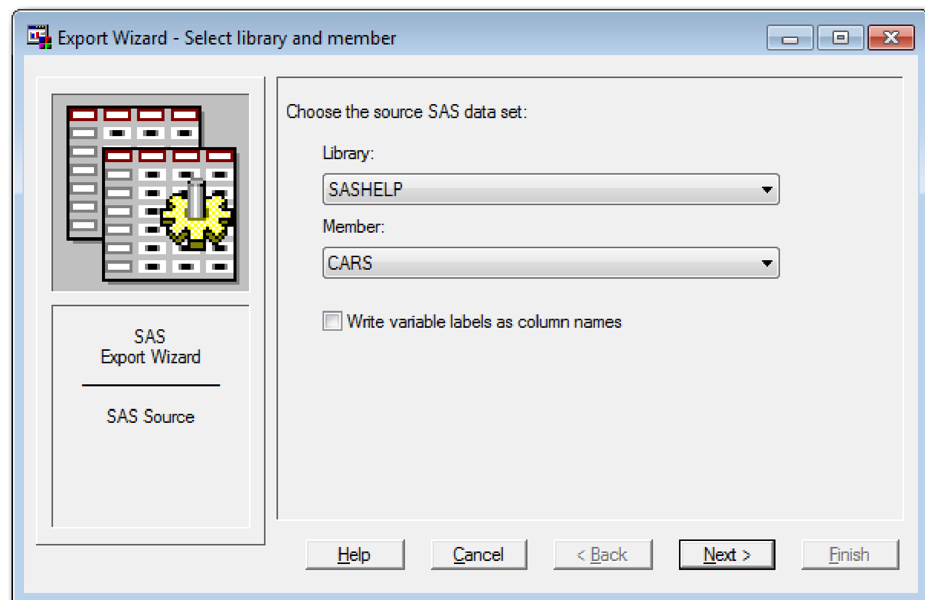
To export data, follow these steps:

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.

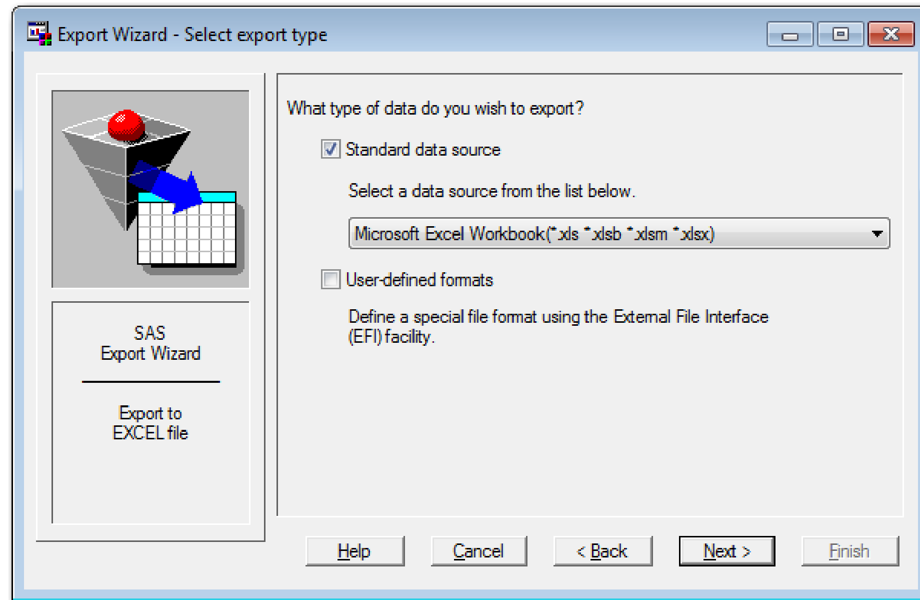
In this example, **Sashelp** is selected as the library, and **Cars** is the member name.



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.

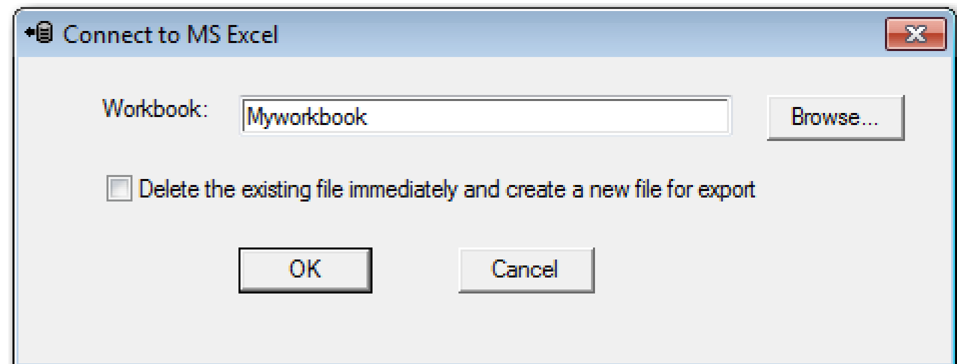
In this example, **Microsoft Excel Workbook** is selected. Note that **Standard data source** is selected by default.



5 Click **Next** to display the Connect to MS Excel window.

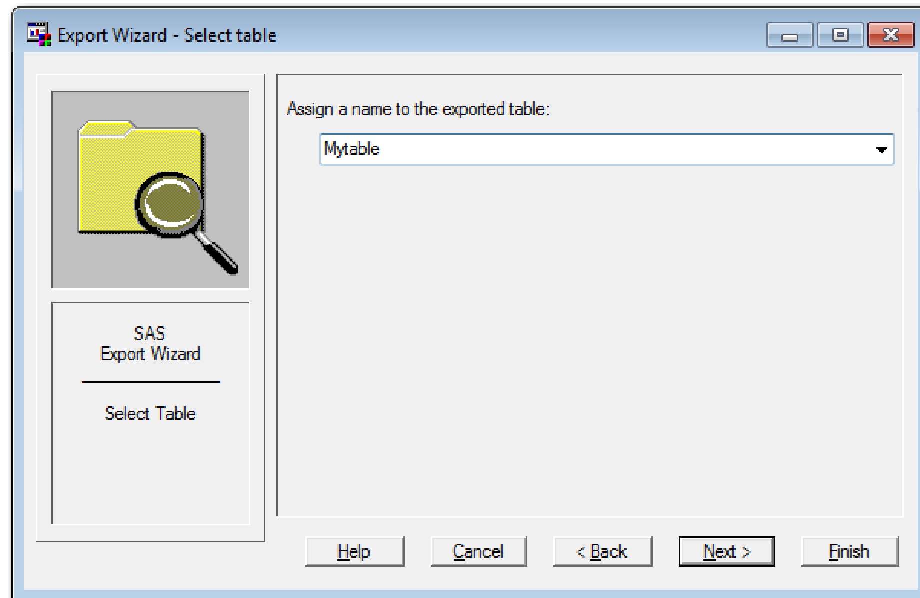
6 In the **Workbook** field, enter the name of the workbook that will contain the exported file and then click **OK**.

In this example, **Myworkbook** is entered as the name of the workbook.



7 When the Export Wizard - Select table window appears, enter a name for the table that you are exporting.

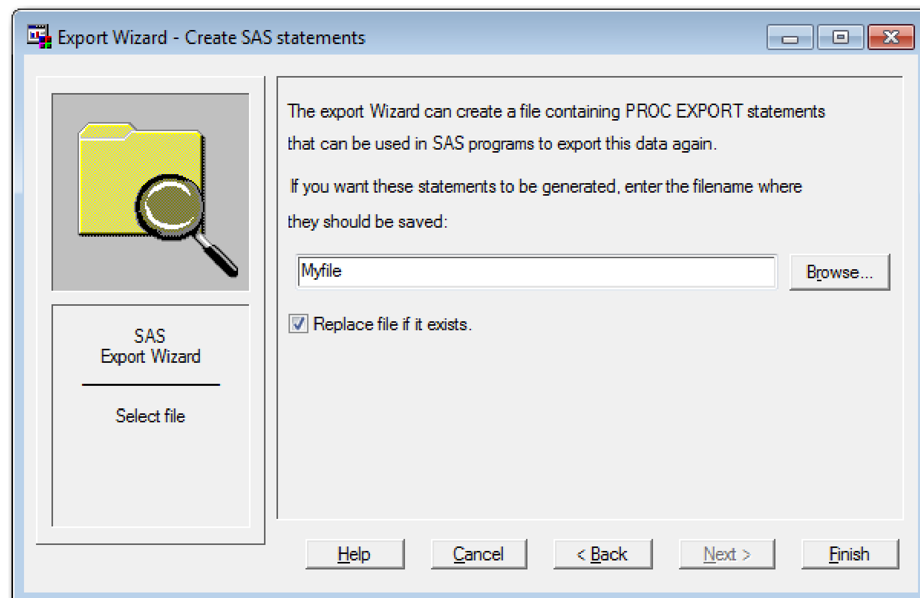
In this example, **Mytable** is the table name.



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.



10 Click **Finish** to complete this task.

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

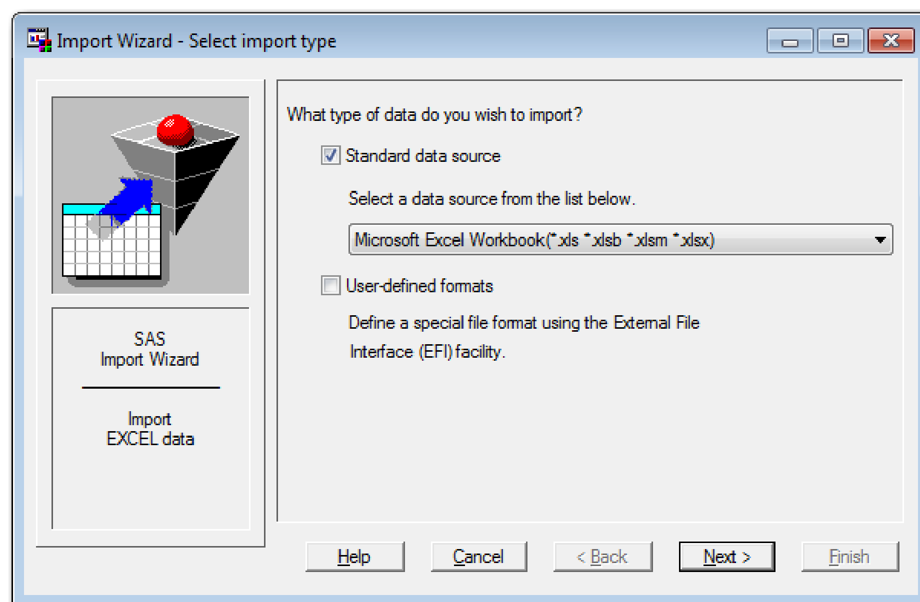
To import a standard file, follow these steps:

1 With the Explorer window active, select **File** ⇨ **Import data**.

The Import Wizard - Select import type window appears.

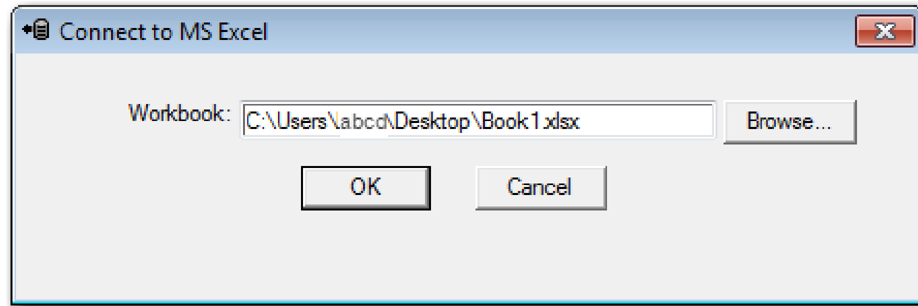
2 Select the type of file that you are importing by selecting a data source from the **Select a data source** menu.

Note that **Standard data source** is selected by default. In this example, **Microsoft Excel Workbook** is selected.

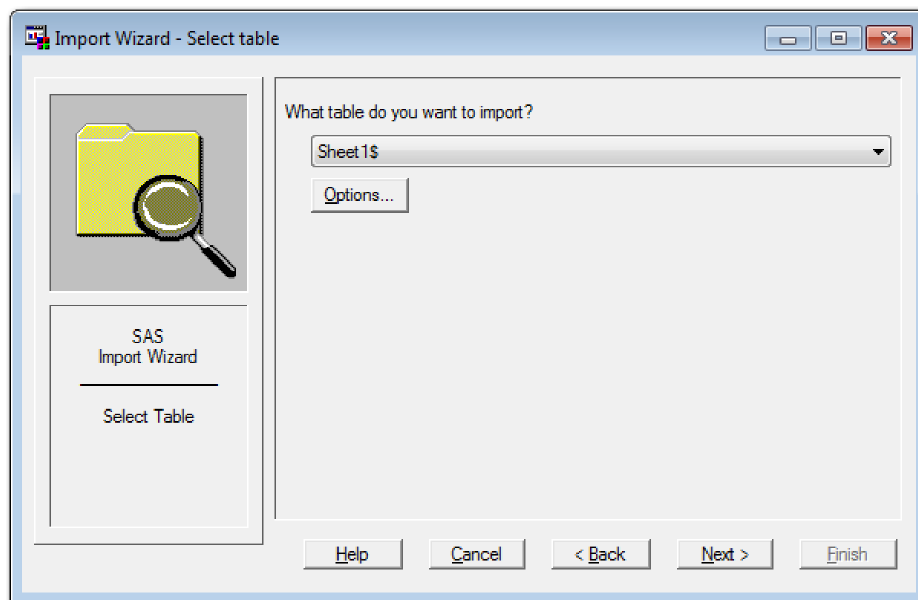


3 Click **Next** to continue.

4 In the Connect to MS Excel window, enter the pathname of the file that you want to export, and then click **OK**.



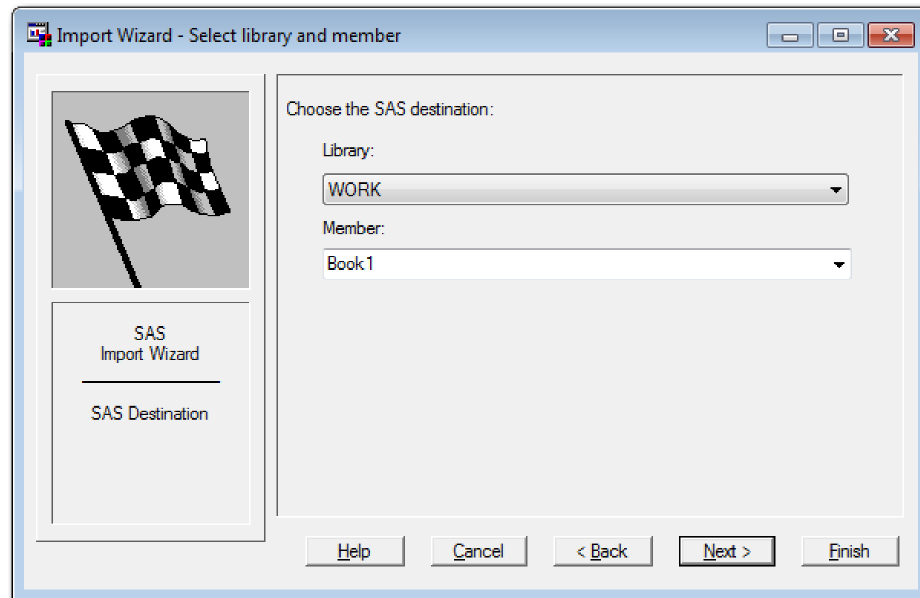
5 In the Import Wizard - Select table window, enter the name of the table that you want to import.



6 Click **Next** to continue.

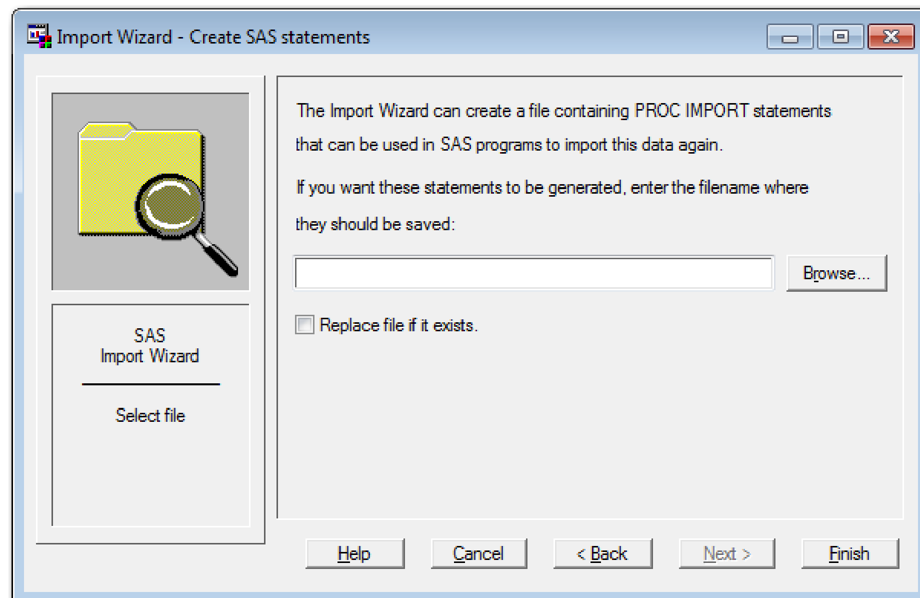
7 In the Import Wizard - Select library and member window, enter a location in which to store the imported file.

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.

37

Cloud Analytic Services

What is SAS Cloud Analytic Services? 831
What Does This Mean for the SAS 9 Programmer? 831
SAS Language Elements for CAS 832
DATA Step Processing 832
DATA Step Language Elements for CAS 832
SAS Procedures for CAS 833
CAS-specific Language Elements 834

What is SAS Cloud Analytic Services?

SAS Cloud Analytic Services (CAS) is a server that provides the cloud-based run-time 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*

SAS language elements that are supported in CAS display “CAS” in the Categories field of the syntax page for the language element:

Figure 37.2 *Example Documentation Syntax Page That Shows Support for CAS in the Categories Field*

BY Statement
CALL Statement
CARDS Statement
CARDS4 Statement

BY Statement

Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement

Valid in:	DATA step or PROC step
Categories:	CAS
	File-Handling

Each language element dictionary also contains a summary table of CAS-supported language elements:

Figure 37.3 *Example Documentation Category Page Showing CAS-Supported Language Elements*

SAS Data Set Options by Category
ALTER= Data Set Option
BUFNO= Data Set Option
BUFSIZE= Data Set Option
COMPRESS= Data Set Option
CNTLLEV= Data Set Option
DLDMGACTION= Data Set Option

SAS Data Set Options by Category

The categories for SAS data set options correspond to the SAS data set option gr

CAS	options that run in the CAS server
Data Set Control	options that are associated with data sets
Observation Control	options that are associated with observations
User Control of SAS Index Usage	options that are associated with indexes
Variable Control	options that are associated with variables
Miscellaneous	option that is associated with tape position

Category	Language Elements
CAS	COMPRESS= Data Set Option
	IN= Data Set Option
	KEEP= Data Set Option

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 Language Element Syntax: *SAS Cloud Analytic Services: User's Guide*
- CAS Conceptual Information: *SAS Cloud Analytic Services: Fundamentals*
- Introduction for SAS 9 programmers: *An Introduction to SAS Viya Programming*
- CAS Actions in SAS Viya Actions and Action Sets by Name and Product, *SAS Viya: System Programming Guide* and CAS DATA Step Action in *SAS Cloud Analytic Services: DATA Step Programming*

Note: A SAS Viya Visual Analytics license is required for access to SAS Cloud Analytic Services.

38

Industry Protocols Used in SAS

SAS Language Elements That Control SMTP E-Mail **836**
- System Options That Control SMTP E-Mail 836
- Statements That Control SMTP E-mail 837

How the SMTP e-Mail Interface Authenticates Users **838**

Universally Unique Identifiers and the Object Spawner **838**
- What Is a Universally Unique Identifier? 838
- What Is the Object Spawner? 839
- Defining the UUID Generator Daemon 839
- Installing the UUID Generator Daemon 840

Using SAS Language Elements to Assign UUIDs **841**
- Overview of Using SAS Language Elements to Assign UUIDs 841
- UUIDGEN Function 841
- UUIDCOUNT= System Option 842
- UUIDGENDHOST System Option 842

Overview of IPv6 . . . . . . **842**

IPv6 Address Format **843**

Examples of IPv6 Addresses **843**
- Example of Full and Collapsed IPv6 Address 843
- Example of an IPv6 Address That Includes a Port Number 844
- Example of an IPv6 Address That Includes a URL 844

Fully Qualified Domain Names (FQDN) **844**

SAS Language Elements That Control SMTP E-Mail

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.

How the SMTP e-Mail Interface Authenticates Users

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.

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:

```
#
##  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:

```
C:\Program Files\SASHome\SASFoundation\9.4
```

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

Feature	IPv6	IPv4
Address Space	128-bit	32-bit
Representation	string	integer
Length (including Field Separators)	39	15
Field Separator	colon (:)	period (.)
Notation	hexadecimal	decimal
Example of IP Address	db8:0:0:1	10.23.2.3

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:

```
PROC METAOPERATE
   SERVER="d6292.us.company.com"
   PORT=2222
   USERID="myuserid"
```

```
PASSWORD="mypassword"
PROTOCOL=BRIDGE

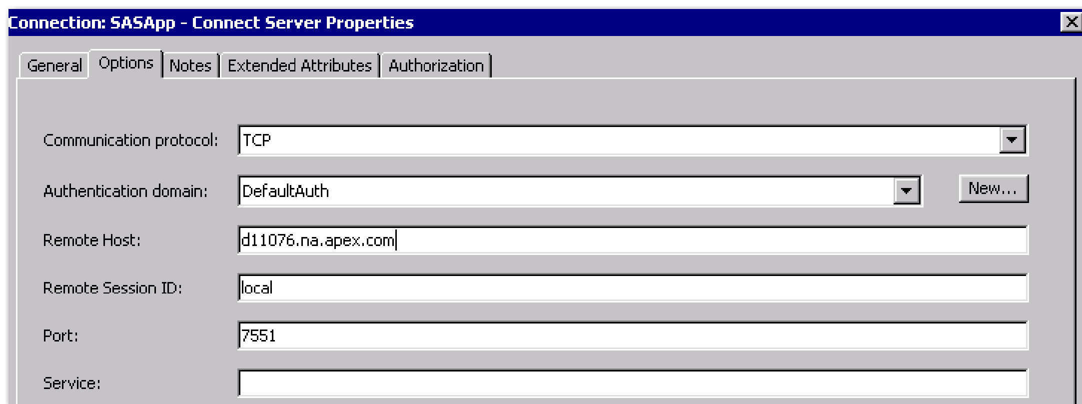
ACTION=RESUME
 OPTIONS=""
NOAUTOPAUSE;
```

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.

Figure 38.1 Example of an FQDN in a SAS Management Console Window



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:

```
%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.

PART 8

Appendixes

Appendix 1
Data Sets Used in Examples 849

Appendix 2
Understanding How the DATA Step Works 863

Appendix 3
Updating Data Using the MODIFY Statement and the KEY= Option 879

Appendix 1

Data Sets Used in Examples

Data Sets Used in Examples 850
Animal 850
AnimalDupes 850
AnimalMissing 850
CarSales 851
CarsSmall 851
Class 852
Classfit 852
Inventory 853
InventoryAdd 853
One 853
Many 854
Master 854
Minerals 854
Plant 855
PlantDupes 855
PlantG 855
PlantMissing 855
PlantMissing2 856
PlantNew 856
PlantNewDupes 856
Product_List 857
Quarter1, Quarter 2, Quarter3, Quarter4 857
Sales 858
Sales2019 858
Supplier 858
Table1 859
Table2 859
Year1 859
Year2 860
Description of Column-Binary Data Storage 860

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
210200100009 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
755  Top Sports                  Jernbanegade 45     DK
772  AllSeasons Outdoor Clothing 553 Cliffview Dr    US
798  Sportico                    C. Barquillo 1      ES
1280 British Sports Ltd          85 Station Street   GB
1303 Eclipse Inc                 1218 Carriole Ct    US
1684 Magnifico Sports            Rua Costa Pinto 2   PT
1747 Pro Sportswear Inc          2434 Edgebrook Dr   US
3298 A Team Sports               2687 Julie Ann Ct   US
3808 Carolina Sports             3860 Grand Ave      US
;
```

Table1

```
data Table1;
   set sashelp.class(where=(age=14));
run;
```

Table2

```
data Table2;
   set sashelp.classfit(where=(age=14));
   drop uppermean lower upper;
run;
proc sort data=Table2; by name; run;
```

Year1

```
data Year1;
input date;
datalines;
2009
2010
2011
2012
;
```

Year2

```
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

```
                row   punch

                12    X X X X X X X X X - - - - - - - - - - - - - - - - -
zone            11    - - - - - - - - - X X X X X X X X X - - - - - - - -
portion         10    - - - - - - - - - - - - - - - - - - X X X X X X X X

                 1    X - - - - - - - - X - - - - - - - - - - - - - - - -
                 2    - X - - - - - - - - X - - - - - - - X - - - - - - -
                 3    - - X - - - - - - - - X - - - - - - - X - - - - - -
digit            4    - - - X - - - - - - - - X - - - - - - - X - - - - -
portion          5    - - - - X - - - - - - - - X - - - - - - - X - - - -
                 6    - - - - - X - - - - - - - - X - - - - - - - X - - -
                 7    - - - - - - X - - - - - - - - X - - - - - - - X - -
                 8    - - - - - - - X - - - - - - - - X - - - - - - - X -
                 9    - - - - - - - - X - - - - - - - - X - - - - - - - X

alphabetic
character             A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

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

row	byte 1 position	byte 2 position
(not used)	1, 2	1, 2
12	3	
11	4	
10 (or 0)	5	
1	6	
2	7	
3	8	
4		3
5		4
6		5
7		6
8		7
9		8

Appendix 2

Understanding How the DATA Step Works

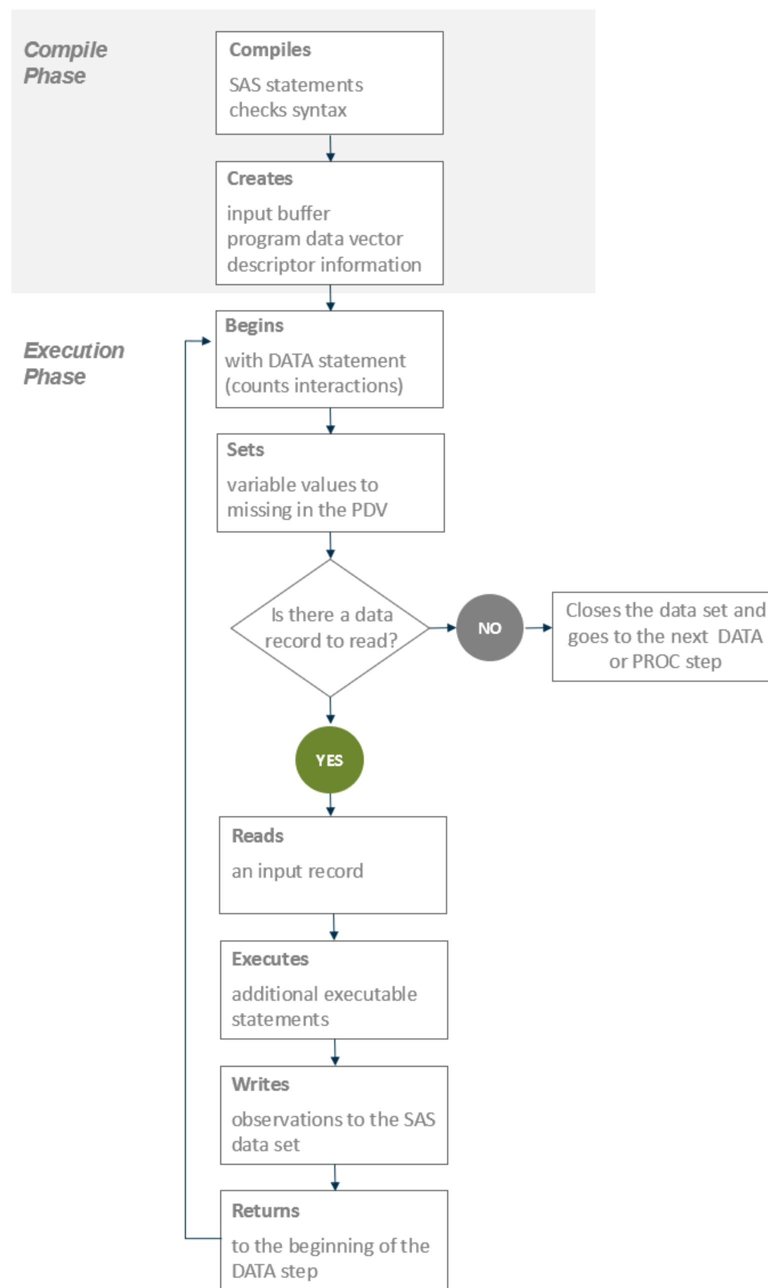
How the DATA Step Processes Data 863
Flow of Action 863
The Compilation Phase 864
The Execution Phase 865
Processing a DATA Step: A Walk-Through 866
Sample DATA Step 866
Creating the Input Buffer and the Program Buffer 867
Reading a Record 868
Writing an Observation to the SAS Data Set 869
Reading the Next Record 869
When the DATA Step Finishes Executing 870
More About DATA Step Execution 871
The Default Sequence of Execution 871
Changing the Sequence of Execution Using Statements 873
Changing the Sequence of Execution Using Functions 873
Altering the Flow for a Given Observation 874
Step Boundaries 875
What Causes the DATA Step to Stop Executing 877

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.

Figure A2.1 *Flow of Action in the 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.

6 SAS counts another iteration, reads the next record or observation, and executes the subsequent programming statements for the current observation.

7 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:

```
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	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5

Program Data Vector

TeamName	ParticipantName	Event1	Event2	Event3	TeamTotal	_N_	_ERROR_
		.	.	.	0	1	0
Drop						Drop	Drop

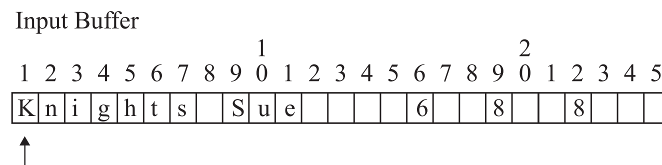
Variables that are created by the INPUT and the Sum statements (TeamName, ParticipantName, Event1, Event2, Event3, and TeamTotal) are set to missing initially. Note that in this representation, numeric variables are initialized with a period and character variables are initialized with blanks. The automatic 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 75 for details about automatic variables.

Reading a Record

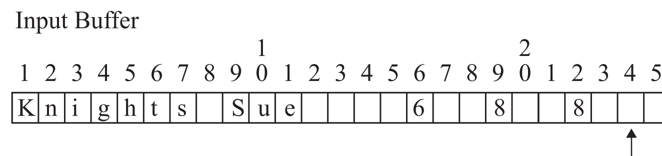
SAS reads the first data line into the input buffer. The input pointer, which SAS uses to keep its place as it reads data from the input buffer, is positioned at the beginning of the buffer, ready to read the data record. The following figure shows the position of the input pointer in the input buffer before SAS reads the data.

Figure A2.3 *Position of the Pointer in the Input Buffer Before SAS Reads Data*



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*



Program Data Vector

TeamName	ParticipantName	Event1	Event2	Event3	TeamTotal	_N_	_ERROR_
Knights	Sue	6	8	8	0	1	0
Drop						Drop	Drop

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
Drop						Drop	Drop

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*

Output SAS Data Set TOTAL_POINTS: 1st observation

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

TeamName	ParticipantName	Event1	Event2	Event3	TeamTotal	_N_	_ERROR_
		•	•	•	22	2	0
Drop						Drop	Drop

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*

Input Buffer

1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	20	1	2	3	4	5
C	a	r	d	i	n	a	l	s		J	a	n	e		9			7			8		↑	

Program Data Vector

TeamName	ParticipantName	Event1	Event2	Event3	TeamTotal	_N_	_ERROR_
Cardinals	Jane	9	7	8	46	2	0
Drop						Drop	Drop

Output SAS Data Set TOTAL_POINTS: 1st and 2nd observations

ParticipantName	Event1	Event2	Event3	TeamTotal
Sue	6	8	8	22
Jane	9	7	8	46

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:

```
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*

Total Team Scores

Obs	ParticipantName	Event1	Event2	Event3	TeamTotal
1	Sue	6	8	8	22
2	Jane	9	7	8	46
3	John	7	7	7	67
4	Lisa	8	9	9	93
5	Fran	7	6	6	112
6	Walter	9	8	10	139

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 data set name if you do not want to create an output data set.) 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*

Structure of a DATA Step	Action
DATA statement	begins the step counts iterations
Data-reading statements: [1]	
INPUT	describes the arrangement of values in the input data record from a raw data source
SET	reads an observation from one or more SAS data sets

Structure of a DATA Step	Action
MERGE	joins observations from two or more SAS data sets into a single observation
MODIFY	replaces, deletes, or appends observations in an existing SAS data set in place
UPDATE	updates a master file by applying transactions
Optional SAS programming statements, for example:	further processes the data for the current observation
FirstQuarter=Jan+Feb+Mar; if RetailPrice < 500;	computes the value for FirstQuarter for the current observation subsets by value of variable RetailPrice for the current observation
Default actions at the end of processing an observation	
At end of DATA step: Automatic write, automatic return At top of DATA step: Automatic reset	writes an observation to a SAS data set returns to the DATA statement resets values to missing in program data vector

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.

Note: 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*. For specific information about SAS functions, see the SAS File I/O and External Files categories in "SAS Functions and CALL Routines by Category" in *SAS Functions and CALL Routines: Reference*.

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*

Task	Possible Methods
Read a record	merge, modify, join data sets read multiple records to create a single observation randomly select records for processing read from multiple external files read selected fields from a record by using statement or data set options
Process data	use conditional logic retain variable values
Write an observation	write to a SAS data set or to an external file control when output is written to a data set write to multiple files

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

SAS Language Element	Function
subsetting IF statement	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.
IF-THEN/ELSE statement	executes a SAS statement for observations that meet the current condition and continues with the next statement.
DO loops	cause parts of the DATA step to be executed multiple times.
LINK and RETURN statements	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.
HEADER= option in the FILE statement	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.
GO TO statement	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.
EOF= option in an INFILE statement	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.

SAS Language Element	Function
_N_ automatic variable in an IF-THEN construct	causes parts of the DATA step to execute only for particular iterations.
SELECT statement	conditionally executes one of a group of SAS statements.
OUTPUT statement in an IF-THEN construct	outputs an observation before the end of the DATA step, based on a condition; prevents automatic output at the bottom of the DATA step.
DELETE statement in an IF-THEN construct	deletes an observation based on a condition and causes a return to the top of the DATA step.
ABORT statement in an IF-THEN construct	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.
WHERE statement or WHERE= data set option	causes SAS to read certain observations based on one or more specified criteria.

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:

```
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.

```
data scores;  1
   set allscores(drop=score5-score7);
run;  2

options firstobs=5 obs=55;  3

data test;
   set alltests;
run;
```

1 The DATA statement is a step boundary.

2 The RUN statement is the boundary for the first DATA step.

3 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

Data Read	Data Source	SAS Statements	DATA Step Stops
no data			after only one iteration
any data			when it executes STOP or ABORT when the data is exhausted
raw data	instream data lines	INPUT statement	after the last data line is read
	one external file	INPUT and INFILE statements	when end-of-file is reached
	multiple external files	INPUT and INFILE statements	when end-of-file is first reached on any of the files
observations sequentially	one SAS data set	SET and MODIFY statements	after the last observation is read
	multiple SAS data sets	one SET, MERGE, MODIFY, or UPDATE statement	when all input data sets are exhausted
	multiple SAS data sets	multiple SET, MERGE, MODIFY, or UPDATE statements	when end-of-file is reached by any of the data-reading statements

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*.

Appendix 3

Updating Data Using the MODIFY Statement and the KEY= Option

Updating Data Using the MODIFY Statement and the KEY= Option 879
Definitions 880
Understanding the MODIFY Statement 881
Sequential Access 883
Matching Access Using BY Statement 883
Comparing Matching Access to the UPDATE Statement 885
Direct (Random) Access by Observation Number Using POINT= 887
Direct Access by Index Values Using KEY= 887
Comparing the Matching Access Method Using the BY Statement to the Direct Access Method Using Index Values 888
Monitoring Update Processing 889
Performing Automatic Updates for Like-Named Variables Using KEY= 890
Using MODIFY in a SAS/SHARE Environment 892
Understanding the KEY= Option 893

Updating Data Using the MODIFY Statement and the KEY= Option

You can use the KEY= option with either the MODIFY statement or the "SET Statement" in *SAS DATA Step Statements: Reference* to update SAS data sets. You can also use the "SET Statement" in *SAS DATA Step Statements: Reference*, MERGE statement, and UPDATE statement in the DATA step to modify and combine data. The MODIFY statement used with the KEY= option provides the following benefits:

- The MODIFY statement uses less disk space than the SET, MERGE, or UPDATE statements.
- The KEY= option in the MODIFY statement enables you to take advantage of indexing.

For more information about combining SAS data sets, see “Modifying” on page 487.

Definitions

Note the following SAS terms and definitions, which are used in this appendix:

data set
: a SAS file that consists of descriptor information and data values organized as a table of rows (SAS observations) and columns (SAS variables) that can be processed by SAS.

IORC
: an automatic variable created by SAS when you use the MODIFY statement or the SET statement with the KEY= option. The value of _IORC_ is a numeric return code that indicates the status of the most recent I/O operation performed on an observation in a SAS data set. The return code indicates whether the retrieval for matching observations was successful:

Table A3.1 *_IORC_ Return Code Values*

Code	Value
0	successful execution
-1	end-of-file error
all other values	non-match

Typically, you use _IORC_ in conjunction with the autocall macro %SYSRC, which enables you to specify a mnemonic name that describes a potential outcome of an I/O operation.

Note: In Version 7, the IORCMSG function returns a formatted error message associated with the current value of _IORC_.

index
: a file that is created when you define an index for a SAS data set.

program data vector (PDV)
: a physical area in memory where SAS builds a data set, one observation at a time. When a program executes, the software reads values from the input buffer or creates them by executing SAS language statements. The values are assigned to the appropriate variables in the PDV. From here, the software writes the values to a SAS data set as a single observation.

%SYSRC
: an autocall macro that provides a convenient way of testing for a specific I/O error condition created by the most recently executed MODIFY statement or

SET statement with the KEY= option. %SYSRC returns a numeric value corresponding to the mnemonic string passed to it. The mnemonic is a literal that corresponds to a numeric value of the _IORC_ automatic variable. SAS supplies a library of autocall macros to each site. The autocall facility enables you to invoke a macro without having previously defined that macro in the same SAS program. To use the autocall facility, specify the MAUTOSOURCE system option.

Note: The DATA= option is used with the PRINT procedure to specify the `master` data set. If DATA= is not specified, PROC PRINT displays the last created data set that was opened for output. Note that the last created data set might not be the one that was just modified.

view
contains only the information required to retrieve data values. The data is obtained from another file. There are three types of views:

DATA step view
a compiled DATA step program that can read from one or more SAS data sets or external files.

SAS/ACCESS view
defines data formatted by other software products. When a SAS/ACCESS view is processed, the data that it accesses remains in its original format.

PROC SQL view
a compiled query that obtains values from one or more data files or views or the SQL Procedure Pass-Through Facility.

Understanding the MODIFY Statement

The MODIFY statement replaces, deletes, or appends observations in an existing data set without creating a copy of the data set. This is referred to as *updating in place*. Because the MODIFY statement does not create a copy of the data set that is open for update, the processing uses less disk space than does the MERGE, SET, or UPDATE statements. These statements create a new data set.

If data set options such as KEEP, RENAME, or DROP are used on the modified data set, the options are in effect only during processing. The descriptor portion of a SAS data set opened in update mode cannot be changed.

For information about the parts of a SAS data set, including the descriptor portion, see PDV.

The data set referenced in the MODIFY statement must also be referenced in the DATA statement. Then, based on the DATA step logic, you can replace, delete, and append new observations. For example, the following simple DATA step uses the MODIFY statement to update the data set Invty.Stock by replacing the date values in all observations for the variable Recdate with the current date:

```
data invty.stock;
   modify invty.stock; recdate=today();
```

```
run;
```

When SAS processes the previous DATA step, the following occurs:

1 The MODIFY statement opens SAS data set Invty.Stock for update processing.

2 The first observation is read from the data set and the values are written into the PDV. The variable `Recdate` exists in the data set `Invty.Stock`.

3 The value of variable Recdate is replaced with the result of the TODAY function.

4 The values in the PDV replace the values in the data set.

5 The process is repeated for each observation, using a sequential access method, until the end-of-file marker is reached.

To further control processing, you can include the following statements in a DATA step execution in conjunction with the MODIFY statement:

- OUTPUT statement — appends the current observation as a new observation to the end of the data set. The data set specified in the MODIFY statement must also be specified in the DATA statement as the output data set.
- REMOVE statement — deletes the current observation from the data set. The deletion is either a physical or logical deletion, depending on the SAS I/O engine maintaining the data set.
- REPLACE statement — writes the current observation to the same physical location; that is, replaces it in the data set. An implicit REPLACE statement at the bottom of the DATA step is the default action.

The processing of the MODIFY statement depends on whether any of those statements are included in the DATA step:

- If a REPLACE, REMOVE, or OUTPUT statement is not specified in the DATA step, the software writes the current observation to its original place in the data set as if a REPLACE statement was the last statement in the DATA step. That is, the MODIFY statement generates a REPLACE statement at the bottom of the DATA step, which is referred to as an implicit REPLACE.
- If a REPLACE, REMOVE, or OUTPUT statement is included, it overrides the implicit REPLACE.

The REPLACE, REMOVE, and OUTPUT statements are independent of each other. More than one statement can apply to the same observation. Note that if both an OUTPUT statement and a REPLACE or REMOVE statement execute on the same observation, be sure the OUTPUT statement is executed last to keep proper positioning in the index.

The MODIFY statement supports four different access methods:

- sequential access
- matching access using the BY statement
- direct (random) access by observation number using POINT=
- direct access by indexed values using KEY=

Sequential Access

Sequential access is the simplest form of processing using the MODIFY statement. Sequential access provides less control than the other access methods, but it provides the quickest method for updating all observations in a data set. The syntax for sequential access is:

MODIFY*master-data-set*<*(data-set-option(s))*><NOBS=*variable*><END=*variable*>

In the following code, SAS sequentially accesses each observation in the `master` data set looking for an observation that contains the value **u** for variable `mod`. When an observation is located, the value of variable `x` is replaced with a **1**. The execution of the implicit REPLACE causes the observation to be rewritten with the updated value. The variable `x` must exist in the `master` data set. The MODIFY statement opens the data set in update mode, and the header information for a data set opened for update cannot be modified.

```
data master;
   modify master;
   if mod='u' then x=1;
run;
```

Matching Access Using BY Statement

Matching access matches the value or values of one or more BY variables in the master data set against the same variables in a second data set, referred to as a *transaction data set*. The syntax for matching access is as follows:

MODIFY *master-data-set*<*(data-set-options)*> *transaction-data-set*<*(data-set-options)*> <NOBS=*variable*> <END=*variable*>
<UPDATEMODE=<MISSINGCHECK>NOMISSINGCHECK>;
BY *by-variables*;

The BY statement is required. The specified variables must exist in both the master data set and the transaction data set. When using MODIFY with the BY statement, the rules that apply to the UPDATE statement also apply to MODIFY, except that the data sets are not required to be in sorted order.

The MODIFY statement executes as follows:

1. The MODIFY statement opens the specified data set for update processing.
2. The MODIFY statement reads an observation from the transaction data set to a temporary storage location in memory.
3. The MODIFY statement then generates a WHERE statement, specifying the values of the BY variables (for example, `partno`) from the transaction observation to locate and fetch an observation with a matching BY variable value from the master data set. This observation is placed in the PDV.

4 Values from the transaction observation in temporary storage are used to overlay the values of like-named variables located in the PDV from the master observation.

The following behavior applies to matching access:

- When matched on the variables specified in the BY statement, all like-named variables in the master data set are automatically updated with the values from the transaction data set.
- If duplicate values for the BY variables exist in the master data set, only the first observation in that BY group is updated; observations with duplicate BY values are ignored. This is because SAS locates the observation in the master data set by generating a WHERE statement containing the transaction data set values for the BY variables. The WHERE statement always returns the first occurrence of the BY group for updating in the master data set.
- If duplicates exist in a BY group in the transaction data set, the updates are applied one after another to the first matching observation in the master data set. An accumulative statement must be used to increment the values from the duplicates. Otherwise, the value from the last duplicate is applied to the matching observation in the master data set.
- Because a WHERE statement is generated, neither the master data set nor the transaction data set has to be sorted on the BY variables. However, sorting of both data sets can greatly reduce the I/O to retrieve an observation from the master data set. If the master data set is indexed for the BY variables, the generated WHERE statement can use the index.
- Missing values from the transaction data set can update the master data set if UPDATEMODE=MISSINGCHECK is the default behavior. If the option UPDATEMODE=NOMISSINGCHECK is not available under your current release, the master data set observation can be updated with a special missing value in the transaction data set observation. For numeric data, you can differentiate among types of missing values. That is, you can specify up to 27 characters (the underscore (_) and the letters A through Z) to designate different types of missing values. Another approach is to use the RENAME= data set option to change the names of those variables in the transaction data set so that the master data set variable can be set equal to the renamed transaction data set variable.

The following example uses the matching access method to update the master data set `invty.stock` using information from the transaction data set `work.addinv` as well as to update the date on which stock was received:

```
data invty.stock;
   modify invty.stock addinv;
   by partno; recdate=today();
   instock=instock + nwstock;
run;
```

Even though you do not have to sort or index either the transaction data set or the master data set, you can improve performance by the following:

- creating an index for the master data set on the variable used as the BY variable
- sorting both data sets by the BY variable

Comparing Matching Access to the UPDATE Statement

The MODIFY statement for matching access is almost identical to that of the UPDATE statement. Both require a master and a transaction data set, and both require a BY statement. The MODIFY statement compares to the UPDATE statement as follows:

- The MODIFY statement opens the SAS data set specified on the MODIFY and DATA statements in update mode. However, the UPDATE statement opens the specified SAS data set for input, and the SAS data set specified in the DATA statement is opened for output.
- The MODIFY statement updates the SAS data set in place. With the UPDATE statement, the original SAS data set is replaced with a copy of the updated version.
- The MODIFY statement results in an implicit REPLACE statement being executed at the time the that DATA step iterates. UPDATE causes an implicit OUTPUT statement to be executed at the time the that DATA step iterates.

The difference between the implicit REPLACE and the implicit OUTPUT statements is illustrated in the following example, which uses two data sets: `master` and `trans`. Their DATA steps are shown below:

```
data master;
   input ssn : $11. nickname $;
datalines;
134-56-9094 Megan
160-58-1223 Kathryn
161-60-5881 Joshua
;

data trans;
   input ssn : $11. nickname $;
datalines;
134-56-9094 Meg
142-67-9888 Bill
160-58-1223 Kate
161-60-5881 .
;
```

To process the previous sorted data, you can use the UPDATE statement. The UPDATE statement generates an implicit OUTPUT statement. The OUTPUT statement appends the values from the current program data vector to the end of the data set. This happens regardless of whether there is a match on the BY variables.

```
data master;
   update master trans
   updatemode=nomissingcheck;
   by ssn;
run;
```

However, if you change the UPDATE statement to a MODIFY statement as shown below, it fails. The generated WHERE statement finds no match for ssn 142-67-9888, and the implicit REPLACE statement at the bottom of the DATA step tries to replace an observation that it was unable to retrieve.

```
data master;
   modify master trans
   updatemode=nomissingcheck;
   by ssn;
run;
```

The SAS log reflects the value **1230013** in the automatic variable _IORC_, because a match for the second observation in the `trans` data set does not exist in the `master` data set. Because an error occurs, the automatic variable _ERROR_ is set to a value of **1**.

```
ERROR: The TransACTION data set observation does not exist on the MASTER data set.
ERROR: No matching observation was found in MASTER data set.
ssn=142-67-9888 nickname=Bill FIRST.ssn=1 LAST.ssn=1 _ERROR_=1 _IORC_=1230013 _N_=2
NOTE: TRANS data set does not exist in the MASTER data set. Because an error occurs,
the automatic variable
_ERROR_ is set to a value of 1.he SAS System stopped processing this step because of
errors. NOTE:
There were 1 observations read from the dataset WORK.MASTER.
NOTE: The data set WORK.MASTER has been updated. There were 1 observations rewritten,
0 observations added and 0 observations deleted.
NOTE: There were 3 observations read from the dataset WORK.TRANS.
```

To prevent the error caused by the implicit REPLACE statement on a non-match record, use a conditional IF statement so that data is replaced only when a match is located. The automatic variable _IORC_ holds a value of zero on a successful I/O operation.

```
data master;
   modify master trans
   updatemode=nomissingcheck;
   by ssn;
   if _iorc_=0 then replace;
run;
```

To prevent displaying the non-matched record in the SAS log, reset the automatic variable _ERROR_ to zero. (Error checking is discussed later in this appendix.)

```
data master;
   modify master trans
   updatemode=nomissingcheck;
   by ssn;
   if _iorc_=0 then replace;
   else _error_=0;
run;
```

Direct (Random) Access by Observation Number Using POINT=

Direct (random) access by observation number requires that you use POINT= to identify the observation to be updated. POINT= specifies a variable from another data source (not the master data set) or one that you define in the DATA step whose value is the number of an observation that you want to modify in the master data set. MODIFY uses the values of the POINT= variable to retrieve observations in the data set that you want to modify. The syntax for direct (random) access by observation number is as follows:

MODIFY *master-data-set* <(*data-set-options*)> <NOBS=*variable*> POINT=*variable*;

To illustrate direct access by observation number, assume that you have a data set named `newp`. `newp` has variable `tool_obs` containing the observation number of each tool in the tool company's master data set `invty.stock` and variable `newprice` containing the new price for each tool.

The following example uses the information in data set `newp` to update data set `invty.stock`. `newp` is specified in the SET statement, which reads values to be supplied for the `tool_obs` and `newprice` variables. Variable `tool_obs` is specified as the value of the POINT= option. Variable `newprice` is specified in the assignment statement to replace existing values of `price` in `invty.stock`. As the SET statement executes, values of `tool_obs` are read from `newp`, placed in the PDV, and then used by the MODIFY statement to retrieve observations directly from `invty.stock`. The observation number in `tool_obs` is used as a key to allow direct retrieval of observations.

```
data invty.stock;
   set newp;
   modify invty.stock point=tool_obs;
   price=newprice;
   recdate=today():
run;
```

Direct Access by Index Values Using KEY=

Direct access by index values requires that you use KEY= to specify an existing index. The software then retrieves observations in the data set through the index based on the index values. The syntax for direct access by index values is as follows:

MODIFY *master-data-set* <(*data-set-options*)> KEY=*index* </ UNIQUE>
<NOBS=*variable*> <END=*variable*>;

Direct access by index values lets you supply a lookup value from a secondary data source such as another SAS data set. An observation is read using a SET statement to supply a lookup value, which is then used as a key to search the master data set

to locate the observation. The search is performed through an index created for that data set. You specify the index name with the KEY= option. Once the observation is located, you can assign variables new values and perform other processing on the observation prior to the implicit REPLACE. If the observation is not located, an appropriate action such as OUTPUT is performed. The SAS data set supplying the lookup value (specified in the SET statement) must contain the same names as those specified in the KEY= option.

The following example uses the KEY= option to specify the index with which to identify observations for retrieval by matching the values of variable `partno` from data set `work.addinv` with the indexed values of variable `partno` in data set `invty.stock`:

```
data invty.stock;
   set addinv;
   modify invty.stock key=partno;
   if _iorc_=0 then do;
      instock=instock+nwstock;
      recdate=today();
      replace;
   end;
   else _error_=0;
run;
```

Comparing the Matching Access Method Using the BY Statement to the Direct Access Method Using Index Values

The matching access method is a form of the MODIFY statement that uses the a BY statement to match observations from the transaction data set with observations in the master data set. For an example that shows the matching access method, see “Modifying Observations Using a Transaction Data Set” in *SAS DATA Step Statements: Reference*.

The direct access method is a form of the MODIFY statement that uses the KEY= option in the MODIFY statement to name an indexed variable from the data set that is being modified. For an example that shows the direct access method by indexed values, see “Modifying Observations Located by an Index” in *SAS DATA Step Statements: Reference*.

These two methods (which uses the BY statement) and the direct access method by using an index (which uses the KEY= option) compare as follows:

Table A3.2 Comparing the Matching Access Method Using the BY Statement to the Direct Access Method Using Index Values

MODIFY with BY Statement	MODIFY with KEY= Option
Matching observations in the master data set are retrieved by a generated WHERE statement. The WHERE statement can use an index created on the master data set.	Matching observations in the master data set are retrieved through the index specified with the KEY= option.
If duplicate BY values exist in the master data set, only the first occurrence is updated.	If duplicate KEY= values exist in the master data set, the duplicate values can be updated by using a DO UNTIL loop. The DO UNTIL loop can be written to force the execution of the KEY= option on the master data set until a non-match condition occurs. See “Using MODIFY with Duplicate Key Values in the Master Data Set” on page 897.
If duplicate BY values exist in the transaction data set, they are applied one after another to the same observation in the master data set, unless you write an accumulation statement. Otherwise, the last duplicate is applied.	All consecutive duplicates in the transaction are applied by using the UNIQUE option. . If the UNIQUE option is not specified, only the first consecutive duplicate is applied. All non-consecutive duplicates are applied to the observation in the master data set one after the other, so only the last duplicate is applied. See“Using MODIFY with Duplicate Key Values in the Transaction Data Set” on page 895.
Performs automatic update of like-named variables in the matching observation of the master data set.	Does not perform automatic update of like-named variables in the matching observation of the master data set. See “Performing Automatic Updates for Like-Named Variables Using KEY=” on page 890.

Monitoring Update Processing

The best way to test for values of _IORC_ is with the %SYSRC autocall macro, which enables you to specify a mnemonic name that describes a potential outcome of an I/O operation. The mnemonic codes provided by %SYSRC are part of the SAS autocall macro library. Each mnemonic code represents a specific condition to determine the success of retrieving matching observations. Below is a list of the most common mnemonics that are available to %SYSRC:

Table A3.3 *Common Mnemonics for the %SYSRC Autocall Macro*

Access Method	Mnemonic	Description
MODIFY with KEY= option	_DSENOM	Specifies that the master data set does not contain the observation.
MODIFY with BY statement	_DSENMR	Specifies that the transaction data set observation does not exist in the master data set.
MODIFY with BY statement	_DSEMTR	Specifies that multiple transaction data set observations with a given BY value do not exist in the master data set.
MODIFY with KEY= or BY statement	_SOK	Specifies that the observation was located.

The complete list of mnemonics and their current, corresponding numeric values and descriptions for _IORC_ are contained in the SYSRC member of the autocall macro library. You can view the contents of the SYSRC member in the SAS log by submitting the following code:

```
options source2;
%include sasautos(sysrc);
run;
```

The following example uses %SYSRC to test for successful matching of observations. The example uses the mnemonic _SOK, which indicates that the I/O operation was successful

```
data master;
modify master trans updatemode=nomissingcheck;
by ssn;
if _iorc_=%sysrc(_sok) then replace;
else _error_ =0;
run;
```

Note: Beginning with Version 7, the IORCMSG function returns a formatted error message associated with the current value of _IORC_.

Performing Automatic Updates for Like-Named Variables Using KEY=

Unlike using the BY statement, automatic updates of like-named variables does not occur with the KEY= option. However, when you have many variables to update, it is convenient to have this ability, which you can achieve by using the SET

statement and the POINT= option in conjunction with the MODIFY statement. Observe the order of the following statements:

```
data master;
   set trans;
   modify master key=ssn;
   i+1;
   if _iorc_=%sysrc(_sok) then do;
      13
      set trans point=i;
      replace;
   end;
   else _error_=0;
run;
```

The SET statement loads the values of the `trans` variables into the PDV. Next, the MODIFY statement is executed. The values of all like-named variables in the PDV are overlaid with the values from the `master` data set when a fetch is successful. If a REPLACE or OUTPUT statement were executed at this point, `master` would be updated with the `master` values in the PDV for the like-named variables.

However, the above code issues a second SET statement, using the POINT= option (with the counter `i+1`) to read the same observation in the `trans` data set. This effectively updates the PDV with the like-named variables from the `trans` data set. As discussed earlier, the POINT= option specifies a variable from another data source whose value is the number of an observation that you want to modify in the master data set.

If you have only a few variables to update, you can rely on the normal processing of the KEY= option, along with explicit assignment statements. When using KEY=, the `trans` data set must have variables with the same name as those key variables used to create the index on the `master` data set. The `trans` data set is read with the SET statement and the values for the key variables are loaded into the PDV. The value loaded into the PDV is used against the index values of the `master` data set specified by the KEY= option to fetch the matching observation.

No automatic update of the values for like-named variables occurs between the `trans` and `master` data set. So in order to change the value of a given variable, an explicit assignment of the `master` data set variable is made.

Explicit assignment of like-named variable values presents a problem, however, because the variables you want to assign have the same name in both the `master` and the `trans` data sets. You can use the RENAME= data set option to rename the variables in one data set before assigning the values.

This use of the RENAME= data set option is illustrated in the following example. Both data sets contain variables with the same name, except for the key variables. An explicit assignment statement is used to update the variable `nickname` in the `master` data set with the values of the variable nickname in the `trans` data set. To make the explicit assignment, the RENAME= data set option is used to rename the variable `nickname` to `tnickname` in the `trans` data set. The `nickname` variable in the `master` data set is set equal to the `tnickname` variable in the `trans` data set

Table A3.4 *Input Data Sets Containing Like-named Variables*

data master(index=(ssn));	data trans;

```
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
160-58-1223 Kathryn
134-56-9094 Megan
;
```

```
   input ssn : $11. nickname $;
datalines;
161-60-5881 Josh
160-58-1223 Kate
134-56-9094 Meg
142-67-9888 Bill
;
```

```
data master;
   set trans(rename=(nickname=tnicknam));
   modify master key=ssn;
   if _iorc_=%sysrc(_sok) then do;
      nickname=tnicknam;
      replace;
   end;
   else _error_ = 0;
run;
```

Using MODIFY in a SAS/SHARE Environment

In a SAS/SHARE environment, the control level for updating using the MODIFY statement is dependent on the access method being used:

- A MODIFY statement without the POINT= option or KEY= option has a control level of RECORD, which means that other tasks can read or update the data set but no SAS task can open it for output.
- A MODIFY statement with the POINT= option or KEY= option has a control level of MEMBER, which means that other tasks cannot access the data.

The differences between the MODIFY and SET statements are listed below:

Table A3.5 Comparing MODIFY to SET Statement

MODIFY Statement	SET Statement
Updates the original data set opened in update mode without creating a copy.	Creates a temporary data set with the updates applied. Upon successful completion of the DATA step, the original data set is replaced with the temporary data set if the DATA statement and the SET statement refer to the same data set name. If the data set name is different, the temporary data set is renamed to that existing in the DATA statement.
Generates an implicit REPLACE statement at the bottom of the DATA step.	Generates an implicit OUTPUT statement at the bottom of the DATA step.

MODIFY Statement	**SET Statement**
Variables cannot be added or dropped from the modified data set. The header information for a data set opened for update cannot be modified.	The output data set can contain new variables and variables existing in the lookup and primary data sets.
The REPLACE statement updates the current observation of the original data set.	The REPLACE statement is not valid since a copy of the original observation is updated and OUTPUT.
The original data set is updated with missing values from the transaction data set using the UPDATEMODE= option.	The UPDATEMODE= option is not valid since a copy of the original observation is updated with missing values and OUTPUT.

Understanding the KEY= Option

Both the MODIFY and SET statements support the KEY= option, which allows applications to specify an index in order to retrieve particular observations in a data set based on the indexed values. Using KEY= with the MODIFY statement is explained earlier in this appendix for the Direct Access by Index Values method. The next topic explains using KEY= with the SET statement. Then subsequent topics cover issues regarding KEY= for both the MODIFY and SET statements, such as when duplicate values exist for the key variable.

Using KEY= with the SET Statement

This topic does not explain the SET statement in general but does explain using the KEY= option in the SET statement. For more details about the SET statement, see “SET Statement” in *SAS DATA Step Statements: Reference*.

The syntax for the SET statement is as follows:

SET *SAS-data-sets* <(*data-set-options*)><NOBS=*variable*> <END=*variable*> <POINT=*variable* | KEY=*index*> </ UNIQUE> ;

KEY= provides non-sequential access to observations in a data set through an index created for one or more variables. The index can be either simple or composite. You cannot, however, use KEY= with the POINT= option.

As with the MODIFY statement, using the _IORC_ automatic variable in conjunction with the %SYSRC autocall macro provides more error-handling information. When you use the SET statement with KEY=, _IORC_ is created and set to a return code that indicates the status of the most recent I/O operation performed on an observation in the data set. If the KEY= value is not found in the master data set,

IORC returns a numeric value that corresponds to the %SYSRC autocall macro's mnemonic _DSENOM and the automatic variable _ERROR_ is set to 1.

Note: When issuing multiple SET statements using the KEY= option, you must perform separate error checking of the automatic variable _IORC_on each data set. That is, in order to produce an accurate output data set, test the _IORC_ variable following each SET statement using the KEY= option.

Using the SET statement with KEY= supports the concept of lookup values. The lookup value can be provided through another SAS data set, an external file, a view, or a FRAME entry. For Version 6, the lookup data set must be a native SAS data set. Version 7 supports SAS/ACCESS views and the DBMS engine for the LIBNAME statement.

```
data combine;
   set lookup;
   set primary key=partno;
   select(_iorc_);
   when (%sysrc(_sok)) do;
      output;
   end;
   when (%sysrc(_dsenom)) do;
      _error_ = 0;
   end;
   otherwise;
   end;
run;
```

The lookup value from the lookup data source is used as a key to locate observations in the primary or `master` data set. This primary data set must be indexed (either simple or composite) and specified with the KEY= option. The lookup values must be provided through variables named the same as the key variables in the primary data set. For example, if the lookup data source is a SAS data set, it must contain variables with the same name as those defined to the index of the primary data set. Once the observation is successfully fetched from the primary data set, an action such as OUTPUT can be performed on the observation.

Note: If a DROP or KEEP is not specified, using the KEY= option combines all variables on the lookup data set and the primary data set. That is, the output data set contains not only the variables from the lookup data set, but also all the variables in the primary data set. This does not happen with KEY= and the MODIFY statement.

Handling Duplicate Values for Key Variable

When using the KEY= option to create an index, multiple observations might contain the same values for the key variable in the input data sets or the data set supplying the index values.

Using MODIFY with Duplicate Key Values in the Transaction Data Set

Having duplicate values for the key variable in the transaction data set can affect the results when they are applied to the master data set. In addition, whether the duplicate values are consecutive or non-consecutive also affect the results.

For example, consider the following two programs, which are identical except for the order of the duplicate values for the key variable ssn in the `trans` data set. Notice that in Program 1, the `trans` data set has duplicate key values of **`160-58-1223`**, but they are not consecutive; Program 2 also has the duplicate values, but they are consecutive.

Table A3.6 *How Ordering of Duplicate KEY= Values Affects Output*

<table>
<tr><th>Program 1</th><th>Program 2</th></tr>
<tr><td><pre>
data master(index=(ssn));
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
160-58-1223 Kathryn
134-56-9094 Megan
;

data trans;
   input ssn : $11. tnicknam $;
datalines;
161-60-5881 Josh
160-58-1223 Kathy
134-56-9094 Meg
142-67-9888 Bill
160-58-1223 Kate
;

data master;
   set trans;
   modify master key=ssn;
   if _iorc_=%sysrc(_sok) then do;
      nickname=tnicknam;
      replace;
   end;
   else_error_=0;
run;
</pre></td><td><pre>
data master(index=(ssn));
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
160-58-1223 Kathryn
134-56-9094 Megan
;

data trans;
   input ssn : $11. tnicknam $;
datalines;
161-60-5881 Josh
160-58-1223 Kathy
160-58-1223 Kate
134-56-9094 Meg
142-67-9888 Bill
;

data master;
   set trans;
   modify master key=ssn;
   if _iorc_=%sysrc(_sok) then do;
      nickname=tnicknam;
      replace;
   end;
   else_error_=0;
run;
</pre></td></tr>
</table>

The following PRINT procedure shows the results of the above programs.

Note: The DATA= option is used with PROC PRINT to specify the `master` data set. If DATA= is not specified, PROC PRINT displays the last created data set opened for output, which might not be the last one you opened for update.

```
proc print data=master;
run;
```

- From Program 1, the value for `nickname` in the resulting `master` data set is **`Kate`**, which is the value of the last observation in the `trans` data set with the corresponding Social Security number.

```
OBS   SSN           nickname
1     161-60-5881   Josh
2     160-58-1223   Kate
3     134-56-9094   Meg
```

- From Program 2, the consecutive value of is Kate for ssn in the `trans` data set. That value does not update the `master` data set. The consecutive record of Kate actually causes a non-match to occur, and only the first observation in `trans` with the corresponding value is applied.

```
OBS   SSN           nickname
1     161-60-5881   Josh
2     160-58-1223   Kathy
3     134-56-9094   Meg
```

The order of duplicate key values in `trans` affects the results of the `master` data set. When searching the index, SAS begins at the top of the index only when the value of the key variable changes in the `trans` data set.

If the duplicate key values in the `trans` data set are *not consecutive*, the search starts from the top of the index in both searches for Social Security number **`160-58-1223`**. Therefore, the one and only matched value in the `master` data set is located and updated both times, so the value in the last duplicate observation is applied.

For *consecutive* duplicate key variable values in the `trans` data set, when the first value of **`160-58-1223`** is supplied by the `trans` data set, the value for the key variable changes – the index search of the `master` data set begins at the top and the observation is found. The `nickname` variable in `master` is updated to **`Kathy`**. When the consecutive value of **`160-58-1223`** is supplied by the `trans` data set, the value does not change. Therefore, the search does not begin at the top of the `master` index; rather, it begins from the current position in the index structure. The observation is not found, and an update is not performed for the subsequent duplicate observations. Only the value in the first duplicate is applied.

To assure the same result whether duplicates in `trans` are consecutive or not, you can force the software to begin the search at the top of the index by specifying the UNIQUE option in the MODIFY statement. The UNIQUE option specifies to search from the top of the index, regardless of whether the key variable value changes from one iteration to the next.

The following code uses the UNIQUE option to produce the same results for each program. For readability purposes, the IF statement is coded as a SELECT statement instead.

Table A3.7 *Input Master and Transition Data Sets*

`data master(index=(ssn));`	`data trans;`

```
   input ssn : $11. Nickname $;
datalines;
161-60-5881 Joshua
160-58-1223 Kathryn
134-56-9094 Megan
;
```

```
   input ssn : $11. tnicknam $;
datalines;
161-60-5881 Josh
160-58-1223 Kathy
160-58-1223 Kate
134-56-9094 Meg
142-67-9888 Bill
;
```

```
data master;
   set trans;
   modify master key=ssn/unique;
   select (_iorc_);
   when (%sysrc(_sok)) do;
      nickname=tnicknam;
      replace master;
   end;
   when (%sysrc(_dsenom)) do;
      _error_=0;
   end;
   otherwise;
   end;

proc print data=master;
run;
```

OBS	SSN	nickname
1	161-60-5881	Josh
2	160-58-1223	Kate
3	134-56-9094	Meg

Using MODIFY with Duplicate Key Values in the Master Data Set

Having duplicate values for the key variable in the master data set can affect the results if you want to update all duplicates in the master data set with a corresponding unique value in the transaction data set.

For example, the `trans` data set observation with the ssn value of **161-60-5881** updates only the first corresponding observation of the ssn variable in the `master` data set to the value of Josh. Notice that in the PROC PRINT results, the highlighted record is updated.

Table A3.8 *Input Data Sets Master and Trans*

```
data master(index=(ssn));
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
161-60-5881 Joshua
160-58-1223 Kathryn
```

```
data trans;
   input ssn : $11. nickname $;
datalines;
161-60-5881 Josh
160-58-1223 Kathy
134-56-9094 Meg
```

134-56-9094 Megan ;	142-67-9888 Bill ;

```
data master;
   set trans(rename=(nickname=tnicknam));
   modify master key=ssn;
   select (_iorc_);
   when (%sysrc(_sok)) do;
      nickname=tnicknam;
      replace master;
   end;
   when (%sysrc(_dsenom)) do;
      _error_=0;
   end;
   otherwise;
   end;

proc print data=master;
run;
```

OBS	SSN	nickname
1	161-60-5881	Josh
2	161-60-5881	Joshua
3	160-58-1223	Kathy
4	134-56-9094	Meg

To update variable Nickname for multiple observations in the `master` data set with the unique, corresponding observation in the `trans` data set, force a continuous search of the `master` data set's index file using a DO UNTIL loop until a non-match condition is encountered. Notice that the highlighted observations from the PRINT procedure indicate that all corresponding records of the ssn value **`161-60-5881`** are now updated in the `master` data set.

Table A3.9 Input Data Sets Master and Trans

data master(index=(ssn));
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
161-60-5881 Joshua
160-58-1223 Kathryn
134-56-9094 Megan
;
	data trans;
	   input ssn : $11. tnicknam $;
	datalines;
	161-60-5881 Josh
	160-58-1223 Kathy
	134-56-9094 Meg
	142-67-9888 Bill
	;

```
data master;
   set trans;
   do until (_iorc_=%sysrc(_dsenom));
      modify master key=ssn;
      select (_iorc_);
         when (%sysrc(_sok)) do;
            nickname=tnicknam;
            replace master;
         end;
```

```
         when (%sysrc(_dsenom)) do;
            _error_=0;
         end;
         otherwise;
      end;
   end;

   proc print data=master;
   run;
```

OBS	SSN	nickname
1	161-60-5881	Josh
2	161-60-5881	Josh
3	160-58-1223	Kathy
4	134-56-9094	Meg

Using MODIFY with Duplicate Key Values in the Master Data Set and the Transaction Data Set

If duplicate values of the key variable exist in both the master data set and the transaction data set, updating each corresponding master observation with each corresponding transaction observation requires additional BY processing.

When consecutive duplicate values of ssn are supplied by the transaction data set, you have seen how the UNIQUE option forces a search to start from the top of the index. For this situation, however, if you use the UNIQUE option, the same observation in the `master` data set is located and updated. Therefore, the duplicate observations in `master` are not updated.

You must force the search to start at the top of the index by changing the key variable without using the UNIQUE option, and it must happen between each observation of the same ssn. Because you need to change the key variable between consecutive values of ssn in the `trans` data set, use BY processing to determine whether more observations of the same `ssn` in the `trans` data set are supplied. Therefore, the `trans` data set must be sorted for BY processing. To continue the search in the index for all corresponding matches, use a DO UNTIL statement to process until a non-match situation is encountered in `master`.

The following example shows how to temporarily change the key variable value to a value that is not located in the `master` data set.

Table A3.10 *Input Data Sets Master and Trans*

```
data master(index=(ssn));
   input ssn : $11. nickname $;
datalines;
161-60-5881 Joshua
161-60-5881 Joshua
160-58-1223 Kathryn
160-58-1223 Kathryn
134-56-9094 Megan
```

```
data trans;
   input ssn : $11. tnicknam $;
datalines;
161-60-5881 Josh
160-58-1223 Kathy
160-58-1223 Kate
134-56-9094 Meg
142-67-9888 Bill
```

;   ;

```
proc sort data=trans;
   by ssn;

data master;
   set trans;
   by ssn;
   dummy=0;
   do until (_iorc_=%sysrc(_dsenom));
      if dummy then ssn='999-99-9999';
      modify master key=ssn;
      select (_iorc_);
         when (%sysrc(_sok)) do;
            nickname=tnicknam;
            replace master;
         end;
         when (%sysrc(_dsenom)) do;
            _error_=0;
            if not last.ssn and not dummy then do;
               dummy=1;
               _iorc_=0;
            end;
         end;
         otherwise;
      end;
   ;

proc print data=master;
run;
```

OBS	SSN	nickname
1	161-60-5881	Josh
2	161-60-5881	Josh
3	160-58-1223	Kate
4	160-58-1223	Kate
5	134-56-9094	Meg

Using SET with Duplicate Key Values in the Lookup Data Set

Like KEY= with MODIFY, execution of the SET statement with the KEY= option forces a search to begin at the top of the index only when the value of the KEY= variable changes in the lookup data set between fetch operations. If duplicate values for the key variable exist consecutively in the lookup data set, the value does not change between executions. Therefore, to force the search to begin at the top, use the UNIQUE option. With non-consecutive duplicate values, the search begins at the top of the index structure.

Using SET with Duplicate Key Values in the the Primary Data Set

Duplicate values for the key variable in the primary data set have the same effect with SET as previously described with MODIFY. The software locates and returns the first occurrence of the key variable in the index, unless the statement with the KEY= option is forced to execute again using the same key value. As with MODIFY, you can force the continuation of this search of the primary index structure with the DO UNTIL statement. Notice that the bolded observations from the PRINT procedure indicate that all corresponding observations for Partno A066 exist in the output data set.

Table A3.11 *Input Data Sets Lookup and Primary*

data lookup;
   input partno $ quantity;
datalines;
A066 8
A220 2
A812 10
;
	data primary(index=(partno));
	   input partno $ desc $;
	datalines;
	A021 motor
	A033 bolt
	A043 nut
	A055 radiator
	A066 hose
	A066 hose2
	A066 hose3
	A078 knob
	A220 switch
	A223 bridge
	;

```
data combine;
   set lookup;
   do until(_iorc_=%sysrc(_dsenom));
      set primary key=partno;
      select(_iorc_);
         when (%sysrc(_sok)) do;
            output;
         end;
         when (%sysrc(_dsenom)) do;
            _error_=0;
         end;
         otherwise;
      end;
   end;

proc print data=combine;
run;
```

OBS	PARTNO	QUANTITY	DESC
1	A066	8	hose
2	A066	8	hose2
3	A066	8	hose3
4	A220	2	switch

Using SET with Duplicate Key Values in the Primary Data Set and the Lookup Data Set

To combine data sets when duplicate values for a key variable exist in both the primary data set and the lookup data set, use the same concept discussed for the MODIFY statement. See Using MODIFY with Duplicate Key Values in `master` data set and `transaction` data set.

Table A3.12 Input Data Sets Lookup and Primary

```
data lookup;
   input partno $ district;
datalines;
A066 1
A066 2
A220 2
A812 10
;
```

```
data primary(index=(partno));
   input partno $ desc $;
datalines;
A021 motor
A033 bolt
A043 nut
A055 radiator
A066 hose
A066 hose2
A066 hose3
A078 knob
A220 switch
223 bridge
;
```

```
proc sql;
   create table shiplst as
   select a.*, b.desc
   from lookup as a, primary as b
   where a.partno=b.partno;
quit;

proc print data=shiplst;
run;
```

OBS	PARTNO	DISTRICT	DESC
1	A066	1	hose
2	A066	2	hose
3	A066	1	hose2
4	A066	2	hose2
5	A066	1	hose3
6	A066	2	hose3
7	A220	2	switch

Combining Data Sets with Non-Matching Observations

Each time that the SET statement is executed, one observation is read from the current data set opened for input into the PDV. The SET statement reads all variables and all observations from the input data sets unless you specify to do otherwise. You can use multiple SET statements to perform one-to-one reading (also called one-to-one matching) of the specified data sets. The new data set contains all the variables from all the input data sets. The number of observations in the new data set is the number of observations in the smallest original data set. If the data sets contain common variables, the values that are read in from the last data set replace those read in from previous ones.

When combining SAS data sets with multiple SET statements, SAS does not reinitialize existing variables to missing for each DATA step iteration. The nonmissing value on the output data set is the retained value in the PDV from the most recent match that occurred.

In the following example, the observation in the lookup data set with the value A812 for Partno is not located in the primary data set. The DESC variable loaded in the PDV currently holds the value switch from the last successful match for Partno value A220. Because a match does not occur, this DESC variable value is not overwritten in the PDV with a new value for Partno A812.

Table A3.13 *Input Data Sets Lookup and Primary*

```
data lookup;
   input partno $ quantity;
datalines;
A066 8
A220 2
A812 10
;
```

```
data primary(index=(partno));
   input partno $ desc $;
datalines;
A021 motor
A033 bolt
A043 nut
A055 radiator
A066 hose
A078 knob
A220 switch
A223 bridge
;
```

```
data combine;
   set lookup;
   set primary key=partno;
   select(_iorc_);
      when (%sysrc(_sok)) do;
         output;
      end;
      when (%sysrc(_dsenom)) do;
         _error_=0;
         *desc = ' ';
         output;
      end;
```

```
      otherwise;
   end;

proc print data=combine;
run;
```

OBS	PARTNO	QUANTITY	DESC
1	A066	8	hose
2	A220	2	switch
3	A812	10	switch

To write a missing value for DESC that does not exist in the primary data set to the output data set, you must execute an explicit assignment statement. In the assignment statement, set the DESC value equal to missing. This overwrites the retained value in the PDV with the desired value of missing. In the example code above, uncomment the DESC= code for the desired output.

Adding Variables to the Output Data Set

Using the SET statement to open a data set for output enables you to add a new variable through an assignment statement to an output data set. When this occurs, the new variable is reset to missing for each iteration of the DATA step. The following example shows that the STATUS variable is automatically set to missing without assigning a missing value, when a non-match occurs in an observation from the primary data set.

Table A3.14 *Input Data Sets Lookup and Primary*

data lookup;
   input partno $ quantity;
datalines;
A066 8
A220 2
A812 10
;
	data primary(index=(partno));
	   input partno $ desc $;
	datalines;
	A021 motor
	A033 bolt
	A043 nut
	A055 radiator
	A066 hose
	A078 knob
	A220 switch
	A223 bridge
	;

```
data combine;
   set lookup;
   set primary key=partno;
   select(_iorc_);
      when (%sysrc(_sok)) do;
         status="available";
         output;
      end;
      when (%sysrc(_dsenom)) do;
         desc=' ';
```

```
          _error_=0;
          output;
       end;
       otherwise;
    end;

 proc print data=combine;
 run;
```

OBS	PARTNO	QUANTITY	DESC	STATUS
1	A066	8	hose	available
2	A220	2	switch	available
3	A812	10		

Using SET with KEY= in a SAS/SHARE Environment

In a SAS/SHARE environment, the control level for updating using SET with KEY= is MEMBER. This means that other tasks can read the data set, but no SAS task can open it for update or output.

Version 7 and Version 8 Considerations

Using the DBKEY= SAS/ACCESS Data Set Option with KEY=

SAS/ACCESS software provides several data set options to improve performance when accessing data in a DBMS. The DBKEY= data set option enables you to specify variable names to use as an index. The software uses the specified variable names as an index by constructing and passing a WHERE expression to the DBMS.

DBKEY= can be used to improve performance just as indexing a SAS data file can improve performance. For example, to improve the performance, the DBKEY= option can be used in a DATA step with the KEY= option in a SET statement. Note that you must specify the keyword DBKEY as the value of the KEY= option.

The following DATA step creates a new data file by joining data file `keyvalues` with the DBMS table `mytable`. The software uses the variable `deptno` with the DBKEY= data set option to cause a WHERE expression to be passed to the DBMS. Performance benefits might occur if the DBMS optimizer selects to optimize the WHERE expression with an existing index on the DBMS table.

```
data sasuser.new;
   set sasuser.keyvalues;
   set dblib.mytable (dbkey=deptno) key=dbkey;
run;
```