



# SAS<sup>®</sup> 9.4 Language Reference: Concepts, Sixth Edition

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® 9.4 Language Reference: Concepts, Sixth Edition*. Cary, NC: SAS Institute Inc.

**SAS® 9.4 Language Reference: Concepts, Sixth Edition**

Copyright © 2016, SAS Institute Inc., Cary, NC, USA

ISBN 978-1-62960-821-1 (Paperback)  
ISBN 978-1-62960-822-8 (PDF)

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

November 2019

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-P9:Ircon

---

# Contents

*What's New in the 9.4 Base SAS Language Reference: Concepts* ..... xi

## PART 1 SAS System Concepts 1

<b>Chapter 1 / Essential Concepts of Base SAS Software</b> .....	<b>3</b>
What Is SAS? .....	3
Overview of Base SAS Software .....	4
Components of the SAS Language .....	5
Ways to Run Your SAS Session .....	8
Customizing Your SAS Session .....	11
Conceptual Information about Base SAS Software .....	12
<b>Chapter 2 / SAS Processing</b> .....	<b>13</b>
Definition of SAS Processing .....	13
Types of Input to a SAS Program .....	14
The DATA Step .....	16
The PROC Step .....	17
SAS Processing Restrictions for Servers in a Locked-Down State .....	18
<b>Chapter 3 / Rules for Words and Names in the SAS Language</b> .....	<b>21</b>
Words in the SAS Language .....	21
Names in the SAS Language .....	24
<b>Chapter 4 / SAS Variables</b> .....	<b>37</b>
Definition of SAS Variables .....	38
SAS Variable Attributes .....	38
Ways to Create Variables .....	42
Variable Type Conversions .....	51
Aligning Variable Values in SAS Output .....	52
Reordering Variables in SAS Output .....	53
Automatic Variables .....	54
SAS Variable Lists .....	55
The OF Operator with Functions and Variable Lists .....	63
Dropping, Keeping, and Renaming Variables .....	64
Encrypting Variable Values .....	67
Numerical Accuracy in SAS Software .....	72
<b>Chapter 5 / Missing Values</b> .....	<b>95</b>
Definition of Missing Values .....	95
Creating Special Missing Values .....	96
Order of Missing Values .....	97
When Variable Values Are Automatically Set to Missing by SAS .....	98
When Missing Values Are Generated by SAS .....	100
Working with Missing Values .....	102

<b>Chapter 6 / Expressions</b> .....	<b>105</b>
Definitions for SAS Expressions .....	106
Examples of SAS Expressions .....	106
SAS Constants in Expressions .....	107
SAS Variables in Expressions .....	113
SAS Functions in Expressions .....	114
SAS Operators in Expressions .....	114
<b>Chapter 7 / Dates, Times, and Intervals</b> .....	<b>127</b>
About SAS Date, Time, and Datetime Values .....	127
About Date and Time Intervals .....	143
<b>Chapter 8 / Error Processing and Debugging</b> .....	<b>155</b>
Types of Errors in SAS .....	155
Error Processing in SAS .....	165
Debugging Logic Errors in the DATA Step .....	174
<b>Chapter 9 / SAS Output</b> .....	<b>175</b>
Definitions for SAS Output .....	175
Routing and Customizing SAS Output .....	177
Sample SAS Output .....	182
The SAS Log .....	184
<b>Chapter 10 / By-Group Processing in SAS Programs</b> .....	<b>195</b>
Definition of BY-Group Processing .....	195
References for BY-Group Processing .....	195
<b>Chapter 11 / WHERE-Expression Processing</b> .....	<b>197</b>
Definition of WHERE-Expression Processing .....	197
Where to Use a WHERE Expression .....	198
Syntax of WHERE Expression .....	199
Combining Expressions By Using Logical Operators .....	209
Improving Performance of WHERE Processing .....	210
Processing a Segment of Data That Is Conditionally Selected .....	211
Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement .....	214
<b>Chapter 12 / Optimizing System Performance</b> .....	<b>217</b>
Definitions for Optimizing System Performance .....	217
Collecting and Interpreting Performance Statistics .....	218
Techniques for Optimizing I/O .....	219
Techniques for Optimizing Memory Usage .....	226
Techniques for Optimizing CPU Performance .....	227
Calculating Data Set Size .....	229
<b>Chapter 13 / Support for Parallel Processing</b> .....	<b>231</b>
Overview .....	231
What Is Threading Technology in SAS? .....	232
How Is Threading Controlled in SAS? .....	233
Threading in Base SAS .....	233
SAS/ACCESS Engines .....	236
SAS Scalable Performance Data Server .....	237
SAS Intelligence Platform .....	237
SAS High-Performance Analytics Portfolio of Products .....	238
SAS Grid Manager .....	239
SAS In-Database Technology .....	240
SAS In-Memory Analytics Technology .....	240

SAS High-Performance Analytics Product Integration .....	242
SAS Viya .....	244
<b>Chapter 14 / The SAS Registry .....</b>	<b>245</b>
Introduction to the SAS Registry .....	245
Managing the SAS Registry .....	248
Configuring Your Registry .....	258
<b>Chapter 15 / Printing with SAS .....</b>	<b>263</b>
Universal Printing .....	265
Configuring Universal Printing Using the Windowing Environment .....	283
System Options That Control Universal Printing .....	300
Managing Universal Printers Using the PRTDEF Procedure .....	302
Forms Printing .....	308
Using Fonts with Universal Printers and SAS/GRAFH Devices .....	309
Creating EMF (Enhanced Metafile Format) Graphics Using Universal Printing .....	328
Creating GIF Images Using Universal Printing .....	331
Creating PCL (Printer Command Language) Files Using Universal Printing .....	333
Creating PDF Files Using Universal Printing .....	335
Creating PNG (Portable Network Graphics) Files Using Universal Printing .....	338
Creating PostScript Files Using Universal Printing .....	341
Creating SVG (Scalable Vector Graphics) Files Using Universal Printing .....	343
Creating TIFF Images Using Universal Printing .....	372
Creating Animated GIF Images and SVG Documents .....	375

## PART 2 Windowing Environment 383

<b>Chapter 16 / Introduction to the SAS Windowing Environment .....</b>	<b>385</b>
What Is the SAS Windowing Environment? .....	385
Main Windows in the SAS Windowing Environment .....	386
Navigating in the SAS Windowing Environment .....	394
Getting Help in SAS .....	399
List of SAS Windows and Window Commands .....	401
<b>Chapter 17 / Managing Your Data in the SAS Windowing Environment .....</b>	<b>405</b>
Introduction to Managing Your Data in the SAS Windowing Environment .....	406
Managing Data with SAS Explorer .....	406
Working with VIEWTABLE .....	411
Subsetting Data By Using the WHERE Expression .....	421
Exporting a Subset of Data .....	425
Importing Data into a Table .....	428

## PART 3 SAS Cloud Analytic Services 431

<b>Chapter 18 / Introduction to SAS Cloud Analytic Services .....</b>	<b>433</b>
What is SAS Cloud Analytic Services? .....	433
What Does This Mean for the SAS 9 Programmer? .....	433

<b>Chapter 19 / SAS Language Support for CAS .....</b>	<b>435</b>
SAS Language Elements for CAS .....	435
CAS-specific Language Elements .....	437

## PART 4 DATA Step Concepts 439

<b>Chapter 20 / DATA Step Processing .....</b>	<b>441</b>
Why Use a DATA Step? .....	441
Overview of DATA Step Processing .....	442
Processing a DATA Step: A Walk-through .....	445
About DATA Step Execution .....	450
About Creating a SAS Data Set with a DATA Step .....	456
Writing a Report with a DATA Step .....	460
The DATA Step and ODS .....	467
DATA Step Processing Time .....	468
<b>Chapter 21 / Reading Raw Data .....</b>	<b>471</b>
Definition of Reading Raw Data .....	471
Ways to Read Raw Data .....	472
Types of Data .....	473
Sources of Raw Data .....	476
Reading Raw Data with the INPUT Statement .....	477
How SAS Handles Invalid Data .....	483
Reading Missing Values in Raw Data .....	484
Reading Binary Data .....	485
Reading Column-Binary Data .....	487
<b>Chapter 22 / BY-Group Processing in the DATA Step .....</b>	<b>491</b>
Definitions for BY-Group Processing .....	491
Syntax for BY-Group Processing .....	492
Understanding BY Groups .....	493
Invoking BY-Group Processing .....	496
Determining Whether the Data Requires Preprocessing for BY-Group Processing .....	496
Preprocessing Input Data for BY-Group Processing .....	497
FIRST. and LAST. DATA Step Variables .....	498
Processing BY-Groups in the DATA Step .....	502
<b>Chapter 23 / Reading, Combining, and Modifying SAS Data Sets .....</b>	<b>509</b>
Definitions for Reading, Combining, and Modifying SAS Data Sets .....	509
Overview of Tools .....	510
Reading SAS Data Sets .....	510
Combining SAS Data Sets: Basic Concepts .....	512
Combining SAS Data Sets: Methods .....	523
Error Checking When Using Indexes to Randomly Access or Update Data .....	555
<b>Chapter 24 / Using DATA Step Component Objects .....</b>	<b>565</b>
Introduction to DATA Step Component Objects .....	565
Using the Hash Object .....	566
Using the Hash Iterator Object .....	579
Using the Java Object .....	582

<b>Chapter 25 / Array Processing .....</b>	<b>603</b>
Definitions for Array Processing .....	603
A Conceptual View of Arrays .....	604
Syntax for Defining and Referencing an Array .....	605
Processing Simple Arrays .....	606
Variations on Basic Array Processing .....	610
Multidimensional Arrays: Creating and Processing .....	612
Specifying Array Bounds .....	614
Examples of Array Processing .....	616

## PART 5 SAS Files Concepts 621

<b>Chapter 26 / SAS Libraries .....</b>	<b>623</b>
Definition of a SAS Library .....	623
Library Engines .....	625
Library Names .....	625
Library Concatenation .....	629
Permanent and Temporary Libraries .....	631
Definition of a Metadata-Bound Library .....	632
SAS System Libraries .....	632
Sequential Data Libraries .....	635
Tools for Managing Libraries .....	636
<b>Chapter 27 / SAS Data Sets .....</b>	<b>639</b>
Definition of a SAS Data Set .....	639
Descriptor Information for a SAS Data Set .....	640
Data Set Names .....	642
Data Set Lists .....	644
Special SAS Data Sets .....	645
Sorted Data Sets .....	646
Tools for Managing Data Sets .....	652
Viewing and Editing SAS Data Sets .....	653
<b>Chapter 28 / SAS Data Files .....</b>	<b>655</b>
Definition of a SAS Data File .....	656
Differences between Data Files and SAS Views .....	657
Understanding the Observation Count in a SAS Data File .....	658
Understanding an Audit Trail .....	662
Understanding Generation Data Sets .....	672
Understanding Integrity Constraints .....	679
Understanding SAS Indexes .....	692
Extended Attributes .....	716
Compressing Data Files .....	717
<b>Chapter 29 / SAS Views .....</b>	<b>721</b>
SAS Views .....	721
Benefits of Using SAS Views .....	722
When to Use SAS Views .....	723
DATA Step Views .....	724
PROC SQL Views .....	729
Comparing DATA Step and PROC SQL Views .....	729
SAS/ACCESS Views .....	730

<b>Chapter 30 / Stored Compiled DATA Step Programs .....</b>	<b>731</b>
Definition of a Stored Compiled DATA Step Program .....	731
Uses for Stored Compiled DATA Step Programs .....	732
Restrictions and Requirements for Stored Compiled DATA Step Programs .....	732
How SAS Processes Stored Compiled DATA Step Programs .....	732
Creating a Stored Compiled DATA Step Program .....	733
Executing a Stored Compiled DATA Step Program .....	735
Differences between Stored Compiled DATA Step Programs and DATA Step Views .....	739
Example of DATA Step Program .....	739
<b>Chapter 31 / DICTIONARY Tables .....</b>	<b>741</b>
Definition of a DICTIONARY Table .....	741
How to View DICTIONARY Tables .....	742
<b>Chapter 32 / SAS Catalogs .....</b>	<b>747</b>
Definition of a SAS Catalog .....	747
SAS Catalog Names .....	748
Tools for Managing SAS Catalogs .....	748
Profile Catalog .....	749
Catalog Concatenation .....	751
<b>Chapter 33 / About SAS/ACCESS Software .....</b>	<b>757</b>
Definition of SAS/ACCESS Software .....	757
Dynamic LIBNAME Engine .....	758
SQL Procedure Pass-Through Facility .....	759
ACCESS Procedure and Interface View Engine .....	760
DBLOAD Procedure .....	761
Interface DATA Step Engine .....	762
<b>Chapter 34 / Processing Data Using Cross-Environment Data Access (CEDA) .....</b>	<b>765</b>
Definition of Cross-Environment Data Access (CEDA) .....	765
Advantages of CEDA .....	766
SAS File Processing with CEDA .....	767
Alternatives to Using CEDA .....	772
Creating Files in a Different Data Representation .....	773
Examples of Using CEDA .....	773
<b>Chapter 35 / Cross-Release Compatibility and Migration .....</b>	<b>779</b>
Introduction to Cross-Release Compatibility and Migration .....	779
Accessing a File That Was Created in a Previous Release .....	779
Using SAS Files in a Previous Release .....	780
SAS Library Engines and the SAS File Format .....	783
<b>Chapter 36 / File Protection .....</b>	<b>785</b>
Definition of a Password .....	786
Assigning Passwords .....	787
Removing or Changing Passwords .....	789
Using Password-Protected SAS Files in DATA and PROC Steps .....	789
How SAS Handles Incorrect Passwords .....	790
Assigning Complete Protection with the PW= Data Set Option .....	790
Encoded Passwords .....	791
Using Passwords with Views .....	792
SAS Data File Encryption .....	794
Blotting Passwords and Encryption Key Values .....	798
Metadata-Bound Libraries .....	801

<b>Chapter 37 / SAS Engines</b>	<b>803</b>
Definition of a SAS Engine	803
Specifying an Engine	803
How Engines Work with SAS Files	804
Engine Characteristics	805
About Library Engines	808
Special-Purpose Engines	811
<b>Chapter 38 / SAS File Management</b>	<b>815</b>
Improving Performance of SAS Applications	815
Moving SAS Files between Operating Environments	815
Repairing Damaged SAS Files	816
<b>Chapter 39 / External Files</b>	<b>821</b>
Definition of External Files	821
Referencing External Files Directly	822
Referencing External Files Indirectly	822
Referencing Many External Files Efficiently	823
Referencing External Files with Other Access Methods	824
Working with External Files	826

## PART 6 Industry Protocols Used in SAS 829

<b>Chapter 40 / The SMTP E-Mail Interface</b>	<b>831</b>
Sending E-Mail through SMTP	831
System Options That Control SMTP E-Mail	832
Statements That Control SMTP E-mail	833
<b>Chapter 41 / Universal Unique Identifiers</b>	<b>835</b>
Universally Unique Identifiers and the Object Spawner	835
Using SAS Language Elements to Assign UUIDs	838
<b>Chapter 42 / Internet Protocol Version 6 (IPv6)</b>	<b>839</b>
Overview of IPv6	839
IPv6 Address Format	840
Examples of IPv6 Addresses	840
Fully Qualified Domain Names (FQDN)	841

**x** *Contents*

# What's New in the 9.4 Base SAS Language Reference: Concepts

---

## Overview

SAS 9.4 has the following changes and enhancements:

- in SAS 9.4M6, [Noto Sans TrueType fonts](#) are added to support languages for harmonious web display.
- in SAS 9.4M6 and later, be aware of a [restriction for PROC SQL views](#).
- in SAS 9.4M5, a new [AES2](#) encryption is supported.
- in SAS 9.4M5, you can access [SAS Cloud Analytics Services](#) from your SAS session when you license and install SAS Viya 3.2 and later releases.
- in SAS 9.4M5, new [AvenirNextforSAS](#) and [HelveticaNeueforSAS](#) fonts replace the Avenir Next LT W04, Avenir Next Cyr W04, and Helvetica LT Pro fonts.
- in SAS 9.4M4, new [TrueType fonts](#) are added.
- in SAS 9.4M3, new [Avenir Next TrueType fonts](#) are added.
- in SAS 9.4M2, it is no longer necessary to use the UUID Generator Daemon to generate UUIDs for SAS sessions that execute on UNIX hosts. See “[What Is the Object Spawner?](#)” on page [835](#) for more information.
- in SAS 9.4M2, the [LOCKDOWN](#) feature is enhanced.
- in SAS 9.4M2, font slanting and emboldening features are added. For more information, see “[Slanting and Emboldening Fonts](#)”.
- in SAS 9.4M1, the [LOCKDOWN](#) statement and system option are added.
- in SAS 9.4, the following are new:
  - [universal printing](#) enhancements to support additional graphic output types and animation for GIF and SVG files
  - [buffer size specification support](#) for SAS DATA step views
  - new multilingual and Asian monolingual [TrueType fonts](#) are added
  - support for [extended attributes](#) on SAS data sets and variables
  - enhanced functionality to extend the [observation count](#) for 32-bit SAS data files
  - enhancements to SAS [data file protection](#)
  - enhanced functionality for VIEWTABLE [column headings](#).

# SAS System Features

---

## Universal Printing

The following features are new in SAS 9.4:

- You can animate multi-page GIF images and SVG files.
- SAS can now create TIFF images, and the EMFPlus and EMFDual metafile formats.
- Transparency is supported for EMF Universal Printers and GIF images that are printed using the PostScript Universal Printer.
- You can add a printer's mark that is not visible in Universal Printing output by using the COLOPHON= system option.
- SVG documents can be magnified by setting the SVGMAGNIFYBUTTON system option. SAS embeds a magnify tool in the document when the SVG document is created.

See “[Creating TIFF Images Using Universal Printing](#)” on page 372.

---

## Fonts

- In [SAS 9.4M6](#) and later, the following monotype TrueType fonts are added to support languages for harmonious web display:
  - NotoSans-Bold
  - NotoSans-BoldItalic
  - NotoSans-Italic
  - NotoSansJP-Bold
  - NotoSansJP-Light
  - NotoSansJP-Regular
  - NotoSansKR-Bold
  - NotoSansKR-Light
  - NotoSansKR-Regular
  - NotoSans-Regular
  - NotoSansSC-Bold
  - NotoSansSC-Light
  - NotoSansSC-Regular
  - NotoSansTC-Bold

- NotoSansTC-Light
- NotoSansTC-Regular
- NotoSansThai-Bold
- NotoSansThai-Regular

**Note:** The NotoSansThai-Regular and the NotoSansThai-Bold fonts do not contain all of the Latin1 glyphs. When a glyph is missing, SAS substitutes the ArialUnicodeMS font, enabling the output to display.

- In SAS 9.4, the following TrueType fonts were added, replacing the fonts shown in the second column:

New Font	Replaces This Font
Arial Unicode MS	Monotype Sans WT
Times New Roman Uni	Thorndale Duospace WJ
CSongGB18030C_Light	Sim Sun
MYingHei_18030_C-Medium	Sim Hei
CSongGB19030-LightHWL	NSimSun
MYingHei_18030C-MediumHWL	Sim Hei

- In SAS 9.4M2, font slanting and emboldening features are new. If you specify italic or bold styles on a universal printer font that does not have italic or bold, the font will display as slanted or bold. See “[Slanting and Emboldening Fonts](#)” on page 321.
- In SAS 9.4M3, the following Avenir Next Fonts were added for the Latin and Cyrillic character sets. These fonts were replaced with AvenirNextforSAS fonts in SAS 9.4M5.

Avenir Next Fonts for the Latin Character Set	Avenir Next Fonts for the Cyrillic Character Set
Avenir Next LT W04 Regular	Avenir Next Cyr W04 Regular
Avenir Next LT W04 Italic	Avenir Next Cyr W04 Italic
Avenir Next LT W04 Demi	Avenir Next Cyr W04 Demi
Avenir Next LT W04 Demi Italic	Avenir Next Cyr W04 Demi Italic
Avenir Next LT W04 Light	Avenir Next Cyr W04 Light
Avenir Next LT W04 Light Italic	Avenir Next Cyr W04 Light Italic

Avenir Next is a sans-serif TrueType font family characterized as a modern typeface designed for on-screen display. Its ancestors are primarily the Futura and Univers typefaces.

- In SAS 9.4M4, the following Helvetica Fonts were added. These fonts were replaced with AvenirNextforSAS fonts in SAS 9.4M5.

#### New Helvetica Fonts

Helvetica LT Pro Regular

Helvetica LT Pro Italic

Helvetica LT Pro Bold

Helvetica LT Pro Bold Italic

Helvetica LT Pro Light

Helvetica LT Pro Light Italic

- In SAS 9.4M5, new AvenirNextforSAS and HelveticaNeueforSAS fonts replace the Avenir Next Fonts and Helvetica LT Pro Fonts. For more information, see [Table 15.14 on page 315](#).

New Font	Replaces This Font
AvenirNextforSAS	Avenir Next LT W04 Regular
AvenirNextforSAS Italic	Avenir Next LT W04 Italic
AvenirNextforSAS Bold	Avenir Next LT W04 Demi
AvenirNextforSAS Bold Italic	Avenir Next LT W04 Demi Italic
AvenirNextforSAS Light	Avenir Next LT W04 Light
AvenirNextforSAS Light Italic	Avenir Next LT W04 Light Italic

The following HelveticaNeueforSAS fonts are new:

New Font	Replaces This Font
HelveticaNeueforSAS	Helvetica LT Pro Regular
HelveticaNeueforSAS Italic	Helvetica LT Pro Italic
HelveticaNeueforSAS Bold	Helvetica LT Pro Bold
HelveticaNeueforSAS Bold Italic	Helvetica LT Pro Bold Italic
HelveticaNeueforSAS Light	Helvetica LT Pro Light

New Font	Replaces This Font
HelveticaNeueforSAS Light Bold	Helvetica LT Pro Light Italic

## Support for SAS DATA Step View Buffers

You can now specify the size of the buffer that is used for DATA step views. Speed up execution time by setting the view buffer so that it can hold more generated observations and require less task switching. See “[VBUFSIZE=](#)” on page 225.

## Extended Attributes

You can create customized attributes for variables and data sets by using extended attributes. Extended attributes are customized metadata for your SAS files. They are user-defined characteristics that you associate with a SAS data set or variable. See “[Extended Attributes](#)” on page 716.

## Extended Observation Count

SAS 9.4 enhances the functionality to extend the observation count by automatically creating a 32-bit SAS data file with an extended observation count and by providing the EXTENDOBSCOUNTER= system option. In SAS 9.4, the EXTENDOBSCOUNTER= data set option, LIBNAME statement option, and system option are, by default, set to YES. See “[Understanding the Observation Count in a SAS Data File](#)” on page 658.

## Cross-Environment Data Access (CEDA)

Under cross-environment data access (CEDA), extended attributes can be read but cannot be updated.

In SAS 9.4M5, CEDA is supported by the SAS Scalable Performance Data Engine (SPD Engine), with some additional restrictions.

See Chapter 34, “[Processing Data Using Cross-Environment Data Access \(CEDA\)](#),” on page 765.

## SMTP Email Authentication Protocol

The SMTP email server authentication protocol is enhanced to look for a user ID that is specified by the EMAILHOST= system option.

## LOCKDOWN State Restrictions

The LOCKDOWN statement and LOCKDOWN system option are new in [SAS 9.4M1](#). With LOCKDOWN, if you are running in a client/server environment, the SAS server administrator can limit access to directories and files. In addition to there being restrictions on directories and files, several language elements are not available when SAS is in a locked-down state.

See “[SAS Processing Restrictions for Servers in a Locked-Down State](#)” on page [18](#).

---

## LOCKDOWN Statement Enhancements

In [SAS 9.4M2](#), the LOCKDOWN statement is enhanced so that certain access methods and their related procedures are disabled by default when a SAS session is locked down.

The LOCKDOWN ENABLE\_AMS= option is also new in [SAS 9.4M2](#). This statement allows administrators to re-enable access methods and procedures that are disabled by default when LOCKDOWN is in effect. See “[SAS Processing Restrictions for Servers in a Locked-Down State](#)” on page [18](#).

---

## Data File Protection

[SAS 9.4](#) supports Advanced Encryption Standard (AES) encryption. AES encryption produces a stronger encryption by using a key value. See “[AES Encryption](#)” on page [796](#).

[SAS 9.4](#) supports the use of metadata-bound libraries for enhanced data security. A metadata-bound library is a physical library that is tied to a corresponding metadata secured table object. See “[Metadata-Bound Libraries](#)” on page [801](#).

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 two algorithms. For more information, see “[Requiring Encryption for Metadata-Bound Data Sets](#)” in [Base SAS Procedures Guide](#).

Beginning in [SAS 9.4M5](#), in the V9 engine, AES2 provides more security than AES. For more information, see [Encryption in SAS](#).

---

## Column Headings in VIEWTABLE

Beginning in [SAS 9.4M1](#), you can save either column labels or column names for the data set that you are viewing in VIEWTABLE.

---

## Restriction for PROC SQL Views

In SAS 9.4M6 and later releases, if you use the V9 engine to create a PROC SQL view that contains a USING clause, the view is not accessible in SAS 9.4M5 or prior releases. For this and similar restrictions, see “[Using SAS Files in a Previous Release](#)” on page 780.

---

## New and Enhanced SAS Language Elements for CAS

In SAS 9.4M5, if you have licensed SAS Viya, then you can access SAS Cloud Analytic Services (CAS) from your SAS 9.4 environment. See “[What is SAS Cloud Analytic Services?](#)” on page 433 for an introduction to CAS.

The following new features and enhancements have been added:

- DATA step processing in CAS. For more information, see [DATA step processing in CAS in SAS Cloud Analytic Services: DATA Step Programming](#).
  - provides multithreaded processing of distributed and non-distributed data
  - provides reading and writing of in-memory tables in CAS
- CAS LIBNAME statement
  - provides DATA step access to CAS tables and to multithreaded processing in CAS. For more information, see [CAS LIBNAME Statement](#).
  - provides access to the new VARCHAR data type, a varying length data type for character strings. For more information, see “[VARCHAR Data Type](#)” in [SAS Cloud Analytic Services: User’s Guide](#).
- Other new SAS language elements, including [SAS language elements for CAS](#) and [CAS-specific language elements](#).



**PART 1**

# SAS System Concepts

<i>Chapter 1</i>	
<b>Essential Concepts of Base SAS Software</b>	3
<i>Chapter 2</i>	
<b>SAS Processing</b>	13
<i>Chapter 3</i>	
<b>Rules for Words and Names in the SAS Language</b>	21
<i>Chapter 4</i>	
<b>SAS Variables</b>	37
<i>Chapter 5</i>	
<b>Missing Values</b>	95
<i>Chapter 6</i>	
<b>Expressions</b>	105
<i>Chapter 7</i>	
<b>Dates, Times, and Intervals</b>	127
<i>Chapter 8</i>	
<b>Error Processing and Debugging</b>	155
<i>Chapter 9</i>	
<b>SAS Output</b>	175
<i>Chapter 10</i>	
<b>By-Group Processing in SAS Programs</b>	195
<i>Chapter 11</i>	
<b>WHERE-Expression Processing</b>	197
<i>Chapter 12</i>	
<b>Optimizing System Performance</b>	217
<i>Chapter 13</i>	
<b>Support for Parallel Processing</b>	231
<i>Chapter 14</i>	
<b>The SAS Registry</b>	245



# Essential Concepts of Base SAS Software

---

<i>What Is SAS?</i>	3
<i>Overview of Base SAS Software</i>	4
Components	4
Other SAS Software	4
Operating System Information	5
<i>Components of the SAS Language</i>	5
SAS Files	5
SAS Data Files	6
External Files	7
Database Management System Files	7
SAS Language Elements	7
SAS Macro Facility	8
<i>Ways to Run Your SAS Session</i>	8
Starting a SAS Session	8
Different Types of SAS Sessions	8
SAS Windowing Environment	8
Interactive Line Mode	9
Noninteractive Mode	9
Batch Mode	10
Object Server Mode	10
<i>Customizing Your SAS Session</i>	11
Setting Default System Option Settings	11
Executing Statements Automatically	11
Customizing the SAS Windowing Environment	11
<i>Conceptual Information about Base SAS Software</i>	12
SAS Concepts	12
DATA Step Concepts	12
SAS Files Concepts	12

---

## What Is SAS?

SAS is a set of solutions for enterprise-wide business users and provides a powerful fourth-generation programming language for performing tasks such as these:

- data entry, retrieval, and management
- report writing and graphics
- statistical and mathematical analysis
- business planning, forecasting, and decision support
- operations research and project management
- quality improvement
- applications development

With Base SAS software as the foundation, you can integrate with SAS many SAS business solutions that enable you to perform large scale business functions.

Examples include data warehousing and data mining, human resources management and decision support, and financial management and decision support.

---

## Overview of Base SAS Software

---

### Components

The core of SAS is Base SAS software, which consists of the following:

**DATA step**

a programming language that you use to manipulate and manage your data.

**SAS procedures**

software tools for data analysis and reporting.

**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, procedure output files, or Hypertext Markup Language (HTML).

**SAS windowing environment**

an interactive, graphical user interface that enables you to easily run and test your SAS programs.

The SAS windowing environment is described in the online Help.

---

## Other SAS Software

This document covers only the concepts related to the core Base SAS language. For other Base SAS software, see these documents:

- [SAS Output Delivery System: User's Guide](#)

- [SAS National Language Support \(NLS\): Reference Guide](#)
- [Base SAS Procedures Guide](#)
- [SAS XMLV2 and XML LIBNAME Engines: User's Guide](#)
- [SAS Macro Language: Reference](#)
- [SAS Logging: Configuration and Programming Reference](#)

---

## Operating System Information

Operating system-specific information about Base SAS can be found in the following documents:

- [SAS Companion for UNIX Environments](#)
- [SAS Companion for Windows](#)
- [SAS Companion for z/OS](#)

---

## Components of the SAS Language

---

### SAS Files

When you work with SAS, you use files that are created and maintained by SAS, as well as files that are created and maintained by your operating environment, and that are not related to SAS. Files with formats or structures known to SAS are referred to as SAS files. All SAS files reside in a SAS library.

The most commonly used SAS file is a SAS data set. A SAS data set is structured in a format that SAS can process. Another common type of SAS file is a SAS catalog. Many different types of information that are used in a SAS job are stored in SAS catalogs. Examples include instructions for reading and printing data values, or function key settings that you use in the SAS windowing environment. A SAS stored program is a type of SAS file that contains compiled code that you create and save for repeated use.

In some operating environments, a SAS library is a physical relationship among files; in others, it is a logical relationship. For more information about the characteristics of SAS libraries, see the SAS documentation for your operating environment:

**Windows Specifics:** "Introduction to SAS Files" in [SAS Companion for Windows](#)

**UNIX Specifics:** "Introduction to SAS Files, Libraries, and Engines"

**z/OS Specifics:** "Introduction to SAS Files" in [SAS Companion for Windows](#)

## SAS Data Files

There are two types of SAS data files:

- SAS data set
- SAS view

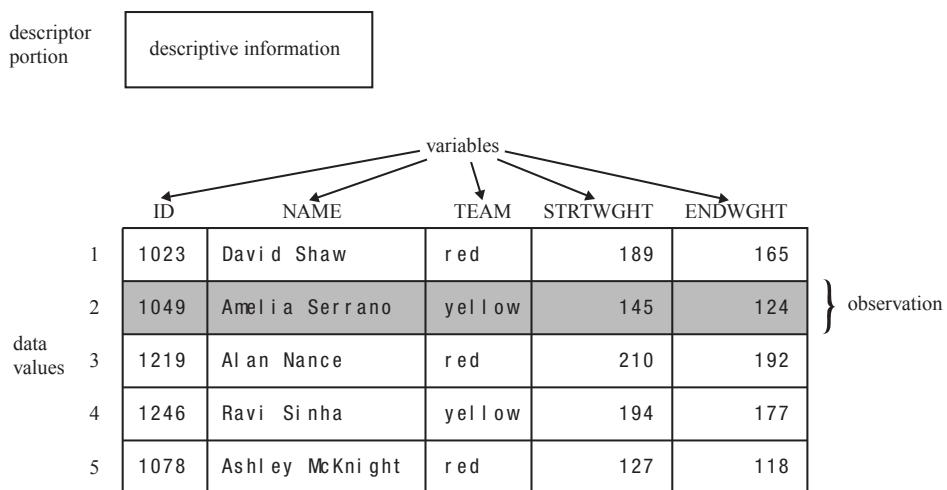
A SAS data set both describes and physically stores your data values. A SAS view, on the other hand, does not actually store values. Instead, it is a query that creates a logical SAS data set that you can use as if it were a single SAS data set. It enables you to look at data stored in one or more SAS data sets or in other vendors' software files. SAS views enable you to create logical SAS data sets without using the storage space required by SAS data files.

A SAS data set consists of the following:

- descriptor information
- data values

The descriptor information describes the contents of the SAS data set to SAS. The data values are data that has been collected or calculated. They are organized into rows, called observations, and columns, called variables. An observation is a collection of data values that usually relate to a single object. A variable is the set of data values that describe a given characteristic. The following figure represents a SAS data set.

**Figure 1.1** Representation of a SAS Data Set



Usually, an observation is the data that is associated with an entity such as an inventory item, a regional sales office, a client, or a patient in a medical clinic. Variables are characteristics of these entities, such as sale price, number in stock, and originating vendor. When data values are incomplete, SAS uses a missing value to represent a missing variable within an observation.

---

## 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 be used for storing

- raw data that you want to read into a SAS data file
- SAS program statements
- procedure output

For more information about the characteristics of external files in your operating environment, see the SAS documentation for your operating environment:

- “Using External Files and Devices” in *SAS Companion for UNIX Environments*,
- “Using External Files under Windows” in *SAS Companion for Windows*, and
- “Assigning External Files” in *SAS Companion for z/OS*.

---

## Database Management System Files

SAS software is able to read and write data to and from other vendors' software, such as many common database management system (DBMS) files. In addition to Base SAS software, you must license the SAS/ACCESS software for your DBMS and operating environment.

---

## SAS Language Elements

The SAS language consists of statements, expressions, options, formats, and functions similar to those of many other programming languages. In SAS, 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 can perform the following tasks:

- read data from external files
- write data to external files
- read SAS data sets and SAS views
- create SAS data sets and SAS views

Once your data is accessible as a SAS data set, you can analyze the data and write reports by using a set of tools known as SAS procedures.

A group of procedure statements is called a PROC step. SAS procedures analyze data in SAS data sets to produce statistics, tables, reports, charts, and plots, to create SQL queries, and to perform other analyses and operations on your data. They also provide ways to manage and print SAS files.

You can also use global SAS statements and options outside of a DATA step or PROC step.

---

## SAS Macro Facility

Base SAS software includes the SAS Macro Facility, 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. Macros are SAS files that contain compiled macro program statements and stored text. 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](#).

---

## Ways to Run Your SAS Session

---

### Starting a SAS Session

You start a SAS session with the SAS command, which follows the rules for other commands in your operating environment. In some operating environments, you include the SAS command in a file of system commands or control statements. In other operating environments, you enter the SAS command at a system prompt or select **SAS** from a menu.

---

### Different Types of SAS Sessions

You can run SAS in any of several ways that might be available for your operating environment:

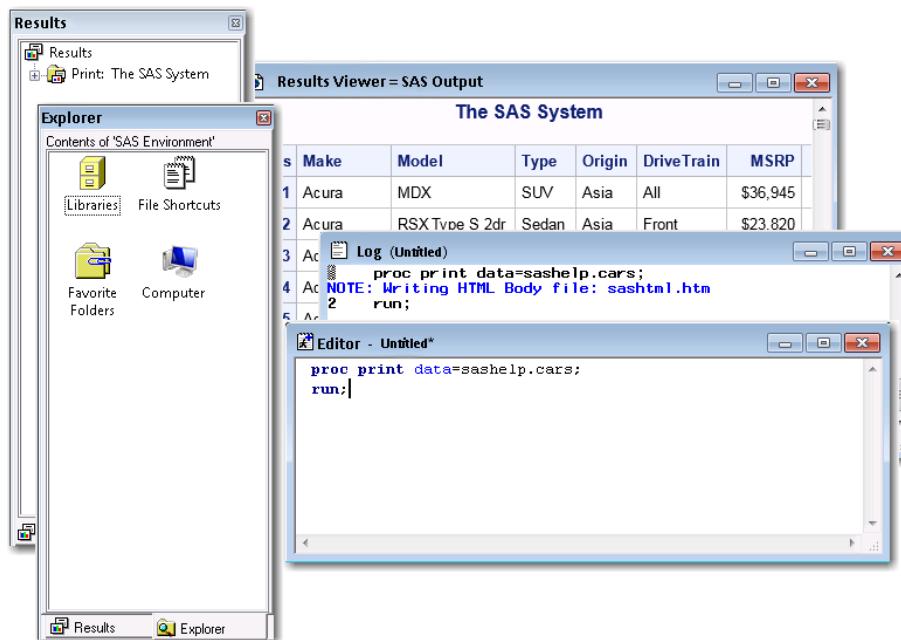
- SAS windowing environment
- interactive line mode
- noninteractive mode
- batch (or background) mode

In addition, SAS/ASSIST software provides a menu-driven system for creating and running your SAS programs.

---

### SAS Windowing Environment

In the SAS windowing environment, you can edit and execute programming statements, display the SAS log, procedure output, and online Help, and more. The following figure shows the SAS windowing environment.

**Figure 1.2** SAS Windowing Environment

In the Explorer window, you can view and manage your SAS files, which are stored in libraries, and create shortcuts to external files. The Results window helps you navigate and manage output from SAS programs that you submit; you can view, save, and manage individual output items. You use the Program Editor, Log, and Output windows to enter, edit, and submit SAS programs, view messages about your SAS session and programs that you submit, and browse output from programs that you submit. For more detailed information about the SAS windowing environment, see [Chapter 16, “Introduction to the SAS Windowing Environment,” on page 385](#).

## Interactive Line Mode

In interactive line mode, you enter program statements in sequence in response to prompts from SAS. DATA and PROC steps execute when one or more of the following happens:

- a RUN, QUIT, or a semicolon on a line by itself after lines of data are entered
- another DATA or PROC statement is entered
- the ENDSAS statement is encountered

By default, the SAS log and output are displayed immediately following the program statements.

## Noninteractive Mode

In noninteractive mode, SAS program statements are stored in an external file. The statements in the file execute immediately after you issue a SAS command referencing the file. Depending on your operating environment and the SAS system

options that you use, the SAS log and output are either written to separate external files or displayed.

For more information about how these files are named and where they are stored, see the SAS documentation for your operating environment:

**UNIX Specifics:** “[The Default Routings for the SAS Log and Procedure Output in UNIX Environments](#)” in *SAS Companion for UNIX Environments*

**Windows Specifics:** “[Routing Procedure Output and the SAS Log to a File](#)” in *SAS Companion for Windows*

**z/OS Specifics:** “[Destinations of SAS Output Files](#)” in *SAS Companion for z/OS*

---

## Batch Mode

You can run SAS jobs in batch mode in operating environments that support batch or background execution. Place your SAS statements in a file and submit them for execution along with the control statements and system commands required at your site.

When you submit a SAS job in batch mode, one file is created to contain the SAS log for the job, and another is created to hold output that is produced in a PROC step or, when directed, output that is produced in a DATA step by a PUT statement.

For more information about executing SAS jobs in batch mode, see the SAS documentation for your operating environment:

**UNIX Specifics:** UNIX operating environment: “[Printing and Routing Output](#)” in *SAS Companion for UNIX Environments*

**Windows Specifics:** Windows operating environment: “[Running SAS in Batch Mode](#)” in *SAS Companion for Windows*

**z/OS Specifics:** z/OS operating environment: “[Directing SAS Log and SAS Procedure Output](#)” in *SAS Companion for z/OS*

Also, see the documentation specific to your site for local requirements for running jobs in batch and for viewing output from batch jobs.

---

## Object Server Mode

When SAS runs in object server mode, SAS runs as an IOM server. Some examples of SAS IOM servers are the SAS Metadata Server, the SAS Workspace Server, the SAS Stored Process Server, and the SAS OLAP Server. For information about running SAS in object server mode, see *SAS Intelligence Platform: Application Server Administration Guide*.

---

# Customizing Your SAS Session

---

## Setting Default System Option Settings

You can use a configuration file to store system options with the settings that you want. When you invoke SAS, these settings are in effect. SAS system options determine how SAS initializes its interfaces with your computer hardware and the operating environment, how it reads and writes data, how output appears, and other global functions.

By placing SAS system options in a configuration file, you can avoid having to specify the options every time you invoke 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.

**Operating Environment Information:** 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.

---

## Executing Statements Automatically

To execute SAS statements automatically each time you invoke 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.

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.

**Operating Environment Information:** 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.

---

## Customizing the SAS Windowing Environment

You can customize many aspects of the SAS windowing environment and store your settings for use in future sessions. With the SAS windowing environment, you can perform the following tasks:

- Change the appearance and sorting order of items in the Explorer window.
- Customize the Explorer window by registering member, entry, and file types.
- Set up favorite folders.
- Customize the toolbar.

- Set fonts, colors, and preferences.

See the SAS online Help for more information and for additional ways to customize your SAS windowing environment.

---

## Conceptual Information about Base SAS Software

### SAS Concepts

SAS concepts include the building blocks of SAS language: rules for words and names, variables, missing values, expressions, dates, times, and intervals, and each of the six SAS language elements — data set options, formats, functions, informats, statements, and system options.

SAS system-wide concepts also include introductory information that helps you begin to use SAS, including information about the SAS log, SAS output, error processing, WHERE processing, and debugging. Information about SAS processing prepares you to write SAS programs. Information about how to optimize system performance as well as how to monitor performance.

---

### DATA Step Concepts

Understanding essential DATA step concepts can help you construct DATA step programs effectively. These concepts include how SAS processes the DATA step, how to read raw data to create a SAS data set, and how to write a report with a DATA step.

More advanced concepts include how to combine and modify information once you have created a SAS data set, how to perform BY-group processing of your data, how to use array processing for more efficient programming, and how to create stored compiled DATA step programs.

---

### SAS Files Concepts

SAS file concepts include advanced topics that are helpful for advanced applications, though not strictly necessary for writing simple SAS programs. These topics include the elements that comprise the physical file structure that SAS uses, including data libraries, data files, SAS views, catalogs, file protection, engines, and external files.

Advanced topics for data files include the audit trail, generation data sets, integrity constraints, indexes, and file compression. In addition, these topics include compatibility issues with earlier releases and how to process files across operating environments.

# SAS Processing

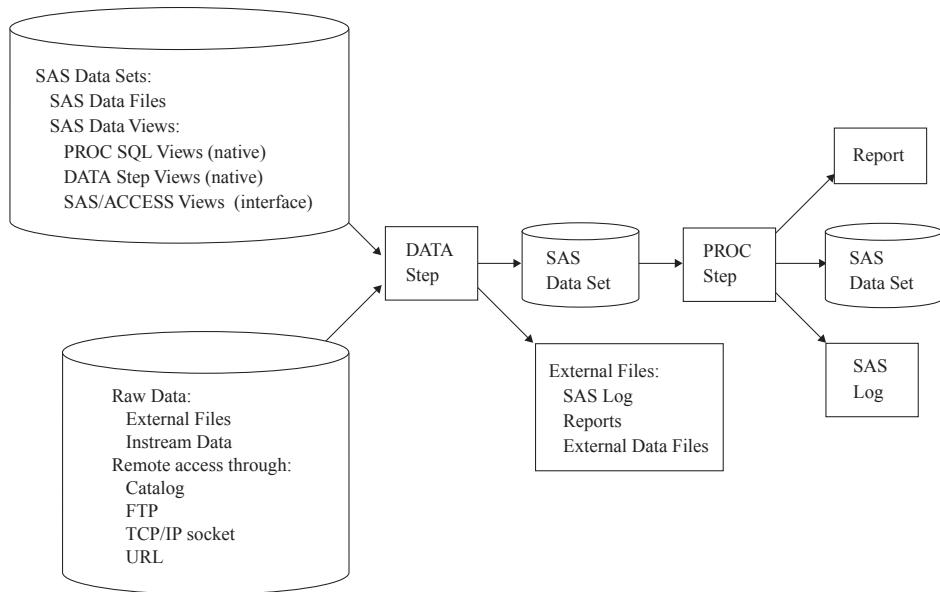
<i>Definition of SAS Processing</i> .....	13
<i>Types of Input to a SAS Program</i> .....	14
<i>The DATA Step</i> .....	16
What Does the DATA Step Do? .....	16
DATA Step Output .....	16
<i>The PROC Step</i> .....	17
What Does the PROC Step Do? .....	17
PROC Step Output .....	17
<i>SAS Processing Restrictions for Servers in a Locked-Down State</i> .....	18
General Information .....	18
z/OS-Specific Information .....	19
Specifying Functions in the Lockdown Path List .....	20

---

## 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 manipulates data, and 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

You can use different types of data as input to a DATA step. The DATA step includes SAS statements that you write, which 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.

## Types of Input to a SAS Program

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

### SAS data sets

can be one of two types:

#### SAS data files

store actual data values. A SAS data file consists of a descriptor portion that describes the data in the file, and a data portion.

#### SAS views

contain references to data stored elsewhere. A SAS view uses descriptor information and data from other files. It enables you to dynamically combine data from various sources, without using storage space to create a new data set. SAS views consist of DATA step views, PROC SQL views, and SAS/ACCESS views. In most cases, you can use a SAS view as if it were a SAS data file.

For more information, see [Chapter 28, “SAS Data Files,” on page 655](#) and [Chapter 29, “SAS Views,” on page 721](#).

### Raw data

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

**External files**

External files can contain raw data, SAS pgm stmts, procedure output, or output created by the PUT stmt. contain records comprised of formatted data (data is arranged in columns) or free-formatted data (data that are not arranged in columns).

**Instream data**

is data included in your program. You use the DATALINES statement at the beginning of your data to identify the instream data.

For more information about raw data, see [Chapter 21, “Reading Raw Data,” on page 471](#).

**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. SAS enables you to access your input data remotely in the following ways:

**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.

**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.

**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.

For more information about accessing data remotely, see the following topics:

- “FILENAME Statement: CLIPBOARD Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: CATALOG Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: DATAURL Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: FTP Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: Hadoop Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: SFTP Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: SOCKET Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: URL Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: WebDAV Access Method” in *SAS Global Statements: Reference*
- “FILENAME Statement: ZIP Access Method” in *SAS Global Statements: Reference*

---

## The DATA Step

---

### What Does the DATA Step Do?

The DATA step processes input data. In a DATA step, you can create a SAS data set, which can be a SAS data file or a SAS view. The DATA step uses input from raw data, remote access, assignment statements, or SAS data sets. 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 20, “DATA Step Processing,” on page 441](#).

---

## 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 headers 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).

---

## 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. Additionally, 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	
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		
PEEKLONG 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:

- “LOCKDOWN system option” and “LOCKDOWN statement” in *SAS Intelligence Platform: Application Server Administration Guide* on the SAS Intelligence Platform Documentation page at [support.sas.com/documentation/onlinedoc/intellplatform](http://support.sas.com/documentation/onlinedoc/intellplatform).
- “Locked-Down Servers” in *SAS Intelligence Platform: Security Administration Guide* on the SAS Intelligence Platform Documentation page at [support.sas.com/documentation/onlinedoc/intellplatform](http://support.sas.com/documentation/onlinedoc/intellplatform).
- 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 installation 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
ZDSLST	ZDSRATT
ZDSNUM	ZDSXATT
ZSIDNM	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 specified in the function has not been added to the lockdown path list, then the function will fail. 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)

# Rules for Words and Names in the SAS Language

---

<b><i>Words in the SAS Language</i></b> .....	<b>21</b>
Definition of Word .....	21
Types of Words or Tokens .....	22
Placement and Spacing of Words in SAS Statements .....	23
<b><i>Names in the SAS Language</i></b> .....	<b>24</b>
Definition of a SAS Name .....	24
Rules for User-Supplied SAS Names .....	24
SAS Name Literals .....	31
Summary of Default Rules for Naming SAS Data Sets and SAS Variables .....	34
Summary of Extended Rules for Naming SAS Data Sets and SAS Variables .....	35

---

## Words in the SAS Language

---

### Definition of Word

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

A word or token ends when SAS encounters one of the following:

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

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

- names
- literals
- numbers
- special characters

## Types of Words or Tokens

There are four basic types of words or tokens:

### 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 token can contain up to 32,767 bytes. In most contexts, however, SAS names are limited to a shorter maximum length, such as 32 or 8 bytes. See [Table 3.1 on page 25](#). 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'
- "Report for the Third Quarter"

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

### number

in general, is composed entirely of numeric digits, with an optional decimal point and a leading plus or minus sign. SAS also 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 number tokens:

- 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 token, although some two-character operators, such as \*\* and <=, form single tokens.

The blank can end a name or a number token, but it is not a token. Here are some examples of special-character tokens:

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

## Placement and Spacing of Words in SAS Statements

### Spacing Requirements

Here are the spacing requirements for words in SAS statements:

- 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. See [Examples on page 23](#).

Although SAS does not have rigid spacing requirements, SAS programs are easier to read and maintain if you consistently indent statements. The examples illustrate useful spacing conventions.

### Examples

- In this statement, blanks are not required because SAS can determine the boundary of every token by examining the beginning of the next token:

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

The first special-character token, the equal sign, marks the end of the name token `total`. The plus sign, another special-character token, marks the end of the name token `x`. The last special-character token, the semicolon, marks the end of the `y` token. Though blanks are not needed to end any tokens in this example, you can add them for readability, as shown here:

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

- This statement requires blank spaces because SAS cannot recognize the individual tokens without them:

```
input group 15 room 20;
```

Without blanks, the entire statement up to the semicolon fits the rules for a name token: it begins with a letter or underscore, contains letters, digits, or underscores thereafter, and is less than 32,767 bytes long. Therefore, this statement requires blanks to distinguish individual name and number tokens.

## Names in the SAS Language

### Definition of a SAS Name

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

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

There are two types of names in SAS:

- SAS language element names (system-supplied names)
- user-supplied names

The following sections will discuss user-supplied SAS names.

## Rules for User-Supplied SAS 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 SAS Variable Names](#)” on page 26 and “[Rules for SAS Data Set Names, View Names, and Item Store Names](#)” on page 28.

- 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 3.1 on page 25](#).
- The first character must be an English letter (A, B, C, . . . , Z) or 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 cannot appear in SAS names.
- 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 filename SASCAT.
- When you create a macro variable, do not use names that begin with SYS.

**Table 3.1** Maximum Length in Bytes of User-Supplied SAS Names

User-Supplied SAS Name	Maximum Length in Bytes
Arrays	32
CALL routines	16
Catalog entries	32
Component objects	32
DATA step statement labels	32
DATA step variable labels	256
DATA step variables	32
DATA step windows	32
Engines	8
Filerefs	8
Formats, character	31
Formats, numeric	32

User-Supplied SAS Name	Maximum Length in Bytes
Functions	16
Generation data sets	28
Informats, character	30
Informats, numeric	31
Librefs	8
Macro variables	32
Macro windows	32
Macros	32
Members of SAS libraries (SAS data sets, SAS views, catalogs, indexes) except for generation data sets	32
Passwords	8
Procedure names (first eight characters must be unique and cannot begin with "SAS")	16
SCL variables	32

## Rules for SAS Variable Names

The rules for SAS variable names have expanded to provide more functionality. The setting of the `VALIDVARNAME=` system option determines what 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=` option has three settings (V7, UPCASE, and ANY), each with varying degrees of flexibility for variable names. If you do not specify the `VALIDVARNAME` option in your SAS session, the default value, V7, is automatically assigned to your SAS session. The following table summarizes the rules for variable names when using the `VALIDVARNAME` system option:

### V7

is the default setting.

Variable name follows these rules:

- The name can be up to 32 bytes in length.
- The name can contain letters of the Latin alphabet, numerals, or underscores.
- The name cannot contain blanks or special characters except for the underscore.
- The name must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore.
- Trailing blanks are ignored. The name alignment is left-justified.

- The name 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, SAS internally converts it to uppercase. You cannot, therefore, use the same variable name with a different combination of upper and lowercase letters to represent different variables. For example, `cat`, `Cat`, and `CAT` all represent the same variable.
- Do not assign variables the names of special SAS automatic variables (such as `_N_` and `_ERROR_`) or variable list names (such as `_NUMERIC_`, `_CHARACTER_`, and `_ALL_`) to variables.

**Examples**

---

```
season='summer';
percent_of_profit=percent;
```

---

### UPCASE

is the same as V7, except that variable names are uppercased, as in earlier versions of SAS.

### ANY

- The name can be up to 32 bytes in length.
- The name can contain any characters, including blanks, national characters, special characters, and multi-byte characters. Names containing these types of characters must be specified as [name literals on page 31](#).
- The name can begin with any characters, including blanks, national characters, special characters, and multi-byte characters.
- The name cannot contain any null bytes.
- Leading blanks are preserved but trailing blanks are ignored.
- The name must contain at least one character. A name with all blanks is not permitted.
- 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, SAS internally converts it to uppercase. You cannot, therefore, 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.

**Requirement** If you use any characters other than the ones that are valid when the `VALIDVARNAME=` system option is set to V7 (letters of the Latin alphabet, numerals, or underscores), then you must express the variable name as a name literal and you must set `VALIDVARNAME=ANY`. If the name includes either the percent sign (%) or the ampersand (&), then you must use single quotation marks in the name literal in order to avoid interaction with the SAS Macro Facility. See “[SAS Name Literals](#)” on page 31 and “[Avoiding Errors When Using Name Literals](#)” on page 34.

**See** [“How Many Characters Can I Use When I Measure SAS Name Lengths in Bytes?” on page 30](#)

---

**Examples** Variable name containing blanks expressed as a name literal:  
`'% of profit'n=percent;`

---

Variable name containing a special character expressed as a name literal:

```
'items@warehouse'n=itemnum;
```

**CAUTION**

**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.

## Rules for SAS Data Set Names, View Names, and Item Store Names

Three types of SAS members, SAS data sets, data views, and item stores, are expanded to have more functionality. The setting of the VALIDMEMNAME= system option determines what rules apply to the names of these members in your SAS session. The [VALIDMEMNAME= option](#) has two settings (COMPATIBLE and EXTEND), each with varying degrees of flexibility for data set names, data view names, and item store names:

### **COMPATIBLE**

specifies that a SAS data set name, a view name, or an item store name must follow these rules:

- The name can be up to 32 bytes in length.
- The name must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore. Subsequent characters can be letters of the Latin alphabet, numerals, or underscores.
- The name cannot contain blanks or special characters except for the underscore.
- The name can contain mixed-case letters. SAS internally converts the member name to uppercase. You cannot, therefore, use the same member name with a different combination of uppercase and lowercase letters to represent different variables. For example, `customer`, `Customer`, and `CUSTOMER` all represent the same member name. How the name on the disk appears is determined by the operating environment.

Alias COMPAT

### **EXTEND**

specifies that a SAS data set name, a SAS view name, or an item store name must follow these rules:

- The name can be up to 32 bytes in length.
- The name can include national characters, but it must be written as a [SAS name literal on page 31](#).
- The name can include special characters, except for the / \ \* ? " < > |: - characters, but it must be written as a [SAS name literal](#).

**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).
- Null bytes are not allowed.
- The name 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.
- The name can contain mixed-case letters. SAS internally converts the member name to uppercase. You cannot, therefore, use the same member name with a different combination of uppercase and lowercase letters to represent different variables. For example, `customer`, `Customer`, and `CUSTOMER` all represent the same member name. How the name appears is determined by the operating environment.

<b>Restrictions</b>	Regardless of the value of VALIDMEMNAME, 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.
---------------------	--

---

The windowing environment supports the extended rules in the Program, Log, and Output windows when VALIDMEMNAME=EXTEND is set. In most SAS windows, these extended rules are not supported. For example, these rules are not supported in SAS Explorer, the VIEWTABLE window, and windows that you open using the Solutions menu.

<b>Note</b>	Hash objects do not support the VALIDMEMNAME=EXTEND system option for data set names. Data set names can not contain special characters or national characters.
-------------	---

<b>Requirement</b>	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 <a href="#">"SAS Name Literals" on page 31</a> .
--------------------	--

<b>Operating environments</b>	For Windows and UNIX operating environments, all Base SAS windows support the extended rules when VALIDMEMNAME=EXTEND is set.
-------------------------------	---

---

For Windows and UNIX operating environments, when you reference a SAS file directly by its physical name, the final embedded period is an extension delimiter. If a physical file reference includes a SAS member name that contains a period, you must add the file extension. For example, if you reference the data set name `my.member` as a physical file, you would add the file extension `sas7bdat` to the reference, as shown in this SET statement: `set './saslib/my.member.sas7bdat'`.

<b>z/OS specifics</b>	The windowing environment for Base SAS supports the extended rules in the Editor, Log, and Output windows when
-----------------------	--

VALIDMEMNAME=EXTEND is set. Other SAS windows, such as the VIEWTABLE window, do not support the extended rules.

When you reference a SAS file directly by its physical name, the final embedded period is considered to be an extension delimiter only if what follows the period is a valid SAS extension. Otherwise, the period is considered to be part of the member name. For example, in the name my.member, member is considered part of the member name and not a file extension. In the name "my.member.sas7bdat", the member name is "my.member" and the file extension is sas7bdat.

---

Tip	The name is displayed in uppercase letters.
See	<a href="#">"How Many Characters Can I Use When I Measure SAS Name Lengths in Bytes?" on page 30</a>
Examples	<pre>data "August Purchases'n; data 'Años de empleo'n;</pre>
CAUTION	<b>Throughout SAS, using the name literal syntax with SAS member names that exceed the 32-byte limit or have excessive embedded quotation marks might cause unexpected results.</b> The intent of the VALIDMEMNAME=EXTEND system option is to enable compatibility with other DBMS member naming conventions, such as allowing embedded blanks and national characters.

---

**Note:** The VALIDMEMNAME= option is not valid for the following tape engines: V9TAPE, V8TAPE, V7TAPE, and V6TAPE.

## How Many Characters Can I Use When I Measure SAS Name Lengths in Bytes?

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 variable name, data set name, view name, or item store name is determined by the number of bytes of storage that are used to store one character. This value is set by the SAS encoding value for your SAS session. VALIDVARNAME=ANY or VALIDMEMNAME=EXTEND must be set to allow the use of national language support (NLS) characters. Otherwise, only one-byte characters are allowed.

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

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

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

- 1 Find the SAS encoding in one of the following ways:
  - Find the ENCODING= system option in the SAS System Options window:
    - 1 Type **options** in the command bar.
    - 2 Right-click **Options** and select **Find Option**.
    - 3 Type **encoding** and click **OK**.
  - In an editor window, specify 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 3.1 on page 25](#). Divide this number by the bytes per character. The result is the maximum number of characters that you can use for the SAS name.

## SAS Name Literals

### Definition of SAS Name Literals

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

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

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

- variable

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 3.2 SAS Name Literal System Option Requirements*

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

The following is an example of a VAR statement and a name literal:

```
var 'a b'n;
```

The following is an example of a VAR statement with variables A and B:

```
var a b;
```

## SAS Name Literal Examples

Here are some examples of SAS name literals:

- libname foo *SAS/ACCESS-engine-name*  
*SAS/ACCESS-engine-connection-options*;
- data foo.'My Table'n;
- data 'Años de empleo'n.;
- data "August Purchases"n;
- input 'Bob''s Asset Number'n;
- input "Bob's Asset Number"n;
- input 'Amount Budgeted'n 'Amount Spent'n  
'iAmount Difference'n;
- 'Statement Label 1'n;

## 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*.  

**Note:** Hash objects do not support the VALIDMEMNAME=EXTEND system option for data set names. Data set names can not contain special characters or national characters.
- 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.  

**Note:** 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

### 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. 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](#)” on page 499 and “[How SAS Identifies the Beginning and End of a BY Group](#)” in *SAS DATA Step Statements: Reference*.

## Avoiding Errors When Using Name Literals

For information about how to avoid creating name literals in error, see “[Avoiding a Common Error with Constants](#)” on page 112.

## Summary of Default Rules for Naming SAS Data Sets and SAS Variables

The table below shows a summary of the rules for naming SAS data sets and SAS variables 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 3.4 on page 35](#).

*Table 3.3 Summary of Default Rules for Naming SAS Data Sets and SAS Variables*

SAS Data Set Names, View Names, and Item Store Names (with VALIDMEMNAME=COMPAT)	Variable Names (with VALIDVARNAME=V7)
<ul style="list-style-type: none"> <li>■ can be up to 32 bytes in length.</li> <li>■ cannot contain special characters (except for the underscore), blanks, or national characters.</li> <li>■ must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore.</li> <li>■ can contain mixed-case letters. SAS internally converts the member name to uppercase. You cannot, therefore, 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.</li> </ul>	<ul style="list-style-type: none"> <li>■ can be up to 32 bytes in length.</li> <li>■ cannot contain special characters (except for the underscore), blanks, or national characters.</li> <li>■ must begin with a letter of the Latin alphabet (A–Z, a–z) or the underscore.</li> <li>■ 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. You cannot, therefore, 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.</li> </ul>

## Summary of Extended Rules for Naming SAS Data Sets and SAS Variables

The following table, shows a summary of the rules for naming SAS data sets (tables) and SAS variables (columns) when the highest level of flexibility is allowed (that is, when the VALIDMEMNAME system option is set to EXTEND and the VALIDVARNAME system option is set to ANY).

**Table 3.4** Summary of Extended Rules for Naming SAS Data Sets and SAS Variables

SAS Data Set Names, View Names, and Item Store Names (with VALIDMEMNAME=EXTEND)	Variable Names (with VALIDVARNAME=ANY)
<ul style="list-style-type: none"> <li>■ can be up to 32 bytes in length.</li> <li>■ can contain special characters except for / \ * ? " &lt; &gt;   : -. A name that contains special characters must be specified as a <a href="#">name literal</a>.</li> <li>■ cannot begin with a blank or a period.</li> <li>■ ignores leading and trailing blanks.</li> <li>■ can contain mixed-case letters. SAS internally converts the member name to uppercase. You cannot, therefore, 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.<sup>1</sup></li> </ul>	<ul style="list-style-type: none"> <li>■ can be up to 32 bytes in length.</li> <li>■ can contain special characters including / \ * ? " &lt; &gt;   : -. A name that contains special characters must be specified as a <a href="#">name literal</a>.</li> <li>■ can begin with any character, including blanks, national characters, special characters, and multi-byte characters.</li> <li>■ preserves leading blanks, but trailing blanks are ignored.</li> <li>■ 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. You cannot, therefore, 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.</li> <li>■ cannot contain all blanks.</li> </ul>

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



# SAS Variables

---

<b>Definition of SAS Variables .....</b>	<b>38</b>
<b>SAS Variable Attributes .....</b>	<b>38</b>
<b>Ways to Create Variables .....</b>	<b>42</b>
Overview .....	42
Create a New Variable Using the LENGTH Statement .....	43
Create a New Variable Using the ATTRIB Statement .....	43
Create a New Variable Using an Assignment Statement .....	45
Reading Data with the INPUT Statement in a DATA Step .....	46
Create a New Variable Using the FORMAT or INFORMAT Statements .....	48
Using the IN= Data Set Option .....	50
<b>Variable Type Conversions .....</b>	<b>51</b>
<b>Aligning Variable Values in SAS Output .....</b>	<b>52</b>
<b>Reordering Variables in SAS Output .....</b>	<b>53</b>
<b>Automatic Variables .....</b>	<b>54</b>
<b>SAS Variable Lists .....</b>	<b>55</b>
Definition .....	55
Numbered Range Lists .....	56
Name Range Lists .....	58
Name Prefix Lists .....	62
Special SAS Name Lists .....	62
<b>The OF Operator with Functions and Variable Lists .....</b>	<b>63</b>
Definition .....	63
<b>Dropping, Keeping, and Renaming Variables .....</b>	<b>64</b>
Using Statements or Data Set Options .....	64
Using the Input or Output Data Set .....	65
Order of Application .....	66
Example: Convert a Variable from Character to Numeric .....	66
Example: Drop Variables from the Output Data Set .....	67
Example: Drop and Rename Variables .....	67
<b>Encrypting Variable Values .....</b>	<b>67</b>
Customized Encryption and Decryption Algorithms for SAS Variables .....	67
Example 1: 1-Byte-to-1-Byte Swap Using the TRANSLATE Function .....	68
Example 2: Using a 1-Byte-to-2-Byte Swap with the TRANWRD Function .....	69
Example 3: Using Different Functions to Encrypt Numeric Values as Character Strings .....	70
<b>Numerical Accuracy in SAS Software .....</b>	<b>72</b>
Overview .....	72

Truncation in Binary Numbers .....	73
How SAS Stores Numeric Values .....	74
Floating-Point Representation Using the IEEE Standard .....	78
Floating-Point Representation on Windows .....	79
Floating-Point Representation on IBM Mainframes .....	83
Troubleshooting Errors in Precision .....	85
Transferring Data between Operating Systems .....	94

---

## Definition of SAS Variables

**variables**

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

**character variables**

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

**numeric variables**

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

**numerical precision**

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

---

## SAS Variable Attributes

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

**Table 4.1** Variable Attributes

Variable Attribute	Possible Values	Default Value
Name	Any valid SAS name	None
Type *	Numeric and Character	Numeric
Length *	Numeric: 2 to 8 bytes ** Character: 1 to 32,767 bytes	8 bytes
		8 bytes

Variable Attribute	Possible Values	Default Value
Format	Numeric formats	BESTw. format
	Character formats	\$w. format
	Date and Time formats	
	See <a href="#">Formats By Category</a> for a complete list of SAS formats.	
Informat	Numeric informats	w.d Informat
	Character informats	\$w. Informat
	Date and Time informats	
	See <a href="#">Informats By Category</a> for a complete list of SAS informats.	
Label	Up to 256 characters.	None
Position in observation	1 - n	None
Index type	Simple	None
	Composite	
	None	
	Both	
Extended attribute	Numeric or character	None

\* If not explicitly defined, a variable's type and length are automatically set by SAS based on the variable's first occurrence in a DATA step.

\*\* The minimum length is 2 bytes in some operating environments, 3 bytes in others. See the [documentation for your operating system](#).

**Note:** The maximum number of variables can be greater than 32,767. This number depends on your environment, the file's attributes and the total length of all the variables, which cannot exceed the [maximum page size](#), which is 32,767.

To get information about a variable's attributes, use the [CONTENTS statement](#) in the [DATASETS procedure](#) or the functions that are named in the following definitions:

#### name

identifies a variable. A variable name must conform to SAS naming rules. See a list of rules in [Table 3.3 on page 34](#)

The names `_N_`, `_ERROR_`, `_FILE_`, `_INFILE_`, `_MSG_`, `_IORC_`, and `_CMD_` are reserved for [automatic variables](#), which are generated automatically during DATA step execution. Note that SAS products use variable names that start and end with an underscore; it is recommended that you do not use names that start and end with an underscore in your own applications.

To determine the value of this attribute, use the [VNAME function](#) or the [VARNAME function](#).

**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.

In an INPUT statement, you can assign a length other than the default length to character variables. You can also assign a length to a variable in the ATTRIB statement.

If you create a variable for the first time in an assignment statement and do not explicitly define its type, then SAS determines its type based on the variable's first occurrence in the DATA step. The variable gets the same type and length as the expression on the right side of the assignment statement.

- A variable that appears for the first time on the left side of an assignment statement has the same length as the expression on the right side of the assignment statement.
- A character variable that appears for the first time in a DATA step in an INPUT statement and whose length has not been otherwise specified, has a default length of 8.
- A character variable that appears for the first time in an FORMAT or INFORMAT statement has a type and length based on the category of format or informat that is applied when it is created. See “[Formats by Category](#)” in [SAS Formats and Informats: Reference](#) and “[Informats by Category](#)” in [SAS Formats and Informats: Reference](#) for a list of these categories.

To determine the value of this attribute, use the [VARTYPE function](#) or the [VTYPE function](#).

**length**

refers to the number of bytes used to store each of the variable's values in a SAS data set. You can use a [LENGTH statement](#) to set the length of both numeric and character variables. Variable lengths that are specified in a LENGTH statement affect the length of numeric variables only in the output data set. During processing, all numeric variables have a length of 8. Lengths of character variables that are specified in a LENGTH statement affect both the length during processing and the length in the output data set.

In an INPUT statement, you can assign a length other than the default length to character variables. You can also assign a length to a variable in the ATTRIB statement.

If you create a variable and do not explicitly define its length, then the length is automatically set by SAS. SAS sets the length based on the variable's first occurrence in the DATA step.

- A variable that appears for the first time on the left side of an assignment statement has the same length as the expression on the right side of the assignment statement.
- A character variable that appears for the first time in a DATA step in an INPUT statement and whose length has not been otherwise specified, has a default length of 8.
- A character variable that appears for the first time in an FORMAT or INFORMAT statement has a type and length based on the category of format or informat that is applied when it is created. See “[Formats by Category](#)” in [SAS Formats and Informats: Reference](#) and “[Informats by Category](#)” in [SAS Formats and Informats: Reference](#) for a list of these categories.

[SAS Formats and Informats: Reference](#) and “[Informats by Category](#)” in [SAS Formats and Informats: Reference](#) for a list of these categories.

See [Table 4.3 on page 45](#) for more information about variable lengths.

To determine the value of this attribute, use the [VARLEN function](#) or the [VLENGTH function](#).

#### format

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

See “[Create a New Variable Using the FORMAT or INFORMAT Statements](#)” on [page 48](#) for an example and information about how SAS sets the length of variables declared in the FORMAT and INFORMAT statements.

To determine the value of this attribute, use the [VARFMT function](#) or the [VFORMAT function](#).

#### informat

refers to the instructions that SAS uses when reading data values. If no informat is specified, the default informat is w.d for a numeric variable, and \$w. for a character variable. You can assign SAS informats to a variable in the INFORMAT or ATTRIB statement. You can use the FORMAT procedure to create your own informat for a variable.

See “[Create a New Variable Using the FORMAT or INFORMAT Statements](#)” on [page 48](#) for an example and information about how SAS sets the length of variables declared in the FORMAT and INFORMAT statements.

To determine the value of this attribute, use the [VARLEN function](#) or the [VFORMAT function](#).

#### label

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

To determine the value of this attribute, use the [VARLABEL function](#) or the [VLABEL function](#).

#### position in observation

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

```
var rent-phone;
```

See “[SAS Variable Lists](#)” on [page 55](#) for more information.

The positions of variables in a SAS data set affect the order in which they appear in the output of SAS procedures. There is an exception if you control the order within your program (for example, with a VAR statement).

To determine the value of this attribute, use the [VARNUM function](#).

#### index type

indicates whether the variable is part of an index for the data set. See “[Understanding SAS Indexes](#)” on [page 692](#) for more information.

To determine the value of this attribute, use the [OUT= option](#) in the [CONTENTS statement](#) of the [DATASETS procedure](#) to create an output data set. The `IdxUsage` variable in the output data set contains one of the following values for each variable:

*Table 4.2 Index Type Attribute Values*

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

#### extended attribute

is a user-defined attribute that is created using the [XATTR ADD VAR statement](#) in the [DATASETS procedure](#). For more information, see [XATTR ADD statement](#).

---

## Ways to Create Variables

---

### Overview

The recommended way to create variables is to use one of the following methods. When using any of these methods, be sure to reference the variable for the first time in the statement that is used to create it.

- [Create a New Variable Using the LENGTH Statement.](#)
- [Create a New Variable Using the ATTRIB Statement.](#)
- [Create a New Variable Using an Assignment Statement.](#)

Here are some additional ways that you can create variables in a DATA step. However, these methods are usually recommended for changing existing variables or for reading in existing variables that are located in external files or raw data:

- [Specify a new variable in a FORMAT or INFORMAT statement.](#)
- [Read data with the INPUT statement in a DATA step.](#)

**Note:** This list is not exhaustive. For example, the SET, MERGE, MODIFY, and UPDATE statements can also create variables.

## Create a New Variable Using the LENGTH Statement

You can use the LENGTH statement to create a new variable and explicitly set its length.

**Important:** Place the LENGTH statement first in the DATA step before any other statements that reference the variable. The maximum length of any character variable in SAS is 32,767 bytes.

```
data Sales;
length Salesperson $25 Price 6; /* 1 */
Salesperson='Forrest, Peter';
Price=49.99;
/* Reset length of numeric variable */
Length Price 8; /* 2 */
run;
```

- 1 You cannot change the length of a *character variable* with a subsequent LENGTH or ATTRIB statement within the same DATA step. You should use the longest possible value in the first statement that references the character variable.
- 2 You can change the length of a *numeric 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. Consider the following DATA step, in which variables are assigned a length of 200 bytes, then concatenated:

```
data sales;
length address1 address2 address3 $ 200;
address3 = address1||address2;
run;
```

Because the length of Address3 is 200 bytes, only the first 200 bytes of the concatenation (the value of Address1) are assigned to Address3. You might be able to avoid this problem by using the [TRIM function](#) to remove trailing blanks from Address1 before performing the concatenation, as follows:

```
data sales;
length address1 address2 address3 $ 200;
address3 = trim(address1)||address2;
run;
```

To change the lengths of existing numeric variables, use the LENGTH statement or the ATTRIB statement. See [Example](#) and “[ATTRIB Statement](#)” in [SAS DATA Step Statements: Reference](#) for more information.

## Create a New Variable Using the ATTRIB Statement

If the variable does not already exist, you can use the ATTRIB statement with one or more of the following options to create a new variable:

- FORMAT=

- INFORMAT=
- LENGTH=

In this example, the ATTRIB statement is specified first in the DATA step. The ATTRIB statement uses the FORMAT= option to create a character variable named Flavor with the \$w. Format and a length of 10 bytes. The ATTRIB statement also specifies the LENGTH= option to create a variable named Sizes with a length of 20 bytes.

**Example Code 4.1 Create a New Variable Using the ATTRIB Statement**

```
data lollipops;
    attrib Flavor format=$10.
                           sizes length=$20;
    Flavor="Cherry";
    Size="Small Medium Large";
run;
proc contents data=lollipops; run;
```

**Alphabetic List of Variables and Attributes**

#	Variable	Type	Len	Format
1	Flavor	Char	10	\$10.
2	Sizes	Char	20	

**Note:**

- You cannot change the length of a *character variable* with a subsequent LENGTH or ATTRIB statement within the same DATA step
- You can change the length of a *numeric variable* by using a subsequent LENGTH statement.

If the assignment statement is specified first in the DATA step (before the ATTRIB statement), then the DATA step would have created the character variable Flavor with a length of 6 and the character variable Sizes with a length of 18.

**Example Code 4.2 Change the Attributes of an Existing Variable Using the ATTRIB Statement**

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

**Alphabetic List of Variables and Attributes**

#	Variable	Type	Len	Format
1	Flavor	Char	6	\$10.
2	Sizes	Char	18	

If the variable already exists, then you can use the ATTRIB statement to specify one or more of the following variable attributes to change the existing variable:

- FORMAT=
- INFORMAT=
- LENGTH=

■ LABEL=

**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.

For more information, see “[ATTRIB Statement](#)” in [SAS DATA Step Statements: Reference](#).

## Create a New Variable Using an Assignment Statement

In SAS, you do not have to declare a variable before assigning a value to it. The variable is automatically declared the first time you specify it in an assignment statement.

- SAS assigns the variable's type and length based on its first occurrence in the DATA step.
- SAS assigns the variable the same type and length as the expression on the right side of the assignment operator.
- If a variable appears for the first time on the right side of an assignment operator, then SAS assumes that it is a numeric variable, that its value is missing, and assigns it a length of 8 bytes. If no later statement gives it a value, SAS prints a note in the log that the variable is not initialized.

**Table 4.3 Resulting Variable Types and Lengths Produced When They Are Not Explicitly Set**

Expression	Example	Resulting Type of X	Resulting Length of X	Explanation
Numeric variable	data test; length a 4; x=a; run;	Numeric	8 bytes	Default numeric length (8 bytes unless otherwise specified)
Character variable	data test; length a \$ 4; x=a; run;	Character	4 bytes	Length of source variable
Character literal	data test; x='ABC'; x='ABCDE'; run;	Character	3 bytes	Length of first literal encountered
Concatenated variables	data test; length a \$4 b \$6 c \$2; x=a  b  c; run;	Character	12 bytes	Sum of the lengths of all variables
Concatenated variables and literal	data test; length a \$ 4; x=a  'CAT'; x=a  'CATNIP'; run;	Character	7 bytes	Sum of the lengths of variables and literals encountered in first assignment statement

If a variable appears for the first time on the right side of an assignment statement, SAS assumes that it is a numeric variable and that its value is missing. If no later statement gives it a value, SAS prints a note in the log that the variable is not initialized.

You can use the “[VARINITCHK= System Option](#)” in [SAS System Options: Reference](#) to specify that no notes, warnings, or error messages are written to the SAS log. If an error is set, the DATA step stops processing.

**Note:** A [RETAIN statement](#) initializes a variable and can assign it an initial value, even if the RETAIN statement appears after the assignment statement.

## Reading Data with the INPUT Statement in a DATA Step

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

- list input ([INPUT statement: List, simple or modified](#))
- formatted input ([INPUT statement: Formatted](#))
- column input ([INPUT statement: Column](#))
- named input ([INPUT statement: Named](#))

### Reading Data with the INPUT Statement in a DATA Step

The following example uses simple list input to create three new variables in a SAS output data set named `Gems`. The INPUT statement defines the variables `Name`, `Carats`, and `Color`. The variables `Name` and `Color` have a dollar sign (\$) following their names in the INPUT statement. This declares the variables as character variables. No length is specified for any of the variables, so SAS automatically assigns all variables a default length of 8 bytes.

```
data gems;
    input Name $ Carats Color $;
    datalines;
        emerald    1 green
        aquamarine 2 blue
    ;
    proc contents data=gems; run;
    proc print data=gems; run;

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

## See Also

- “Reading Unaligned Data with Simple List Input” in *SAS DATA Step Statements: Reference*
- “When to Use List Input” in *SAS DATA Step Statements: Reference*

## Reading Data Using Formatted Input (With Length Specified)

Formatted input uses special instructions called informats in the INPUT statement to determine how values should be read. Informats can include information about variable length.

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. See [Informats by Category](#) for a list of these categories.

- 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.

The example below uses formatted input to create a SAS data set named `Gems`. The INPUT statement defines the variables `Name` and `Color` as character variables by specifying the character informat, `$CHARw.`, in the INPUT statement. The program defines the `Carats` variable by specifying the numeric informat, `COMMAw.d`, in the INPUT statement.

```
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;
```

**Figure 4.1** Results for Creating New Variables Using Simple List Input with Lengths Specified

Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
2	Carats	Num	8
3	Color	Char	8
1	Name	Char	10

Obs	Name	Carats	Color
1	emerald	1.5	green
2	aquamarine	2.0	blue

To change the lengths of existing numeric variables, use the LENGTH statement or the ATTRIB statement. See “[“Example: ”” in SAS DATA Step Statements: Reference](#)” and “[ATTRIB Statement](#)” in *SAS DATA Step Statements: Reference* for more information.

## See Also

- [“INPUT Statement: List” in SAS DATA Step Statements: Reference](#)
- [“INPUT Statement: Formatted” in SAS DATA Step Statements: Reference](#)

## Create a New Variable Using the FORMAT or INFORMAT Statements

You can use the FORMAT or INFORMAT statement to create a new variable and simultaneously associate a format or informat with the new variable. To create a new variable using either the FORMAT or INFORMAT statement, make sure that you place the FORMAT or INFORMAT statement first in the DATA step.

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’s or informat’s category. See [Formats by Category](#) and [Informats by Category](#) for a list of these categories.

- 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.

In the following example, 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 (`COMMAw.d`), 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 4.3 Specifying 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 4.1 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 4.4 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 4.2 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 below, the variables `Flavor` and `Amount` are created using an assignment statement rather than using a `FORMAT` statement. 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.

**Example Code 4.5** *Changing the Format of an Existing Variable Using the FORMAT Statement*

```
data lollipops;
  Flavor='Cherry';
  Amount=10;
  format Flavor $uppercase10. Amount comma10.;
run;
proc contents data=lollipops; run;
```

**Output 4.3** *PROC CONTENTS Output for Changing the Format of an Existing Variable Using the FORMAT Statement*

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

Since the expression on the right is the 6-letter character string, `cherry`, SAS assigns a length of 6 bytes to the character variable `Flavor`. SAS assigns a length of 8 bytes to the numeric variable, `Amount`.

## See Also

- [Table 4.1 on page 38](#)
- [SAS Formats and Informats: Reference](#)

## Using the IN= Data Set Option

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

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

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

# Variable Type Conversions

If you define a numeric variable and assign the result of a character expression to it, SAS tries to convert the character result to a numeric value and execute the statement. If the conversion is not possible, SAS prints a note to the log, assigns the numeric variable a value of missing, and sets the automatic variable \_ERROR\_ to 1. For a listing of the rules by which SAS automatically converts character variables to numeric variables and vice versa, see “[Automatic Numeric-Character Conversion](#)” on page 113.

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

**Example Code 4.1 Automatic Variable Type Conversions (partial SAS log)**

```

44   data _null_;
45     x= 3626885;
46     length y $ 4;
47     y=x;
48     put y;
49   run;
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
      47:6
36E5

50   data _null_;
51     x1= 3626885;
52     length y1 $ 1;
53     y1=x1;
54     xs=0.000005;
55     length ys $ 1;
56     ys=xs;
57     put y1= ys=;
58   run;
NOTE: Numeric values have been converted to character
      values at the places given by: (Line):(Column).
      53:7    56:7
NOTE: Invalid character data, x1=3626885.00 , at line 53 column 7.
y1= * ys=0
x1=3626885 y1= * xs=5E-6 ys=0 _ERROR_=1 _N_=1
NOTE: At least one W.D format was too small for the number to be printed. The
      decimal may be shifted by the "BEST" format.

59   proc printto; run;

```

In the first DATA step of the example, SAS is able to fit the value of Y into a 4-byte field by representing its value in scientific notation. In the second DATA step, SAS cannot fit the value of Y1 into a 1-byte field and displays an asterisk (\*) instead.

## Aligning Variable Values in SAS Output

In SAS output, numeric variables are right-aligned and character values are left-aligned. You can further control their alignment by using a format.

However, in SAS LISTING output, when you use an assignment statement to assign a character value to a variable, SAS stores the value as it appears in the statement and does not perform an alignment. [Output 4.4 on page 52](#) illustrates the character value alignment produced by the following program:

```
ods listing;
options nodate;
data aircode;
  input city $ 1-13 WAC 15-17;
  length airport $ 30;

  if city='San Francisco' then airport='SFO';
  else if city='Paris' then airport='CDG';
  else if city='New York' then airport='JFK';
  else if city='Moscow' then airport='MOW';
  else if city='Melbourne' then airport='      MEB';

  datalines;
  San Francisco 67
  Paris          427
  New York       67
  Moscow         770
  Melbourne      802
;
proc print data=aircode;
run;
ods listing close;
```

This example produces the following LISTING output:

[Output 4.4 Character Variable Alignment in SAS Listing Output](#)

Obs	city	WAC	airport
1	San Francisco	67	SFO
2	Paris	427	CDG
3	New York	67	JFK
4	Moscow	770	MOW
5	Melbourne	802	MEB

In HTML output, when you assign a character value in an assignment statement, SAS ignores the white space and left-aligns the characters as usual. The same example above produces the following output in HTML:

**Output 4.5 Character Variable Alignment in SAS HTML Output**

Obs	city	WAC	airport
1	San Francisco	67	SFO
2	Paris	427	CDG
3	New York	67	JFK
4	Moscow	770	MOW
5	Melbourne	802	MEB

---

## Reordering Variables in SAS Output

You can control the order in which variables are displayed in SAS output by using the following declarative statements:

- ARRAY
- ATTRIB
- FORMAT
- INFORMAT
- LENGTH
- RETAIN

For any of these statements to work, they must be placed prior to any one of the following declarative statements:

- SET
- MERGE
- UPDATE

Only the variables whose positions are relevant need to be listed. Variables not listed in these statements retain their original position.

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. Specifying the LENGTH statement before the SET statement causes the variable Height to be moved to the first position in the output data set.

**Example Code 4.6 Using the LENGTH Statement to Reorder Variables**

```
data Class1;
length Height 3; /* The LENGTH statement precedes the SET statement */
set Sashelp.Class; /* and causes the variable Height to be placed first */
```

```

run;                                /* in the output */
proc print data=Class1;
run;

```

**Output 4.6 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

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. In the following example, the RETAIN statement causes the variable Weight to be listed first in the output data set:

**Example Code 4.7 Using the RETAIN Statement to Reorder Variables**

```

data Class2;
retain Weight;      /* The RETAIN statement precedes the SET statement */
set Sashelp.Class;  /* and causes the variable Weight to be placed first */
run;                /* in the output */
proc print data=Class2;
run;

```

**Output 4.7 Using the RETAIN Statement to Reorder Variables**

Obs	Weight	Name	Sex	Age	Height
1	112.5	Alfred	M	14	69.0
2	84.0	Alice	F	13	56.5
3	98.0	Barbara	F	13	65.3
4	102.5	Carol	F	14	62.8

## Automatic Variables

Automatic variables are created automatically by the DATA step or by DATA step statements. These variables are added to the program data vector but are not written to 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.

Automatic variables that are created by specific statements are documented with those statements. For examples, see the “[BY Statement](#)” in [SAS DATA Step](#)

*Statements: Reference*, the “**MODIFY Statement**” in *SAS DATA Step Statements: Reference*, and “**WINDOW Statement**” in *SAS DATA Step Statements: Reference*.

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

#### `_N_`

`_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.

#### `_ERROR_`

`_ERROR_` is 0 by default but is set to 1 whenever an error is encountered, such as an input data error, a conversion error, or a math error, as in division by 0 or a floating point overflow. You can use the value of this variable to help locate errors in data records and to print an error message to the SAS log.

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

```
if _error_=1 then put _infile_;
if _error_ then put _infile_;
```

## SAS Variable Lists

### Definition

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

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

With the exception of the *numbered range list*, you refer to the variables in a variable list in the same order that SAS uses to keep track of the variables. SAS keeps track of active variables in the order in which the compiler encounters them within a DATA step. This happens whether the active variables are read from existing data sets, an external file, or created in the step.

In a *numbered range list*, you can refer to variables that were created in any order, provided that their names have the same prefix.

**Note:** Only the numbered range list is used in the RENAME= option.

## Numbered Range Lists

### Definition

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

```
Var1 Var2 Var3 Var4 Var5 Var6
```

```
Var1-Var6
```

### Example

For example, suppose you decide to give some of your variables sequential names, as in Score1, Score2, Score3, and so on. You can write an INPUT statement as follows:

```
data exam;
    input Score1-Score10;
datalines;
1 2 3 4 5 6 7 8 9 10
;
```

Score1	Score2	Score3	Score4	Score5	Score6	Score7	Score8	Score9	Score10
1	2	3	4	5	6	7	8	9	10

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

```
data exam;
    input Score11-Score20;
datalines;
1 2 3 4 5 6 7 8 9 10
;
proc print data=exam noobs; run;
```

Score11	Score12	Score13	Score14	Score15	Score16	Score17	Score18	Score19	Score20
1	2	3	4	5	6	7	8	9	10

### Example

Using the same data set from the previous example, the following example shows how you can use a numbered range list to reference a subset of the variables:

```
data exam2;
set exam;
drop Score13-Score18;
run;
proc print data=exam2 noobs; run;
```

Score11	Score12	Score19	Score20
1	2	9	10

## Example

You can also use a numbered range list in an ARRAY statement. In the following example, notice how the variables are first defined in the INPUT statement before they are used in the array declaration. The variables in the INPUT statement can either be written out individually, as shown in the first DATA step below, or they can be written as a numbered range list, as shown in the second DATA step:

```
data temperatures;
    input day1 day2 day3 day4 day5 day6 day7;
datalines;
44.4 44.6 44.9 45.2 45.4 45.7 45.9
;
proc print data=temperatures;
    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;
```

Average Daily Low Temperature

Obs	day1	day2	day3	day4	day5	day6	day7
1	44.4	44.6	44.9	45.2	45.4	45.7	45.9

Average Daily Low Temperature in Celsius

Obs	day1	day2	day3	day4	day5	day6	day7
1	6.88889	7	7.16667	7.33333	7.44444	7.61111	7.72222

## Restriction for Numbered Range Lists

You cannot specify numbered range lists for data sets or variables that end in numbers larger than 2147483647. For example, when the following DATA step program executes, an error will occur because the last variable defined (a2147483648=3) is greater than the limit for variable names in a numbered range lists. The number range list specified in the KEEP statement extends beyond 2147483647:

```
data test;
a2147483646=1;
a2147483647=2;
a2147483648=3;
keep a2147483646=1-a2147483648=3; /* 1 */
```

```
run;
```

One solution to this limitation is to rename the variables that make up the list by appending a character to the ends of the variable names:

```
proc sql noprint; /* 2 */
  select cats(name,'=',name,'A')
    into :list
    separated by ' '
    from dictionary.columns
    where libname = 'WORK' and memname = 'TEST';
quit;

proc datasets library = work nolist; /* 3 */
  modify TEST;
  rename &list;
run; quit;

data test;
  set test;
  keep a2147483646A--a2147483648A; /* 3 */
run;
proc print data=test; run;
```

- 1 Create a data set that contains variable names longer than the maximum allowed for lists. Specify the variables as a numbered range list in the KEEP statement. An error is returned.
- 2 Use PROC SQL and SAS dictionary tables to concatenate the letter A to the ends of all the variable names. Read those values into a macro variable named list.
- 3 Use PROC DATASETS MODIFY to rename the variables by specifying the macro variable as the argument to the RENAME statement.
- 4 Use the variables in a *name* range list in the KEEP statement (instead of in a *numbered* range list). See “[Name Range Lists](#)” on page 58 for more information.

**Note:** Notice the difference in syntax between a numbered range list and the name range list used in the solution above. In a name range list, the range is indicated using double hyphens.

## Name Range Lists

Name range lists rely on the order of variable definition, as shown in the following table:

**Table 4.4 Name Range Lists**

Variable List	Included Variables
<i>x</i> - - <i>a</i>	all variables in order of variable definition, from variable <i>x</i> to variable <i>a</i> inclusive

Variable List	Included Variables
x -NUMERIC- a	all numeric variables from variable x to variable a inclusive
x -CHARACTER- a	all character variables from variable x to variable a inclusive

**Note:** 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.

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.

Below are some examples that show how name range lists can be used with various SAS statements and options.

The following DATA step creates the data set that will be used in Examples 1 through 3.

```
data patients;
  input Idnum Name $ Weight Pulse BMI Gender $;
  datalines;
  123 Jones 155 82 27 F
  456 Smith 175 78 24 M
  789 Kamda 172 69 22 F
;
```

Obs	Idnum	Name	Weight	Pulse	BMI	Gender
1	123	Jones	155	82	27	F
2	456	Smith	175	78	24	M
3	789	Kamda	172	69	22	F

### Example 1

In the following example, the name range list specified in the KEEP statement keeps all variables between and including Name and Pulse.

```
data patientsConsec;
  set patients;
  keep Name--Pulse;
run;
proc print data=patientsConsec; run;
```

Obs	Name	Weight	Pulse
1	Jones	155	82
2	Smith	175	78
3	Kamda	172	69

### Example 2

In the following example, the name range list specified in the KEEP statement keeps all numeric variables between and including Idnum and BMI.

```
data patientsNumerics;
  set patients;
  keep Idnum-numeric-BMI;
run;
```

```
proc print data=patientsNumerics; run;
```

Obs	Idnum	Weight	Pulse	BMI
1	123	155	82	27
2	456	175	78	24
3	789	172	69	22

### Example 3

In the following example, the name range list specified in the KEEP statement keeps all character variables between and including Idnum and Pulse.

```
data patientsCharacters;
  set patients;
  keep Idnum-character-Pulse;
run;
proc print data=patientsCharacters; run;
```

Obs	Name
1	Jones
2	Smith
3	Kamda

### Example 4

The following example uses the Sashelp.Fish data set.

This example shows how you can use a name range list to specify the variables in an array. The ARRAY statement reads all variables between and including the variables Length1 and Width into an array named fish. The DO loop iterates through the items in the array and converts the values to centimeters.

```
/* Array is used to convert inches to centimeters */
data fishConvert(drop=i);
  set sashelp.fish(where=(species="Whitefish"));
  array fish{5} Length1--Width;
  do i=1 to 5;
    fish{i}=fish{i} * 2.54;
  end;
run;
proc print data=fishConvert; run;
```

Obs	Species	Weight	Length1	Length2	Length3	Height	Width
1	Whitefish	270	59.944	66.040	72.898	21.2662	10.7889
2	Whitefish	270	61.214	67.310	74.422	20.6893	10.7912
3	Whitefish	306	65.024	71.120	78.232	22.2961	11.8913
4	Whitefish	540	72.390	78.740	86.360	27.2898	16.8675
5	Whitefish	800	85.598	92.456	100.584	29.8734	16.8969
6	Whitefish	1000	94.742	101.600	110.490	31.3792	16.8735

### Example 5

The following example uses the Sashelp.Baseball data set, in which the following variables are defined:

#### Output 4.8 Sashelp.Baseball Data Set Variables

Name	Team	League	nAtBat	nHits	nHome	nRuns	nRBI	nBB	YrMajor	CrAtBat	CrHits	CrHome
CrRuns	CrRbi	CrBB	Division	Position	nOuts	nAssts	nError	Salary	Div	logSalary		

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 character variables between and including Name and logSalary into an array named stats.

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.

```
/* Array is used to multiply stats by 10 */
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	68	92
5	Perez, Tony	2000	510	20	140	29	25
6	Simmons, Ted	1270	320	40	140	25	12

### Example 6

The following example uses the Sashelp.Baseball data set. See [Output 4.8 on page 60](#) for a list of variables defined in the data set.

In the example, the name range list specified in the ARRAY statement reads all character variables between and including Name and logSalary into an array named case. The KEEP statement specifies a named range list to keep all variables between nAtBat and CrBB.

```
/* Array is used to uppercase character values */
data baseballUpcase;
  set sashelp.baseball(where=(YrMajor>18));
  array case{6} Name-character-Div;
  do i=1 to 6;
    case{i}=upcase(case{i});
  end;
  keep crRuns-numeric-nOuts Name-character-Div;
run;
proc print data=baseballUpcase; run;
```

Obs	Name	Team	CrRuns	CrRbi	CrBB	League	Division	Position	nOuts	Div
1	BAKER, DUSTY	OAKLAND	984	1013	762	AMERICAN	WEST	OF	90	AW
2	NETTLES, GRAIG	SAN DIEGO	1172	1287	1057	NATIONAL	WEST	3B	83	NW
3	ROSE, PETE	CINCINNATI	2185	1314	1566	NATIONAL	WEST	1B	523	NW
4	JACKSON, REGGIE	CALIFORNIA	1509	1659	1342	AMERICAN	WEST	DH	0	AW
5	PEREZ, TONY	CINCINNATI	1272	1652	925	NATIONAL	WEST	1B	398	NW
6	SIMMONS, TED	ATLANTA	1048	1348	819	NATIONAL	WEST	UT	167	NW

### Example 7

The following example uses the Sashelp.Baseball data set. See [Output 4.8 on page 60](#) for a list of variables defined in the data set.

In the example, the name range list specified in the `VAR` statement prints all variables between `nAtBat` and `CrBB`.

```
proc print data=sashelp.baseball(obs=5);
  var Name nAtBat--nHome Salary;
  run;
```

Obs	Name	nAtBat	nHits	nHome	Salary
1	Allenson, Andy	293	66	1	-
2	Ashby, Alan	315	81	7	475.0
3	Davis, Alan	479	130	18	480.0
4	Dawson, Andre	496	141	20	500.0
5	Galarraga, Andres	321	87	10	91.5

Note: You can use the `VARNUM` option in `PROC CONTENTS` or the `VAR` statement in `PROC PRINT` to print the variables in the order of definition.

For more information about using arrays with variable lists, see “[Using Variable Lists to Define an Array Quickly](#)” on page 611.

## Name Prefix Lists

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

```
sum(of Sales:)
```

This character string tells SAS to calculate the sum of all the variables that begin with “Sales,” such as `Sales_Jan`, `Sales_Feb`, and `Sales_Mar`.

## Special SAS Name Lists

Special SAS name lists include:

`_NUMERIC_`

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.

# 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 4.5** 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 – xn)	Performs the function on variable values between x1 and xn inclusive. <sup>1</sup>
Arrays	Function(OF array-name(*))	Performs the function on the named array. <sup>2</sup>
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.

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

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

```

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 difference between x1 and x3 (inclusive).

## Dropping, Keeping, and Renaming Variables

### Using Statements or Data Set Options

The DROP, KEEP, and RENAME statements or the DROP=, KEEP=, and RENAME= data set options control which variables are processed or output during the DATA step. You can use one or a combination of these statements and data set options to achieve the results 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 4.6** 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
effect all output data sets	effect 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 4.7 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

Where Specified	Data Set Option or Statement	Purpose	Status of Variable or Variable Name
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=	specifies which variables are written to individual output data sets	all variables are available for processing
	KEEP=	changes name of variables in individual output data sets	use old name in program statements and other output data set options

## Order of Application

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

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

## Example: Convert a Variable from Character to Numeric

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

This example uses the DROP= and RENAME= data set options and the INPUT function to convert the variable `poprank` from a character to a numeric type. The name `poprank` is changed to different name, `tempvar`, before processing. This enables a new variable with the same name (`poprank`) to be created in the next assignment statement, in which it is created as a numeric type. Note that the variable `tempvar` is dropped from the output data set. The `tempvar` variable is needed only temporarily in the DATA step.

```

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

```

## Example: Drop Variables from the Output Data Set

This example uses the DROP statement and the DROP= data set option to control the output of variables to two new SAS data sets. The DROP statement applies to both data sets, Corn and Bean. You must use the RENAME= data set option to rename the output variables BeanWt and CornWt in each data set.

```

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

```

## Example: Drop and Rename Variables

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

```

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

```

# Encrypting Variable Values

## Customized Encryption and Decryption Algorithms for SAS Variables

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.

This section provides sample programs that use the DATA step with different functions and methods to encrypt and decrypt variables.

## Example 1: 1-Byte-to-1-Byte Swap Using the TRANSLATE Function

The first sample program shows how to use a simple 1-byte-to-1-byte swap with the TRANSLATE function. Because this method is not complicated, you might assume that it is not as secure. However, because you are designing the pattern of characters, special characters, or values for the TRANSLATE arguments `from` and `to`, you are creating your own unique encryption algorithm.

The values that are listed for both the TRANSLATE `from` and the TRANSLATE `to` arguments do not have to be in any sequential, alphabetical, or numerical order.

In the following sample code, 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.

The first DO loop creates the encrypted value and the second DO loop creates the decrypted value by reversing the order and returning the original value.

### *Example Code 4.8 A Simple 1-Byte-to-1-Byte Swap Using the TRANSLATE Function*

```
data sample1;
input @1 name $;
length encrypt decrypt $ 8;

/*ENCRYPT*/
do i = 1 to 8;
  encrypt=strip(encrypt) ||translate(substr(name,i,1),
  '0123456789!@#$%^&*()-=,./?<', 'ABCDEFGHIJKLMNPQRSTUVWXYZ') ;
end;

/*DECRYPT*/
do j = 1 to 8;
  decrypt=strip(decrypt) ||translate(substr(encrypt,j,1),
  'ABCDEFGHIJKLMNPQRSTUVWXYZ', '0123456789!@#$%^&*()-=,./?<') ;
end;

drop i j;
datalines;
ROBERT
JOHN
GREG
;
proc print;
run;
```

The following output shows the results of the PROC PRINT for Example 1:

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

---

## Example 2: Using a 1-Byte-to-2-Byte Swap with the TRANWRD Function

This sample program shows how to encrypt values using a 1-byte-to-2-byte swap with the TRANWRD function. Each 1-byte character is replaced with a 2-digit number. To change values that go from 1-byte to many bytes, or many bytes to 1 byte, you need to use the TRANWRD function and you must assign the resulting variable the same name as the variable that is being translated. The values that are listed for the `from` values do not have to be in any special order.

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.

The values that are being changed are the letters A, B, C, D, E, and F, but they are listed in a random order as the `from_1` value. The `to_1` value is assigned a starting value of 21. However, this can be any other 2-digit number, such as 11 or 38, as long as the last value that is assigned does not go over 99, which then turns into a 3-digit number.

The first DO loop creates the encrypted value and 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.

**Example Code 4.9 Using a 1-Byte-to-2-Byte Swap with the TRANWRD Function**

```

data sample2;
input @1 id $12.;

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

```

```

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
BADC
ABC
;
proc print;
run;

```

The following output shows the results of the PROC PRINT for Example 2:

The SAS System			
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	BADC	2624252122	BADC
7	ABC	242621	ABC

---

## Example 3: Using Different Functions to Encrypt Numeric Values as Character Strings

This sample program shows you 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.

The program uses two DATA steps: one to encrypt the values and the other to reverse the process and decrypt the values. You can merge the encrypt and decrypt DATA steps into a single DATA step, if needed.

The first DATA step reads numeric values that are 5 digits or fewer. The numeric variable is converted to a character variable and is split into five separate values.

Four ARRAY statements are used: the first array sets up the `from` values; the second sets up the `to` values; the third holds the five separate numeric values; and the fourth holds the five new, separate encrypted values.

The `from` and `to` arrays are each created with three elements. The `from` ARRAY is assigned the same string of numbers for all three elements, and the `to` ARRAY is assigned a different string of letters for each of the three elements to build the every-third-time rotating pattern.

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

The first DO loop uses the SUBSTR function to split the value into five separate values and assigns each to the old ARRAY. The second DO loop translates each value by using the INDEXC function to find the original number in the `from` ARRAY and, if found, translates the value using the `from` ARRAY, and rotates through the list of elements every third time. The encrypted value is created by using the CATS function to concatenate the five translated values.

If you compare the two DATA steps in the example below, 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.

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 to convert the final values to numeric values.

**Example Code 4.10 Using Different Functions to Encrypt Numeric Values into Character Strings**

```

data sample3;
  input num;
  array from(3) $ 10 from1-from3 ('0123456789','0123456789','0123456789');
  array to(3) $ 10 to1-to3 ('ABCDEFGHIJ','JKLMNOPQRST','UVWXYZABCD');
  array old(5) $ old1-old5;
  array new(5) $ new1-new5;
  char_num=put(num,5.);
  do i = 1 to 5;
    old(i)=substr(char_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;
  encrypt_num=cats(of new1-new5);
  keep num encrypt_num;
  datalines;
12345
70707
99
1111
;
run;

```

```

data sample3;
  set sample3;
  array to(3) $ 10 to1-to3 ('0123456789','0123456789','0123456789');
  array from(3) $ 10 from1-from3 ('ABCDEFGHIJ','JKLMNOPQRST','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;

```

The following output shows the results of the PROC PRINT for Example 3:

### The SAS System

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

---

## Numerical Accuracy in SAS Software

---

### 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 must be approximated. 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.

**Note:** Calculated statistics can vary slightly depending on the order in which observations are processed. Such variations are due to numerical errors that are introduced by floating-point arithmetic, the results of which should be considered approximate and inexact. The order of observation processing can be affected by non-deterministic effects of multi-threaded or parallel processing. The order of processing can also be affected by inconsistent or non-deterministic ordering of observations that are produced by a data source, such as a DBMS that delivers query results through an ACCESS engine.

## 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 will be rounded when stored on a computer.

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

For example, in decimal arithmetic, the expression  $(3 \times 0.1)$  is expected to be equal to 0.3, so the difference between  $(3 \times 0.1)$  and  $(0.3)$ , must be 0. Because the decimal values 0.1 and 0.3 do not have exact binary representations, this equality does not hold true in binary arithmetic. If you compute the difference between the two values in a SAS program, the result is not 0, as [Example Code 4.11 on page 74](#) illustrates.

In the example, SAS sets the variables `point_three` and `three_times_point_three` to 0.3 and  $(3 \times 0.1)$ , respectively. It then compares

the two values by subtracting one from the other and writing the result to the SAS log:

**Example Code 4.11 Comparing Imprecise Values in SAS**

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

**Output 4.9 Log Output for Comparing Imprecise Values in SAS**

```
1  data a;
2    point_three=0.3;
3    three_times_point_one=3 * 0.1;
4    difference=point_three - three_times_point_one;
5    put 'The difference is ' difference;
6    run;

The difference is -5.55112E-17
NOTE: The data set WORK.A has 1 observations and 3 variables.
NOTE: DATA statement used (Total process time):
      real time          0.99 seconds
      cpu time           0.12 seconds
```

The log output shows that  $(3 \times 0.1) - 0.3$  does not equal 0, as it does in decimal arithmetic. This is because the variable "difference" is the result of calculations that are performed on rounded values, or, infinitely repeating binary values.

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 4.8** Largest Integer That Can Be Safely Stored in a Given Length

When Variable Length Equals ...	Largest Integer z/OS	Largest Integer Window and 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 value of your numeric variables will always be integers between -8192 and 8192, then you can safely specify a length of 3 to store the number:

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

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

## Floating-Point Representation

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

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

$$987 = \underbrace{.987}_{\text{mantissa}} \times \underbrace{10^3}_{\text{base}}$$

- 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.

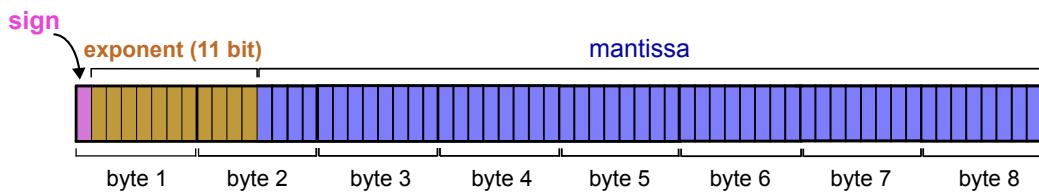
One major difference between scientific notation and floating-point representation is that in scientific notation, the base is 10. In floating-point representation, on most operating systems, the base is either 2 or 16 depending on the system.

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.

$$987 = 0 \underbrace{100\ 0100}_{\text{exponent}} \underbrace{0111\ 0110}_{\text{mantissa}}$$

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. In this way, 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.

The byte layout for a double-precision binary floating-point number 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, then the number is positive.

**Figure 4.2** 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 v. 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 4.9** IBM and IEEE Standard for Floating-Point Formats

Specifications	IBM Mainframe	IEEE Standard (Windows and 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

$$\begin{aligned}3000 &= (B \times 16^2) + (B \times 16^1) + (8 \times 16^0) \\&= (11 \times 256) + (11 \times 16) + (8 \times 1)\end{aligned}$$

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

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

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](#)” on page 91.

- **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 M M M M M
Byte 1	Byte 2	Byte 3	Byte 4

M M M M M M M M Byte 5	M M M M M M M M Byte 6	M M M M M M M M Byte 7	M M M M M M M M 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.

## Conversion Example

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; that is, the mantissa is the sum of a series of fractions such as 1 half , 1 fourth , 1 eighth , 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 <sup>7</sup>	2 <sup>6</sup>	2 <sup>5</sup>	2 <sup>4</sup>	2 <sup>3</sup>	2 <sup>2</sup>	2 <sup>1</sup>	2 <sup>0</sup>	.2 <sup>-1</sup>	2 <sup>-2</sup>	
	128	64	32	16	8	4	2	1	1/2	1/4	
255.75 =	1 × 2 <sup>7</sup>	1 × 2 <sup>6</sup>	1 × 2 <sup>5</sup>	1 × 2 <sup>4</sup>	1 × 2 <sup>3</sup>	1 × 2 <sup>2</sup>	1 × 2 <sup>1</sup>	1 × 2 <sup>0</sup>	1 × 2 <sup>-1</sup>	1 × 2 <sup>-2</sup>	

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

↑  
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 in such a way 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

$$\begin{aligned} 1030 &= (4 \times 16^2) + (0 \times 16^1) + (6 \times 16^0) \\ &= 1024 + 0 + 6 \end{aligned}$$

The converted hexadecimal value for 1030 will be 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) to get this result:

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.

## Accuracy on x64 Windows Processors

Consider this example:

```
data _null_;
x=.50000000000000000000000000000000;
y=.50000000000000000000000000000000;
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=3FDFFFFFFF
```

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

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

### Log Output

```
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). The routine is written in x64 assembly to maximize performance, and any changes to the routine for Windows x64 could not only lead to poorer performance, but they could also potentially introduce other unintended side effects or errors in other computations.

## 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 Byte 1	M M M M M M M M Byte 2	M M M M M M M M Byte 3	M M M M M M M M Byte 4
M M M M M M M M Byte 5	M M M M M M M M Byte 6	M M M M M M M M Byte 7	M M M M M M M M 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.

### Conversion Example

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:

$$\text{true exponent} + \text{bias} = 3 + 40 = 43 \text{ (hexadecimal)} = \text{stored exponent}$$

The final portion to be determined is the sign of the mantissa. By convention, the sign bit for positive mantissas is 0, and the sign for negative mantissas is 1. This information is stored in the first bit of the first byte. From the hexadecimal value in Step 4, compute the decimal equivalent and write it in binary format. Add the sign bit to the first position. The stored value now looks like this:

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

$$11000003 = 195 \text{ in decimal} = C3 \text{ in hexadecimal}$$

- 6 The final step is to put it all together:

432001999999999 - floating point representation for 512.1

C32001999999999 - 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'.

This example shows how 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 (like the above value has repeating '9s'), there is a good chance that the value cannot be represented exactly.

## 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 ROUND Function to Avoid Computational Errors

Errors that are caused by the accumulation of performing calculations on imprecise values can be resolved by rounding. The following example shows how you can use the ROUND function to round the results or make decisions for each iteration.

**Example Code 4.12 Using the ROUND Function to Avoid Computational Errors**

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

```
run;
```

**Example Code 4.2 Log Output for Using the ROUND Function to Avoid Computational Errors**

```
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
```

Here is another example of a numerical precision issue that occurs on z/OS but not on the PC.

**Example Code 4.13 Using the ROUND Function with the IF Statement**

```
data a;
    input gender $ height;
    datalines;
        m 60
        m 58
        m 59
        m 70
        m 60
        m 58
    ;
proc freq;
    tables gender/out=new;
    run;

data final;
    set new;
    if percent=100 then put 'equal';
    else put 'not equal';
run;
```

**Output 4.10** Output for Using the ROUND Function with the IF Statement

The FREQ Procedure				
gender	Frequency	Percent	Cumulative Frequency	Cumulative Percent
m	6	100.00	6	100.00

In the example, PROC FREQ creates an output data set that contains the variable Percent. Because all of the values for the variable Gender are the same, you might expect Percent to have an exact value of 100. However, when the value of Percent is tested, the log indicates that Percent is not exactly 100.

The algorithm used by PROC FREQ to produce the variable Percent involves mathematical computations. The result is very close to 100 but not exactly. Using the ROUND function (or the COMPFUZZ function) in the IF statement resolves this issue.

A work-around for very simple calculations (for example, retaining only 2 digits to the right of the decimal point) is to multiply the values by 100 and use the ROUND function to round them to integers. Once you have performed the calculations on the new whole numbers, divide by 100 to convert the values back to decimal form.

In the following example, the values for variable x are stored in the SAS data set as real numbers. The number is multiplied by 1,000 and the ROUND function is used to change the values to integers. The SUM statement is used to sum all the values of New. On the last observation, which is detected using the END= option, the sum is divided by 1,000 to convert the values back to fractions.

**Example Code 4.14** Summing Rounded Values

```
data a;
  set b end=last;
  new=round(x*1000);
  sum+new;
  if last then sum=sum/1000;
run;
```

See “[ROUND Function](#)” in [SAS Functions and CALL Routines: Reference](#) for more information about this function.

## Numeric Comparison Considerations

When comparing non-integer values that do not have precise decimal or hexadecimal floating-point representations you can sometimes encounter surprising results. For example, 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.’

**Example Code 4.15** Comparing Values That Have Imprecise Representations

```
data a;
  x=15.7-11.9;
  if x=3.8 then put 'equal';
  else put 'not equal';
```

```
run;
```

**Example Code 4.3 Log Output for Comparing Values That Have Imprecise Representations**

```
988      data a;
989      x=15.7-11.9;
990      if x=3.8 then put 'equal';
991      else put 'not equal';
992      run;

not equal
NOTE: The data set WORK.A has 1 observations and 1 variables.
```

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 binary and hexadecimal.

If you add the PRINT procedure to display the results, you can see that the PROC PRINT output is different from the stored value. 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.

```
proc print data=a;
run;
```

**Output 4.11 Output for Comparing Values**

**The SAS System**

Obs	x
1	3.8

## Using Formats to Confirm Precision Errors

In the next example, two different formats are applied to the results given in [Example Code 4.15 on page 87](#) and displayed in the SAS log. The first format, 10.8 displays the value for `x` as 3.8; The second format, 18.16 displays `x` as 3.777 ... (repeating).

**Example Code 4.16 Using Formats to Confirm Precision Errors**

```
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;
```

**Example Code 4.4** Log Output: Using Formats to Confirm Precision Errors

```

102  data a;
103    x=15.7-11.9;
104    if x=3.8 then put 'equal';
105    else put 'not equal';
106  put x=10.8;
107  put x=18.16;
run;

not equal
x=3.80000000
x=3.7999999999999900
NOTE: The data set WORK.A has 1 observations and 1 variables.

```

Another way to verify the stored value of `x` is to apply the HEX16. format to the calculated result. The HEX16. format is a special format that can be used to show floating-point representation.

**Example Code 4.17** Using the HEX16 Format to Verify Calculated Results

```

data a;
x=15.7-11.9;
if x=3.8 then put 'equal';
else put 'not equal';
put x=hex16.;
run;

```

**Example Code 4.5** Using the HEX16 Format to Verify Calculated Results

```

123  data a;
124    x=15.7-11.9;
125    if x=3.8 then put 'equal';
126    else put 'not equal';
127  put x=hex16.;
128 run;

not equal
x=400E666666666664
NOTE: The data set WORK.A has 1 observations and 1 variables.

```

See “[HEXw. Format](#)” in [SAS Formats and Informats: Reference](#) for more information about this format. See “[Dictionary of Formats](#)” in [SAS Formats and Informats: Reference](#) for more information about formats in general.

## Using the ROUND Function to Avoid Comparison Errors

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.

**Example Code 4.18** Using the ROUND Function to Avoid Comparison Errors

```

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

```

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

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

**Example Code 4.6 Log Output: Using the ROUND Function to Avoid Comparison Errors**

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

MATCH
```

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 “[ROUND Function](#)” in *SAS Functions and CALL Routines: Reference* for more information about this function.

## 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 4.10** Table Representation of the Numbers 256 to 272 in 8 Bytes

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

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

```

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

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

```

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

You receive no warning that the value of 257 is truncated in the first DATA step. Note, however, that Y1 has the value 258 because the values of X are kept in full, 8-byte floating-point representation in the program data vector. The value is 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.

See “[LENGTH Statement](#)” in *SAS DATA Step Statements: Reference* for more information about this statement.

## 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.

## Determining How Many Bytes Are Needed to Store a Number Accurately

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:

**Example Code 4.19 Determining How Many Bytes Are Needed to Store a Number Accurately**

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

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

proc print noobs;
  var value minlen;
run;
```

The following output shows the results from this example:

**Output 4.12 Determining How Many Bytes Are Needed to Store a Number Accurately****The SAS System**

<b>value</b>	<b>minlen</b>
269	3
270	3
271	3
272	2

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

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 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. This means 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. [Table 4.9 on page 77](#) 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](#).

# Missing Values

---

<b>Definition of Missing Values</b>	<b>95</b>
<b>Creating Special Missing Values</b>	<b>96</b>
Definition .....	96
Tips .....	96
Example .....	96
<b>Order of Missing Values</b>	<b>97</b>
Numeric Variables .....	97
Character Variables .....	98
<b>When Variable Values Are Automatically Set to Missing by SAS</b>	<b>98</b>
When Reading Raw Data .....	98
When Reading a SAS Data Set .....	99
<b>When Missing Values Are Generated by SAS</b>	<b>100</b>
Propagation of Missing Values in Calculations .....	100
Invalid Operations .....	100
Invalid Character-to-Numeric Conversions .....	100
Creating Special Missing Values .....	100
Preventing Propagation of Missing Values .....	101
<b>Working with Missing Values</b>	<b>102</b>
How to Represent Missing Values in Raw Data .....	102
How to Set Variable Values to Missing in a DATA Step .....	102
How to Check for Missing Values in a DATA Step .....	103

---

## Definition of Missing Values

### 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

By default, SAS prints a missing numeric value as a single period (.) and a missing character value as a blank space. See “[Creating Special Missing Values](#)” on page 96 for more information about special numeric missing values.

# Creating Special Missing Values

## Definition

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

- 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. For further information, see the “[MISSING Statement](#)” in *SAS Global Statements: Reference*.

## Example

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 5.1 Output with Multiple Missing Values**

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

X indicates TESTER ABSENT  
 I indicates TEST WAS INCOMPLETE

## 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 5.1** 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
	+n	positive numbers
largest		

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 usually 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

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
;
```

This DATA step produces a SAS data set with the following variable values:

OBS	X	Y
1	4	.
2	1	2
3	3	.
4	1	2

When X equals 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.

## 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

### Propagation of Missing Values in Calculations

SAS assigns missing values to prevent problems from arising. 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.

### 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 nonnumerical 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.

### Creating Special Missing Values

The result of any numeric missing value in a SAS expression is a period. Thus, both special missing values and ordinary numeric missing values propagate as a period.

```
data a;
  x=.d;
  y=x+1;
  put y=;
```

```
run;
```

This DATA step results in the following log:

**Example Code 5.1 SAS Log Results for a Missing Value**

```
130  data a;
131      x=.d;
132      y=x+1;
133      put y=;
134      run;
y=.
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 132:10
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
```

## Preventing Propagation of Missing Values

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. 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;
```

**Example Code 5.2 SAS Log Results for a Missing Value in a Statistic Function**

```
143  data test;
144  x=.;
145  y=5;
146  a=x+y;
147  b=sum(x,y);
148  c=5;
149  c+x;
150  put a= b= c=;
151  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 146:6
NOTE: The data set WORK.TEST has 1 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time          0.12 seconds
      cpu time          0.01 seconds
```

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.

**Note:** The SUM statement also ignores missing values, so the value of C is also 5.

For more information about functions, see “[SAS Functions and CALL Routines by Category](#)” in *SAS Functions and CALL Routines: Reference*.

## Working with Missing Values

### 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 5.2 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)

### How to Set Variable Values to Missing in a DATA Step

You can set values to missing within your DATA step by using program statements such as this one:

```
if age<0 then
age=.;
```

This statement sets the stored value of Age to a numeric missing value if Age has a value less than 0.

**Note:** You can display a missing numeric value with a character other than a period by using the DATA step's MISSING statement or the MISSING= system option.

The following example sets the stored value of Name to a missing character value if Name has a value of “none”:

```
if name="none" then name='';
```

Alternatively, if you want to set to a missing value for one or more variable values, you can use the CALL MISSING routine. For example:

```
call missing(sales, name);
```

This sets both variable values to a missing value.

**Note:** You can mix character and numeric variables in the CALL MISSING routine argument list.

---

## How to Check for Missing Values in a DATA Step

You can use the N and NMISS functions to return the number of nonmissing and missing values, respectively, from a list of numeric arguments.

When you check for ordinary missing numeric values, you can use code that is similar to the following:

```
if numvar=. then do;
```

If your data contains special missing values, you can check for either an ordinary or special missing value with a statement that is similar to the following:

```
if numvar<=.z then do;
```

To check for a missing character value, you can use a statement that is similar to the following:

```
if charvar=' ' then do;
```

The MISSING function enables you to check for either a character or numeric missing value, as in:

```
if missing(var) then do;
```

In each case, SAS checks whether the value of the variable in the current observation satisfies the condition specified. If it does, SAS executes the DO group.

**Note:** Missing values have a value of `false` when you use them with logical operators such as AND or OR.



# Expressions

---

<i>Definitions for SAS Expressions</i>	106
<i>Examples of SAS Expressions</i>	106
<i>SAS Constants in Expressions</i>	107
Definition	107
Character Constants	107
Using Quotation Marks with Character Constants	107
Comparing Character Constants and Character Variables	108
Character Constants Expressed in Hexadecimal Notation	108
Numeric Constants	108
Numeric Constants Expressed in Standard Notation	109
Numeric Constants Expressed in Scientific Notation	109
Numeric Constants Expressed in Hexadecimal Notation	109
Date, Time, and Datetime Constants	110
Bit Testing Constants	110
Avoiding a Common Error with Constants	112
<i>SAS Variables in Expressions</i>	113
Definition	113
Automatic Numeric-Character Conversion	113
<i>SAS Functions in Expressions</i>	114
<i>SAS Operators in Expressions</i>	114
Definitions	114
Arithmetic Operators	115
Comparison Operators	115
The IN Operator	116
Numeric Comparisons	117
The IN Operator in Numeric Comparisons	117
Character Comparisons	118
The IN Operator in Character Comparisons	119
Logical (Boolean) Operators and Expressions	120
The AND Operator	121
The OR Operator	121
The NOT Operator	121
Boolean Numeric Expressions	122
The MIN and MAX Operators	122
The Concatenation Operator	123
Order of Evaluation in Compound Expressions	124

---

## Definitions for SAS Expressions

**expression**

is generally a sequence of operands and operators that form a set of instructions that are performed to produce a resulting value. You use expressions in SAS program statements to create variables, assign values, calculate new values, transform variables, and perform conditional processing. SAS expressions can resolve to numeric values, character values, or Boolean values.

**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 expressions**

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. For syntax and further information about WHERE expressions, see [Chapter 11, “WHERE-Expression Processing,” on page 197](#).

---

## Examples of SAS Expressions

The following are examples of SAS expressions:

- 3
- x
- x+1
- age<100
- trim(last) || ', ' || first

---

# SAS Constants in Expressions

---

## Definition

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. They can also be used as values for certain options. Constants are also called literals.

The following are types of SAS constants:

- character
- numeric
- date, time, and datetime
- bit testing

---

## Character Constants

A character constant consists of 1 to 32,767 characters and must be enclosed in quotation marks. Character constants can also be represented in hexadecimal form.

---

## Using Quotation Marks with Character Constants

In the following SAS statement, `Tom` is a character constant:

```
if name='Tom' then do;
```

If a character constant includes a single quotation mark, enclose it in double quotation marks. For example, to specify the character value `Tom's` as a constant, enter the following:

```
name="Tom's"
```

Another way to write the same string is to enclose the string in single quotation marks and to express the apostrophe as two consecutive quotation marks. SAS treats the two consecutive quotation marks as one quotation mark:

```
name='Tom''s'
```

The same principle holds true for double quotation marks:

```
name="Tom""s"
```

**CAUTION! Matching quotation marks correctly is important.** Missing or extraneous quotation marks cause SAS to misread both the 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.

## Comparing Character Constants and Character Variables

It is important to remember that character constants are enclosed in quotation marks, but names of character 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 expressions. For example, the character values 'Smith' and 'SMITH' are not equivalent.

## Character Constants Expressed in Hexadecimal Notation

SAS character constants can be expressed in hexadecimal notation. A character hexadecimal constant is a string of an even number of hexadecimal characters enclosed in single or double quotation marks, followed immediately by an X, as in this example:

```
'534153'x
```

A comma can be used to make the string more readable, but it is not part of and does not alter the hexadecimal value. If the string contains a comma, the comma must separate an even number of hexadecimal characters within the string, as in this example:

```
if value='3132,3334'x then do;
```

**Note:** Any trailing blanks or leading blanks within the quotation marks cause an error message to be written to the log.

## Numeric Constants

A numeric constant is a number that appears in a SAS statement. Numeric constants can be presented in many forms, including

- 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 6.1 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 Expressed in Scientific Notation

In scientific notation, the number before the E is multiplied by the power of ten that is indicated by the number after the E. For example, 2E4 is the same as  $2 \times 10^4$  or 20,000. For numeric constants larger than  $(10^{32}) - 1$ , you must use scientific notation. Additional examples follow:

- 1.2e23
- 0.5e-10

## Numeric Constants Expressed in Hexadecimal Notation

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. The constant can contain up to 16 valid hexadecimal characters (0 to 9, A to F). The following are numeric hexadecimal constants:

- 0c1x

- 9x

You can use numeric hexadecimal constants in a DATA step, as follows:

```
data test;
  input abend pib2.;
  if abend=0c1x or abend=0b0ax then do;
    more SAS statements
  run;
```

## Date, Time, andDatetime Constants

You can create a date constant, time constant, or datetime constant by specifying the date or time in single or double quotation marks, followed by a D (date), T (time), or DT (datetime) to indicate the type of value.

Any trailing blanks or leading blanks included within the quotation marks do not affect the processing of the date constant, time constant, or datetime constant.

Use the following patterns to create date and time constants:

'ddmmm<yy>yy'D or "ddmmm<yy>yy"D represents a SAS date value:

- date='1jan2013'd;
- date='01jan09'd;

'hh:mm<::ss.s>'T or "hh:mm<::ss.s>"T represents a SAS time value:

- time='9:25't;
- time='9:25:19pm't;

'ddmmm<yy>yy:hh:mm<::ss.s>'DT or "ddmmm<yy>yy:hh:mm<::ss.s>"DT represents a SAS datetime value:

- if begin='01may12:9:30:00'dt then end='31dec13:5:00:00'dt;
- dtime='18jan2003:9:27:05am'dt;

'yyyy-mm-ddThh:mm:ssZ'DT or 'yyyy-mm-ddThh:mm:ss+|-hh:ss'DT represent a SAS datetime constant for Universal Coordinate Time (UTC) based on the ISO 8601 standard.

- tstamp='2013-05-17T09:15:30-05:00'dt; and  
tstamp='2013-05-17T09:15:30-05'dt; specifies the UTC for Eastern Standard Time.
- tstamp='2013-07-20T12:00:00+00:00'dt; and  
tstamp='2013-07-20T12:00:00Z'dt; specifies the UTC for the zero meridian near Greenwich, England.

For more information about SAS dates, see [Chapter 7, “Dates, Times, and Intervals,” on page 127](#).

## Bit Testing Constants

Bit masks are 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*

The following 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. See “[Comparison Operators](#)” on [page 115](#)for a discussion of these operators.

#### bit-mask

is a string of 0s, 1s, and periods in quotation marks that is immediately followed by a 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 prints a warning in the log, stating that the bit mask is truncated on the left, and continues processing.

The following example tests a character variable:

```
if a='..1.0000'b then do;
```

If the third bit of A (counting from the left) is on, and the fifth through eighth bits are off, the comparison is true and the expression result is 1. Otherwise, the comparison is false and the expression result is 0. The following is a more detailed example:

```
data test;
  input @88 bits $char1.;
  if bits='10000000'b
    then category='a';
  else if bits='01000000'b
    then category='b';
  else if bits='00100000'b
    then category='c';

run;
```

**Note:** Bit masks cannot be used as bit literals in assignment statements. For example, the following statement is not valid:

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

The \$BINARYw. and BINARYw. formats and the \$BINARYw., BINARYw.d, and BITSw.d informats can be useful for bit testing. You can use them to convert character and numeric values to their binary values, and vice versa, and to extract specified bits from input data. See [SAS Formats and Informat Reference](#) for complete descriptions of these formats and informats.

## Avoiding a Common Error with Constants

When you use a string in quotation marks followed by a variable name, always put a blank space between the closing quotation mark and the variable name. Otherwise, SAS might interpret a character constant followed by a variable name as a special SAS constant as illustrated in this table.

**Table 6.2 Characters That Cause Misinterpretation When Following a Character Constant**

Characters That Follow a Character Constant	Possible Interpretations	Examples
b	bit testing constant	'0010000'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

In the following example, '821't is evaluated as a time constant. For more information about SAS time constants, see “[Date, Time, and Datetime Constants](#)” on page 110.

```
data work.europe;
  set ia.europe;
  if flight='821'then
    flight='230';
run;
```

The program writes the following lines to the SAS log:

**Example Code 6.1 Log Results from an Error Caused by a Time Literal Misinterpretation**

```
ERROR: Invalid date/time/datetime constant '821't.
```

```
ERROR 77-185: Invalid number conversion on '821't.
```

```
ERROR 388-185: Expecting an arithmetic
operator.
```

Inserting a blank space between the ending quotation mark and the succeeding character in the IF statement eliminates this 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';
```

---

# SAS Variables in Expressions

---

## Definition

### variable

is a set of data values that describe a given characteristic. A variable can be used in an expression.

---

## Automatic Numeric-Character Conversion

If you specify a variable in an expression, but the variable value does not match the type called for, SAS attempts to convert the value to the expected type. SAS automatically converts character variables to numeric variables and numeric variables to character variables, according to the following rules:

- 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.
- If you use a numeric variable with an operator that requires a character value, such as the concatenation operator, the numeric value is converted to character using the BEST12. format. Because SAS stores the results of the conversion beginning with the right-most byte, you must store the converted values in a variable of sufficient length to accommodate the BEST12. format. You can use the LEFT function to left-justify a result.
- If you use a numeric variable on the left side of an assignment statement and a character variable on the right, the character variable is converted to numeric. In the opposite situation, where the character variable is on the left and the numeric is on the right, SAS converts the numeric variable to character using the BESTn. format, where n is the length of the variable on the left.

When SAS performs an automatic conversion, it prints a note in the SAS log informing you that the conversion took place. 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.

**Note:** You can also use the PUT and INPUT functions to convert data values. These functions can be more efficient than automatic conversion. See “[The Concatenation Operator](#)” on page 123 for an example of the PUT function. See [SAS Functions and CALL Routines: Reference](#) for more details about these functions.

For more information about SAS variables, see [Chapter 4, “SAS Variables,” on page 37](#).

---

## 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 further information about SAS functions, see [SAS Functions and CALL Routines: Reference](#).

---

## SAS Operators in Expressions

---

### Definitions

A SAS operator is a symbol that represents a comparison, arithmetic calculation, or logical operation; a SAS function; or grouping parentheses. SAS uses two major types of operators:

- prefix operators
- infix operators

A prefix operator is an operator that is applied to the variable, constant, function, or parenthetical expression that immediately follows it. The plus sign (+) and minus sign (−) can be used as prefix operators. The word NOT and its equivalent symbols are also prefix operators. The following are examples of prefix operators used with variables, constants, functions, and parenthetical expressions:

- $+y$
- $-25$
- $-\cos(\text{angle1})$
- $+(x*y)$

An infix operator applies to the operands on each side of it (for example,  $6 < 8$ ). Infix operators include the following:

- arithmetic
- comparison
- logical, or Boolean
- minimum
- maximum
- concatenation.

When used to perform arithmetic operations, the plus and minus signs are infix operators.

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 [Chapter 11, “WHERE-Expression Processing,” on page 197](#). The \_NEW\_ operator is used to create an instance of a DATA step component object. For more information, see [Chapter 24, “Using DATA Step Component Objects,” on page 565](#).

## Arithmetic Operators

Arithmetic operators indicate that an arithmetic calculation is performed, as shown in the following table:

**Table 6.3** Arithmetic Operators

Symbol	Definition	Example	Result
<code>**</code>	exponentiation	<code>a**3</code>	raise A to the third power
<code>*</code>	multiplication*	<code>2*Y</code>	multiply 2 by the value of Y
<code>/</code>	division	<code>var/5</code>	divide the value of VAR by 5
<code>+</code>	addition	<code>num+3</code>	add 3 to the value of NUM
<code>-</code>	subtraction	<code>sale-discount</code>	subtract the value of DISCOUNT from the value of SALE

\* The asterisk (\*) is always necessary to indicate multiplication; `2Y` and `2(Y)` are not valid expressions.

If a missing value is an operand for an arithmetic operator, the result is a missing value. See [Chapter 5, “Missing Values,” on page 95](#) for a discussion of how to prevent the propagation of missing values.

See [“Order of Evaluation in Compound Expressions” on page 124](#) for the order in which SAS evaluates these operators.

## Comparison Operators

Comparison operators set up a comparison, operation, or calculation with two variables, constants, or expressions. If the comparison is true, the result is 1. If the comparison is false, the result is 0.

Comparison operators can be expressed as symbols or with their mnemonic equivalents, which are shown in the following table:

Table 6.4 Comparison Operators

Symbol	Mnemonic Equivalent	Definition	Example
=	EQ	equal to	a=3
^=	NE	not equal to*	a ne 3
~=	NE	not equal to	
~=	NE	not equal to	
>	GT	greater than	num>5
<	LT	less than	num<8
>=	GE	greater than or equal to**	sales>=300
<=	LE	less than or equal to***	sales<=100
	IN	equal to one of a list	num in (3, 4, 5)

\* The symbol that you use for NE depends on your personal computer.

\*\* The symbol => is also accepted for compatibility with previous releases of SAS. It is not supported in WHERE clauses or in PROC SQL.

\*\*\* The symbol =< is also accepted for compatibility with previous releases of SAS. It is not supported in WHERE clauses or in PROC SQL.

See “Order of Evaluation in Compound Expressions” on page 124 for the order in which SAS evaluates these operators.

You can add a colon (:) modifier to any of the operators to compare only a specified prefix of a character string. See “Character Comparisons” on page 118 for details.

## The IN Operator

You can use the IN operator to compare a value that is produced by an expression on the left of the operator to a list of values that are given on the right. Individual values 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:

```
expression IN(value-1<...,value-n)
expression IN(value-1<... value-n>)
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](#)” on page 117.

## Numeric Comparisons

SAS makes numeric comparisons that are based on values. 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. If A is 5 and B is 9, then the expression has the value 1, or true. If A and B each have the value 47, then the expression is true and has the value 1.

Comparison operators appear frequently in IF-THEN statements, as in this example:

```
if x<y then c=5;
else c=12;
```

You can also use comparisons in expressions in assignment statements. For example, the preceding statements can be recoded as follows:

```
c=5*(x<y)+12*(x>=y);
```

Since SAS evaluates quantities inside parentheses before performing any operations, the expressions `(x<y)` and `(x>=y)` are evaluated first, and the result (1 or 0) is substituted for the expressions in parentheses. Therefore, if X=6 and Y=8, the expression evaluates as follows:

```
c=5*(1)+12*(0)
```

The result of this statement is C=5.

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 [Chapter 4, “SAS Variables,” on page 37](#) 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 [Chapter 5, “Missing Values,” on page 95](#) for more information.

## The IN Operator in Numeric Comparisons

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

You can use multiple ranges in the same IN list, and you can use ranges with other constants in an IN list. The following example shows a range that is used with other constants to test if X is 0, 1, 2, 3, 4, 5, or 9.

```
if x in (0,9,1:5);
```

You can also use the IN operator to search an array of numeric values. For example, the following code creates an array *a*, defines a constant *x*, and then uses the IN operator to search for *x* in array *a*. Note that the array initialization syntax of array *a{10} (2\*1:5)* creates an array that contains the initial values of 1, 2, 3, 4, 5, 1, 2, 3, 4, 5.

```
data _null_;
array a{10} (2*1:5);
x=99;
y = x in a;
put y=;
a{5} = 99;
y = x in a;
put y=;
run;
```

**Example Code 6.2 Results from Using the IN Operator to Search an Array of Numeric Values (Partial Output)**

```
173 data _null_;
174   array a{10} (2*1:5);
175   x=99;
176   y = x in a;
177   put y=;
178   a{5} = 99;
179   y = x in a;
180   put y=;
181 run;
y=0
y=1
```

Note: PROC SQL does not support this syntax.

## Character Comparisons

You can perform comparisons on character operands, but the comparison always yields a numeric result (1 or 0). Character operands are compared character by character from left to right. Character order depends on the collating sequence, usually ASCII or EBCDIC, used by your computer.

For example, in the EBCDIC and ASCII collating sequences, G is greater than A. Therefore, this expression is true:

```
'Gray' > 'Adams'
```

Two-character values of unequal length are compared as if blanks were attached to the end of the shorter value before the comparison is made. A blank, or missing character value, is smaller than any other printable character value. For example, because . is less than h, this expression is true:

```
'C. Jones' < 'Charles Jones'
```

Since trailing blanks are ignored in a comparison, 'fox ' is equivalent to 'fox'. However, because blanks at the beginning and in the middle of a character value are significant to SAS, ' fox' is not equivalent to 'fox'.

You can compare only a specified prefix of a character expression by using a colon (:) after the comparison operator. SAS truncates the longer value to the length of the shorter value during the comparison. In the following example, the colon modifier after the equal sign tells SAS to look at only the first character of values of the variable LastName and to select the observations with names beginning with the letter S:

```
if lastname=':S';
```

Because printable characters are greater than blanks, both of the following statements select observations with values of LastName that are greater than or equal to the letter S:

- if lastname>='S' ;
- if lastname>=':S' ;

**Note:** 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.

The operations that are discussed in this section show you how to compare entire character strings and the beginnings of character strings. Several SAS character functions enable you to search for and extract values from within character strings. See [SAS Functions and CALL Routines: Reference](#) for complete descriptions of all SAS functions.

## The IN Operator in Character Comparisons

You can use the IN operator with character strings to determine whether a variable's value is among a list of character values. The following statements produce the same results:

- 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;

You can also use the IN operator to search an array of character values. For example, the following code creates an array **a**, defines a constant **x**, and then uses the IN operator to search for **x** in array **a**.

```
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 6.3 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='bl';
193    y = x in a;
194    put y=;
195    a{5} = 'bl';
196    y = x in a;
197    put y=;
198  run;
y=0
y=1

```

## Logical (Boolean) Operators and Expressions

Logical operators, also called Boolean operators, are usually used in expressions to link sequences of comparisons. The logical operators are shown in the following table:

*Table 6.5 Logical Operators*

Symbol	Mnemonic Equivalent	Example
&	AND	(a>b & c>d)
	OR*	(a>b or c>d)
!	OR	
	OR	
¬	NOT**	not (a>b)
◦	NOT	
~	NOT	

\* The symbol that you use for OR depends on your operating environment.

\*\* The symbol that you use for NOT depends on your operating environment.

See “Order of Evaluation in Compound Expressions” on page 124 for the order in which SAS evaluates these operators.

In addition, a numeric expression without any logical operators can serve as a Boolean expression. For an example of Boolean numeric expressions, see “Boolean Numeric Expressions” on page 122.

---

## The AND Operator

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. For example, in the following comparison,

```
a<b& c>0
```

the result is true (has a value of 1) only when both  $A < B$  and  $C > 0$  are 1 (true): that is, when A is less than B and C is positive.

Two comparisons with a common variable linked by AND can be condensed with an implied AND. For example, the following two subsetting IF statements produce the same result:

- if 16<=age and age<=65;
- if 16<=age<=65;

---

## The OR Operator

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. For example, consider the following comparison:

```
a<b|c>0
```

The result is true (with a value of 1) when  $A < B$  is 1 (true) regardless of the value of C. It is also true when the value of  $C > 0$  is 1 (true), regardless of the values of A and B. Therefore, it is true when either or both of those relationships hold.

Be careful when using the OR operator with a series of comparisons (in an IF, SELECT, or WHERE statement, for example). Remember that only one comparison in a series of OR comparisons must be true to make a condition true, and any nonzero, nonmissing constant is always evaluated as true. For more information about how SAS computes Boolean expressions, see “[Boolean Numeric Expressions](#)” on page 122. Therefore, the following subsetting IF statement is always true:

```
if x=1 or 2;
```

SAS first evaluates  $X=1$ , and the result can be either true or false. However, since the 2 is evaluated as nonzero and nonmissing (true), the entire expression is true. In this statement, however, the condition is not necessarily true because either comparison can evaluate as true or false:

```
if x=1 or x=2;
```

---

## The NOT Operator

The prefix operator NOT is also a logical operator. The result of putting NOT in front of a quantity whose value is 0 (false) is 1 (true). That is, the result of negating a false statement is 1 (true). For example, if  $X=Y$  is 0 (false) then  $\text{NOT}(X=Y)$  is 1

(true). The result of NOT in front of a quantity whose value is missing is also 1 (true). The result of NOT in front of a quantity with a nonzero, nonmissing value is 0 (false). That is, the result of negating a true statement is 0 (false).

For example, the following two expressions are equivalent:

- `not (name='SMITH')`
- `name ne 'SMITH'`

Furthermore, NOT(A&B) is equivalent to NOT A|NOT B, and NOT(A|B) is the same as NOT A & NOT B. For example, the following two expressions are equivalent:

- `not (a=b & c>d)`
- `a ne b | c le d`

## Boolean 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.

```
0 | . = False
1 = True
```

For example, suppose that you want to fill in 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:

```
if cost ne . and cost ne 0
  then remarks='Ready to budget';
```

A numeric expression can be simply 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;
```

## The MIN and MAX Operators

The MIN and MAX operators are used 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<B, then A><B returns the value of A.

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 [97](#). For example, the maximum value that is returned by .A>>.Z is the value .Z.

**Note:** In a WHERE statement or clause, the <> operator is equivalent to NE.

## The Concatenation Operator

The concatenation operator (||) concatenates character values. The results of a concatenation operation are usually stored in a variable with an assignment statement, as in `level='grade' || 'A'`. The length of the resulting variable is the sum of the lengths of each variable or constant in the concatenation operation, unless you use a LENGTH or 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. See [SAS Functions and CALL Routines: Reference](#) for descriptions and examples of additional character functions.

For example, in this DATA step, the value that results from the concatenation contains blanks because the length of the Color variable is eight:

```
data namegame;
  length color name $8 game $12;
  color='black';
  name='jack';
  game=color||name;
  put game=;
run;
```

The value of Game is 'black jack'. To correct this problem, use the TRIM function in the concatenation operation as follows:

```
game=trim(color)||name;
```

This statement produces a value of 'blackjack' for the variable Game. The following additional examples demonstrate uses of the concatenation operator:

- If A has the value 'fortune', B has the value 'five', and C has the value 'hundred', then the following statement produces the value 'fortunefivehundred' for the variable D:

```
d=a||b||c;
```

- This example concatenates the value of a variable with a character constant.

```
newname='Mr. or Ms. '||oldname;
```

If the value of OldName is 'Jones', then NewName has the value 'Mr. or Ms. Jones'.

- Because the concatenation operation does not trim blanks, the following expression produces the value 'JOHN SMITH':

```
name='JOHN' || 'SMITH';
```

- This example uses the PUT function to convert a numeric value to a character value. The TRIM function is used to trim blanks.

```
month='sep' ;
year=99;
date=trim(month)||left(put(year,8.));
```

The value of DATE is the character value 'sep99'.

## Order of Evaluation in Compound Expressions

[Table 6.6 on page 124](#) shows the order of evaluation in compound expressions. The table contains the following columns:

### Priority

lists the priority of evaluation. In compound expressions, SAS evaluates the part of the expression containing operators in Group I first, then each group in order.

### Order of Evaluation

lists the rules governing which part of the expression SAS evaluates first.

Parentheses are often used in compound expressions to group operands; expressions within parentheses are evaluated before those outside of them. The rules also list how a compound expression that contains more than one operator from the same group is evaluated.

### Symbols

lists the symbols that you use to request the comparisons, operations, and calculations.

### Mnemonic Equivalent

lists alternate forms of the symbol. In some cases, such as when your keyboard does not support special symbols, you should use the alternate form.

### Definition

defines the symbol.

### Example

provides an example of how to use the symbol or mnemonic equivalent in a SAS expression.

**Table 6.6 Order of Evaluation in Compound Expressions**

Priority	Order of Evaluation	Symbols	Mnemonic Equivalent	Definition	Example
Group I	right to left	**		exponentiation*	<code>y=a**2;</code>
		+		positive prefix**	<code>y=+ (a*b) ;</code>
		-		negative prefix***	<code>z=- (a+b) ;</code>
		◦ ¬ ~	NOT	logical not†	<code>if not z then put x;</code>
		><	MIN	minimum††	<code>x=(a&gt;b) ;</code>
		<>	MAX	maximum	<code>x=(a&lt;&gt;b) ;</code>
Group II	left to right	*		multiplication	<code>c=a*b;</code>
		/		division	<code>f=g/h;</code>
Group III	left to right	+		addition	<code>c=a+b;</code>

Priority	Order of Evaluation	Symbols	Mnemonic Equivalent	Definition	Example
		-		subtraction	f=g-h;
Group IV	left to right	!!		concatenate character values <sup>ttt</sup>	name= 'J'    'SMITH';
Group V <sup>#</sup>	left to right <sup>##</sup>	<	LT	less than	if x<y then c=5;
		<=	LE	less than or equal to	if x le y then a=0;
		=	EQ	equal to	if y eq (x+a) then output;
		~=	NE	not equal to	if x ne z then output;
		>=	GE	greater than or equal to	if y>=a then output;
		>	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);
Group VI	left to right	&	AND	logical and	if a=b & c=d then x=1;
Group VII	left to right	!	OR	logical or <sup>##</sup>	if y=2 or x=3 then a=d;

- \* Because Group I operators are evaluated from right to left, the expression  $x=2^{**}3^{**}4$  is evaluated as  $x=(2^{**}(3^{**}4))$ .
- \*\* 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.
- \*\*\* 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.
- † 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.
- †† For example, the SAS System evaluates  $-3><-3$  as  $-(3)<-3$ , which is equal to  $-(3)$ , which equals +3. This is because Group I operators are evaluated from right to left.
- ††† 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 (!!).
- # Group V 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 the SAS System. When making character comparisons, you can use a colon (:) after any of the comparison operators to compare only the first character or characters of the value. SAS truncates the longer value to the length of the shorter value during the comparison. For example, if name=:P compares the value of the first character of NAME to the letter P.
- ## 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  $(y<z)$ .
- ### 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.



# Dates, Times, and Intervals

---

<b>About SAS Date, Time, and Datetime Values .....</b>	<b>127</b>
Creating a SAS Date .....	127
Definitions .....	128
Julian Date Formats and Astronomical Dates .....	128
Two-Digit and Four-Digit Years .....	131
Five-Digit Years .....	132
The Year 2000 .....	132
Working with SAS Dates and Times .....	134
Examples .....	141
<b>About Date and Time Intervals .....</b>	<b>143</b>
Definitions .....	143
Syntax .....	144
Intervals by Category .....	144
Example: Calculating a Duration .....	146
Boundaries of Intervals .....	147
Single-Unit Intervals .....	147
Multi-Unit Intervals .....	148
Shifted Intervals .....	150
Custom Intervals .....	151
Retail Calendar Intervals: ISO 8601 Compliant .....	151

---

## About SAS Date, Time, and Datetime Values

---

### Creating a SAS Date

You can use these methods to create a SAS date:

- Read a value into SAS with an informat.
- Apply a format to an existing value.
- Use a date function.

## Definitions

### SAS date value

is a value that represents the number of days between January 1, 1960, and a specified date. SAS can perform calculations on dates ranging from A.D. November 1582 to A.D. 19,900. Dates before January 1, 1960, are negative numbers; dates after January 1, 1960, are positive numbers.

- SAS date values account for all leap year days, including the leap year day in the year 2000.
- SAS date values can reliably tell you what day of the week a particular day fell on as far back as September 1752. That was when the calendar was adjusted by dropping several days. SAS day-of-the-week and length-of-time calculations are accurate in the future to A.D. 19,900.
- Various SAS language elements handle SAS date values: functions, formats, and informats.

### SAS time value

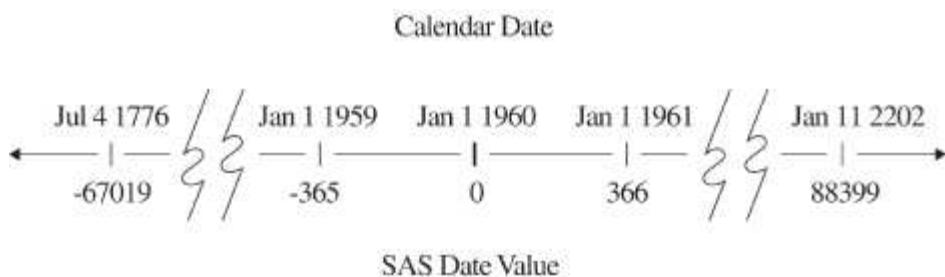
is a value representing the number of seconds since midnight of the current day. SAS time values are between 0 and 86400.

### SAS datetime value

is a value representing the number of seconds between January 1, 1960, and an hour/minute/second within a specified date.

The following figure shows some dates written in calendar form and as SAS date values.

*Figure 7.1 How SAS Converts Calendar Dates to SAS Date Values*



## Julian Date Formats and Astronomical Dates

### SAS Date Values

SAS uses SAS date values, which are ordinal numbers, to calculate dates. SAS date values represent the number of days between January 1, 1960, and a specified date. All SAS formats, informats, and functions use SAS dates. You have to use an informat to convert a Julian date to a SAS date before SAS can use it to perform calculations.

The following SAS language elements do not convert SAS dates to Julian dates. They apply a Julian date format to a SAS date.

Formats	Informats	Functions
JULDAY	JULIAN	DATEJUL
JULIAN	PDJULG	JULDATE
PDJULG	PDJULI	JULDATE7
PDJULI		

SAS can perform calculations on raw SAS date values and on formatted SAS date values. This includes performing calculations on Julian formatted date values.

SAS uses these definitions of Julian dates and Julian formats:

#### Julian date

is the number of continuous days since January 1, 4713 BC, which is also known as an astronomical date.

#### Julian format

is the representation of an ordinal SAS date in the form of a calendar day, YYDDD or YYDD.

SAS uses the Julian format (ordinal date) definition of dates. Julian-related language elements in SAS do not convert SAS dates internally to Julian astronomical dates. These Julian-related language elements make a SAS date look like an ordinal date with the form YYDDD or YYYYDDD. For example, January, 23, 2018 is 18023 when you apply a Julian format in SAS.

You must define the values as SAS dates before using them in calculations. The only way you can convert a SAS date to an astronomical date is to add 2,436,934.5 to the SAS date value. This conversion enables SAS to use the values to perform calculations. Otherwise, SAS treats the values as regular integer numeric values, and you might get unexpected results.

## Example: Performing Calculations on Dates That Have Different Formats

The following example performs these tasks:

- Creates a data set that contains SAS dates.
- Converts the dates into the MMDDYY10 format and the Julian format.
- Performs calculations on the two sets of dates, even though they have different formats.

```
data dates; /* 1 */
  input sas_date;
  datalines;
21519
21522
21528
21535
21545
21555
```

```
21565
;
proc print data = dates;
run;

data dates2; /* 2 */
  set dates;
  formatted_sas_date = sas_date;
  julian_formatted_SAS_date = sas_date;
  format formatted_sas_date mmddyy10. julian_formatted_SAS_date julian.; /* 3 */
run;
proc print data=dates2;
run;

data dates3; /* 4 */
  set dates2;
  datediff=sas_date - lag(julian_formatted_SAS_date); /* 5 */
run;
proc print data =dates3;
run;
```

- 1 Create the data set of SAS dates.
- 2 Convert the dates into the MMDDYY10 format and the Julian format.
- 3 The FORMAT statement creates the two formats for the dates.
- 4 Perform calculations on the formatted dates.
- 5 The LAG function compares the SAS date values in the previous row with the value in the current row and returns the difference.

**Output 7.1** Converting SAS Dates and Using the Results in Calculations

The SAS System	
Obs	sas_date
1	21519
2	21522
3	21528
4	21535
5	21545
6	21555
7	21565

The SAS System				
Obs	sas_date	formatted_sas_date	julian_formatted_SAS_date	
1	21519	12/01/2018	18335	
2	21522	12/04/2018	18338	
3	21528	12/10/2018	18344	
4	21535	12/17/2018	18351	
5	21545	12/27/2018	18361	
6	21555	01/06/2019	19006	
7	21565	01/16/2019	19016	

The SAS System					
Obs	sas_date	formatted_sas_date	julian_formatted_SAS_date	datediff	
1	21519	12/01/2018	18335	.	
2	21522	12/04/2018	18338	3	
3	21528	12/10/2018	18344	6	
4	21535	12/17/2018	18351	7	
5	21545	12/27/2018	18361	10	
6	21555	01/06/2019	19006	10	
7	21565	01/16/2019	19016	10	

## Two-Digit and Four-Digit Years

SAS software can read two-digit or four-digit year values. If SAS encounters a two-digit year, the YEARCUTOFF= option can be used to specify which century within a 100- year span the two-digit year should be attributed to. For example, YEARCUTOFF=1950 means that two-digit years 50 through 99 correspond to 1950

through 1999. Two-digit years 00 through 49 correspond to 2000 through 2049. Note that while the default value of the YEARCUTOFF= option in SAS is 1926, you can adjust the YEARCUTOFF= value in a DATA step to accommodate the range of date values that you are working with at the moment. To correctly handle two-digit years representing dates between 2000 and 2099, you should specify an appropriate YEARCUTOFF= value between 1901 and 2000. For more information, see the “[YEARCUTOFF= System Option](#)” in *SAS System Options: Reference*.

## Five-Digit Years

Although some formats that specify a width large enough to accommodate formatting a five-digit year, such as DATETIME20., the SAS documentation does not display five-digit years.

## The Year 2000

### Using the YEARCUTOFF= System Option

SAS software treats the year 2000 like any other leap year. If you use two-digit year numbers for dates, you probably need to adjust the default setting for the YEARCUTOFF= option to work with date ranges for your data or switch to four-digit years. The following program changes the YEARCUTOFF= value to 1950. This change means that all two-digit dates are now assumed to fall in the 100-year span from 1950 to 2049.

```
options yearcutoff=1950;
data _null_;
  a='26oct02'd;
  put 'SAS date='a;
  put 'formatted date='a date9. ;
run;
```

The PUT statement writes the following lines to the SAS log:

```
SAS date=15639
formatted date=26OCT2002
```

**Note:** Whenever possible, specify a year using all four digits. Most SAS date and time language elements support four-digit year values.

### Example: How YEARCUTOFF= Affects Two- and Four-Digit Years

The following example shows what happens with data that contains both two and four-digit years. By default, the YEARCUTOFF= option is set to 1926.

```
options nodate;

data schedule;
  input @1 jobid $ @6 projdate mmddyy10. ;
  datalines;
```

```

A100 01/15/25
A110 03/15/2025
A200 01/30/96
B100 02/05/12
B200 06/15/2012
;

proc print data=schedule;
   format projdate mmddyy10.;
run;

```

The resulting output from the PROC PRINT statement looks like this:

*Output 7.2 Output Showing Four-Digit Years That Result from Setting YEARCUTOFF= to 1926*

The SAS System		
Obs	jobid	projdate
1	A100	01/15/2025
2	A110	03/15/2025
3	A200	01/30/1996
4	B100	02/05/2012
5	B200	06/15/2012

Here are some facts to note in this example:

- In the data lines in the DATA step, the first record contains a two-digit year of 25, and the second record contains a four-digit year of 2025. The century for the first record defaults to the 2000s because 2025 is in the range of 1926–2025. The four-digit year in the second record is unaffected by the YEARCUTOFF= option.
- In the third record, the century defaults to the 1900s because the year 1996 is in the range of 1926–2025.
- The output from the fourth and fifth records show results that are similar to the first and second records. The fourth record specifies a two-digit year of 12, and the fifth one specifies a four-digit year of 2012. The century in the fourth record defaults to the 2000s because 2012 is in the range of 1926–2025. The four-digit year in the fifth record is unaffected by the YEARCUTOFF= option.

As you can see, specifying a two-digit year might or might not result in the intended century prefix. The optimal value of the YEARCUTOFF= option depends on the range of the dates that you are processing.

In releases SAS 6.06 through SAS 6.12, the default value for the YEARCUTOFF= system option is 1900. Starting with SAS 7, the default value is 1920; starting with SAS 9.4, the default value is 1926.

For more information about how SAS handles dates, see the section on dates, times, and datetime values.

## Practices That Help Ensure Date Integrity

The following practices help ensure that your date values are correct during all the conversions that occur during processing:

- Store dates as SAS date values, not as simple numeric or character values.
- Use the YEARCUTOFF= system option when converting two-digit dates to SAS date values.
- Examine sets of raw data coming into your SAS process to make sure that any dates containing two-digit years are correctly interpreted by the YEARCUTOFF= system option. Look out for the following situations:
  - two-digit years that are distributed over more than a 100-year period. For dates covering more than a 100-year span, you must either use four-digit years in the data, or use conditional logic in a DATA step to interpret them correctly.
  - two-digit years that need an adjustment to the default YEARCUTOFF= range. For example, if the default value for YEARCUTOFF= in your operating environment is 1926 and you have a two-digit date in your data that represents 1925, you have to adjust your YEARCUTOFF= value downward by a year in the SAS program that processes this value.
- Make sure that output SAS data sets represent dates as SAS date values.
- Check your SAS programs to make sure that formats and informats that use two-digit years, such as DATE7., MMDDYY6., or MMDDYY8., are reading and writing data correctly.

**Note:** The YEARCUTOFF= option has no effect on dates that are already stored as SAS date values.

## Working with SAS Dates and Times

### Informats and Formats

SAS converts date, time, and datetime values back and forth between calendar dates and clock times with SAS language elements called formats and informats.

- Formats present a value, recognized by SAS, such as a time or date value, as a calendar date or clock time in a variety of lengths and notations.
- Informats read notations or a value, such as a clock time or a calendar date, which might be in a variety of lengths, and then convert the data to a SAS date, time, or datetime value.

SAS can read date and time values that are delimited by the following characters:

! # \$ % & ( ) \* + - . / : ; < = > ? [ \ ] ^ \_ { | } ~

The blank character can also be used.

Only one delimiter can be used for a date. Otherwise, an error message is written to the SAS log. For example, 01/Jan/2007 uses a single delimiter, and can be read by SAS. In the case of 01-Jan/2007, two different delimiters separate the parts of the date, which results in an error message.

## Date and Time Tools by Task

The following table correlates tasks with various SAS language elements that are available for working with time and date data.

**Table 7.1** Tasks with Dates and Times, Part 1

Task	Type of Language Element	Language Element	Input	Result
Write SAS date values in recognizable forms	Date formats	DATE.	19434	17MAR13
		DATE9.	19434	17MAR2013
		DAY.	19434	17
		DDMMYY.	19434	17/03/13
		DDMMYY10.	19434	17/03/2013
		DDMMYYB.	19434	17 03 13
		DDMMYYB10.	19434	17 03 2013
		DDMMYYC.	19434	17:03:13
		DDMMYYC10.	19434	17:03:2013
		DDMMYYD.	19434	17-03-13
		DDMMYYD10.	19434	17-03-2013
		DDMMYYN.	19434	17032013
		DDMMYYN6.	19434	170313
		DDMMYYP.	19434	17.03.13
		DDMMYYP10.	19434	17.03.2013
		DDMMYY5.	19434	17/03/13
		DDMMYY5S10.	19434	17/03/2013
		DOWNNAME.	19434	Sunday
		JULDAY.*	19434	76
		JULIAN.*	19434	13076
		MMDDYY.	19434	03/17/13
		MMDDYY10.	19434	03/17/2013
		MMDDYYB.	19434	03 17 13
		MMDDYYB10.	19434	03 17 2013
		MMDDYYC.	19434	03:17:13
		MMDDYYC10.	19434	03:17:2013
		MMDDYYD.	19434	03-17-13

Task	Type of Language Element	Language Element	Input	Result
Write SAS date values in recognizable forms	Date formats	MMDDYYD10.	19434	03-17-2013
		MMDDYYN.	19434	03172013
		MMDDYYN8.	19434	03172013
		MMDDYYP.	19434	03.17.13
		MMDDYYP10.	19434	03.17.2013
		MMDDYY5.	19434	03/17/13
		MMDDYY510.	19434	03/17/2013
		MMYY.	19434	03M2013
		MMYYC.	19434	03:2013
		MMYYD.	19434	03-2013
		MMYYN.	19434	032013
		MMYYP.	19434	03.2013
		MMYY5.	19434	03/2013
		MONNAME.	19434	March
		MONTH.	19434	3
		MONYY.	19434	MAR13
		PDJULG. *	19434	2013076F
		PDJULI. *	19434	0100076F
		WEEKDATE.	19434	Sunday, March 17, 2013
		WEEKDAY.	19434	1
		WORDDATE.	19434	March 17, 2013
		WORDDATX.	19434	17 March 2013
Quarter formats	Quarter formats	QTR.	19434	1
		QTRR.	19434	I
	Time formats	TIME.	19434	5:23:54
	TIMEAMPM.	19434	5:23:54 AM	
	TOD.	19434	05:23:54	

Task	Type of Language Element	Language Element	Input	Result
Write SAS date values in recognizable forms	Year formats	YEAR.	19434	2013
		YYMM.	19434	2013M03
		YYMMC.	19434	2013:03
		YYMMD.	19434	2013-03
		YYMMP.	19434	2013.03
		YYMMS.	19434	2013/03
		YYMMN.	19434	201303
		YYMMDD.	19434	13-03-17
		YYMON.	19434	2013MAR
	Year/Quarter formats	YYQ.	19434	2013Q1
		YYQC.	19434	2013:1
		YYQD.	19434	2013-1
		YYQP.	19434	2013.1
		YYQS.	19434	2013/1
		YYQN.	19434	20131
		YYQR.	19434	2013QI
		YYQRC.	19434	2013:I
		YYQRD.	19434	2013-I
		YYQRP.	19434	2013.I
		YYQRS.	19434	2013/I
		YYQRN.	19434	2013I

\* In SAS, a Julian date is a date in the form YYNNN or YYYYNNN, where YY is a two-digit year, YYYY is a four-digit year, and NNN is the ordinal offset from January 1 of the year YY or YYYY. SAS processes Julian dates only for valid SAS dates.

Table 7.2 Tasks with Dates and Times, Part 2

Task	Type of Language Element	Language Element	Input	Result
Date Tasks				

Task	Type of Language Element	Language Element	Input	Result
Read calendar dates as SAS date  Note: YEARCUTOFF=1926	Date informats	DATE.	17MAR13	19434
		DATE9.	17MAR2013	19434
		DDMMYY.	170313	19434
		DDMMYY8.	17032013	19434
		JULIAN.*	13076	19434
		JULIAN7.*	2013077	19434
		MMDDYY.	031713	19434
		MMDDYY8.	03172013	19434
		MONYY.	MAR13	19418
		YYMMDD.	130317	19434
		YYMMDD8.	20130317	19434
		YYQ.	13q1	19359
		DATETIME	17MAR2013 00:00:00	1679097600
		TIME	14:45:32	53132
Return today's date as a SAS date value	Date functions	DATE() or TODAY() (equivalent)	( )	The SAS date value for today.
Extract calendar dates from SAS	Date functions	DAY	19434	17
		HOUR	19434	5
		JULDATE*	19434	13076
		JULDATE7*	19434	2013076
		MINUTE	19434	23
		MONTH	19434	3
		QTR	19434	1
		SECOND	19434	54
		WEEKDAY	19434	1
		YEAR	19434	2013

Task	Type of Language Element	Language Element	Input	Result
Write a date as a constant in an expression	SAS date constant	'ddmmmyy'd or 'ddmmmyyy'd	'17mar13'd  '17mar2013'd	19434
Write today's date as a string	SYSDATE automatic macro variable	SYSDATE	&SYSDATE	The date at the time of SAS initialization in the form DDMMYY.
	SYSDATE9	SYSDATE9	&SYSDATE9	The date at time of SAS initialization, in the form DDMMYYYY.
<b>Time Tasks</b>				
Write SAS time values as time values	time formats	HHMM.	19434	5:24
		HOUR.	19434	5
		MMSS.	19434	323
		TIME.	19434	5:23:54
		TIMEAMPM.	19434	5:23:54 AM
		TOD.	19434	05:23:54
Read time values as SAS time values	Time informats	TIME.	05:23:54	19434
Write the current time as a string	SYSTIME automatic macro variable	SYSTIME	&SYSTIME	The time at the moment of execution, in the form HH:MM
Return the current time of day as a SAS time value	Time functions	TIME( )	( )	The SAS time value at moment of execution, in the form NNNNN.NNN.
Return the time part of a SAS datetime value	Time functions	TIMEPART	17mar2013 05:11:43	5:11:43
<b>Datetime Tasks</b>				
Write SAS datetime values as datetime values	Datetime formats	DATEAMPM	1679097600	17MAR13:12:00 :00 AM
		DATETIME	1679097600	17MAR13:00:00 :00
Read datetime values as SAS datetime values	Datetime informats	DATETIME	17MAR13:00:00:00	1679097600

Task	Type of Language Element	Language Element	Input	Result
Return the current date and time of day as a SAS datetime value	Datetime functions	DATETIME()	()	The SAS datetime value at the moment of execution, in the form NNNNNNNNNN.N.
<b>Interval Tasks</b>				
Return the number of specified time intervals that lie between the two date or datetime values	Interval functions	INTCK	week2 01aug60 01jan13	1368
Advances a date, time, or datetime value by a given interval, and returns a date, time, or datetime value	Interval functions	INTNX	day 17mar12 365	19434

\* In SAS, a Julian date is a date in the form YYNNN or YYYYNNN, where YY is a two-digit year, YYYY is a four-digit year, and NNN is the ordinal offset from January 1 of the year YY or YYYY. SAS processes Julian dates only for valid SAS dates.

SAS also supports these formats and informats:

- ISO 8601 basic and extended forms for dates, times, datetimes, durations, intervals, and time zones. For more information, see “[Working with Dates and Times by Using the ISO 8601 Basic and Extended Notations](#)” in *SAS Formats and Informats: Reference* and “[Reading Dates and Times by Using the ISO 8601 Basic and Extended Notations](#)” in *SAS Formats and Informats: Reference*.
- International formats and informats that are equivalent to some of the most commonly used English-language date formats and informats. For more information, see “[Dictionary of Formats for NLS](#)” in *SAS National Language Support (NLS): Reference Guide* and “[Dictionary of Formats for NLS](#)” in *SAS National Language Support (NLS): Reference Guide*.

---

## Examples

### Example 1: Displaying Date, Time, and Datetime Values as Recognizable Dates and Times

The following example demonstrates how a value might be displayed as a date, a time, or a datetime. Remember to select the SAS language element that converts a SAS date, time, or datetime value to the intended date, time, or datetime format. See the previous tables for examples.

**Note:**

- Time formats count the number of seconds within a day, so the values are between 0 and 86400.

- DATETIME formats count the number of seconds since January 1, 1960. For datetimes that are greater than 02JAN1960:00:00:01 (integer of 86401), the datetime value is always greater than the time value.
- When in doubt, look at the contents of your data set for clues as to which type of value you are dealing with.

This program uses the DATETIME, DATE, and TIMEAMPM formats to display the value 86399 to a date and time, a calendar date, and a time.

```

options nodate;
data test;
  Time1=86399;
  format Time1 datetime.;
  Date1=86399;
  format Date1 date9.;
  Time2=86399;
  format Time2 timeampm.;
run;
proc print data=test;
  title 'Same Number, Different SAS Values';
  footnote1 'Time1 is a SAS DATETIME value';
  footnote2 'Date1 is a SAS DATE value';
  footnote3 'Time2 is a SAS TIME value';
run;
footnote;

```

**Output 7.3 Datetime, Date, and Time Values for 86399**

<b>Same Number, Different SAS Values</b>			
<b>Obs</b>	<b>Time1</b>	<b>Date1</b>	<b>Time2</b>
<b>1</b>	01JAN60:23:59:59	20JUL2196	11:59:59 PM

Time1 is a SAS DATETIME value  
 Date1 is a SAS DATE value  
 Time2 is a SAS TIME value

## Example 2: Reading, Writing, and Calculating Date Values

This program reads four regional meeting dates and calculates the dates on which announcements should be mailed.

```

data meeting;

  input region $ mtg : mmddyy8. ;
  sendmail=mtg-45;
 datalines;
N 11-24-12
S 12-28-12
E 12-03-12
W 10-04-12
;

```

```
proc print data=meeting;
  format mtg sendmail date9.;
  title 'When To Send Announcements';
run;
```

*Output 7.4 Calculated Date Values: When to Send Mail*

### When To Send Announcements

Obs	region	mtg	sendmail
1	N	24NOV2012	10OCT2012
2	S	28DEC2012	13NOV2012
3	E	03DEC2012	19OCT2012
4	W	04OCT2012	20AUG2012

---

## About Date and Time Intervals

---

### Definitions

#### duration

is an integer representing the difference between any two dates or times or datetimes. Date durations are integer values representing the difference, in the number of days, between two SAS dates. Time durations are decimal values representing the number of seconds between two times or datetimes.

**TIP** Date and datetimes durations can be easily calculated by subtracting the smaller date or datetime from the larger. When dealing with SAS times, special care must be taken if the beginning and the end of a duration are on different calendar days. Whenever possible, the simplest solution is to use datetimes rather than times.

#### interval

is a unit of measurement that SAS can count within an elapsed period of time, such as DAYS, MONTHS, or HOURS. SAS determines date and time intervals based on fixed points on the calendar, the clock, or both. The starting point of an interval calculation defaults to the beginning of the period in which the beginning value falls, which might not be the actual beginning value specified. For example, if you are using the INTCK function to count the months between two dates, regardless of the actual day of the month specified by the date in the beginning value, SAS treats it as the first of that month.

## Syntax

SAS provides date, time, and datetime intervals for counting different periods of elapsed time. You can create multiples of the intervals and shift their starting point. Use them with the INTCK and INTNX functions and with procedures that support numbered lists (such as the PLOT procedure). This is the form of an interval:

*name*<*multiple*><.*starting-point*>

The terms in an interval have the following definitions:

*name*

is the name of the interval. See the following table for a list of intervals and their definitions.

*multiple*

creates a multiple of the interval. *multiple* can be any positive number. The default is 1. For example, YEAR2 indicates a two-year interval.

.*starting-point*

is the starting point of the interval. By default, the starting point is 1. A value greater than 1 shifts the start to a later point within the interval. The unit for shifting depends on the interval, as shown in the following table. For example, YEAR.3 specifies a yearly period from the first of March through the end of February of the following year.

## Intervals by Category

Table 7.3 Intervals Used with Date and Time Functions

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
Date	DAY	Daily intervals	Each day	Days	DAY3	Three-day intervals starting on Sunday
	WEEK	Weekly intervals of seven days	Each Sunday	Days (1=Sunday ... 7=Saturday)	WEEK.7	Weekly with Saturday as the first day of the week
	WEEKDAY <daysW>	Daily intervals with Friday-Saturday-Sunday	Each day	Days	WEEKDAY1W	Six-day week with Sunday as a weekend day

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
		counted as the same day (five-day work week with a Saturday-Sunday weekend). <i>days</i> identifies the weekend days by number (1=Sunday ... 7=Saturday). By default, <i>days</i> =17.			WEEKDAY35W	Five-day week with Tuesday and Thursday as weekend days (W indicates that day 3 and day 5 are weekend days)
	TENDAY	Ten-day intervals (a U.S. automobile industry convention)	First, eleventh, and twenty-first of each month	Ten-day periods	TENDAY4.2	Four ten-day periods starting at the second TENDAY period
	SEMIMONTH	Half-month intervals	First and sixteenth of each month	Semi-monthly periods	SEMIMONTH2.2	Intervals from the sixteenth of one month through the fifteenth of the next month
	MONTH	Monthly intervals	First of each month	Months	MONTH2.2	February-March, April-May, June-July, August-September, October-November, and December-January of the following year
	QTR	Quarterly (three-month) intervals	January 1 April 1 July 1 October 1	Months	QTR3.2	Three-month intervals starting on April 1, July 1, October 1, and January 1
	SEMIYEAR	Semiannual (six-month) intervals	January 1 July 1	Months	SEMIYEAR.3	Six-month intervals, March-August, and September-February

Category	Interval	Definition	Default Starting Point	Shift Period	Example	Description
	YEAR	Yearly intervals	January 1	Months		
Datetime	Add DT to any of the date intervals	Interval corresponding to the associated date interval	Midnight of January 1, 1960		DTMONTH	
					DTWEEKDAY	
Time	SECOND	Second intervals	Start of the day (midnight)	Seconds		
	MINUTE	Minute intervals	Start of the day (midnight)	Minutes		
	HOUR	Hourly intervals	Start of the day (midnight)	Hours		

---

## Example: Calculating a Duration

This program reads the project start and end dates. Then, the program calculates the duration between them.

```

options nodate pageno=1 linesize=80 pagesize=60;

data projects;
  input Projid @5 startdate date9. @15 enddate date9. ;
  Duration=enddate-startdate;
  datalines;
398 17oct1997 02nov1997
942 22jan1998 11mar1998
167 15dec1999 15feb2000
250 04jan2001 11jan2001
;

proc print data=projects;
  format startdate enddate date9. ;
  title 'Days Between Project Start and Project End';
run;

```

**Output 7.5 Calculating the Duration between Start and End Dates****Days Between Project Start and Project End**

Obs	Projid	startdate	enddate	Duration
1	398	17OCT1997	02NOV1997	16
2	942	22JAN1998	11MAR1998	48
3	167	15DEC1999	15FEB2000	62
4	250	04JAN2001	11JAN2001	7

**Boundaries of Intervals**

SAS associates date and time intervals with fixed points on the calendar. For example, the MONTH interval represents the time from the beginning of one calendar month to the next, not a period of 30 or 31 days. When you use date and time intervals (for example, with the INTCK or INTNX functions), SAS bases its calculations on the calendar divisions that are present. Consider the following examples:

**Table 7.4 Using INTCK and INTNX**

Example	Results	Explanation
<pre>mnthnum1=intck( 'month', '25aug2000'd, '05sep2000'd);</pre>	mnthnum1=1	The number of MONTH intervals the INTCK function counts depends on whether the first day of a month falls within the period.
<pre>mnthnum2=intck( 'month', '01aug2000'd, '31aug2000'd);</pre>	mnthnum2=0	
<pre>next=intnx('month', '25aug2000'd,1);</pre>	next represents 01sep2000	The INTNX function produces the SAS date value that corresponds to the beginning of the next interval.

**Note:** The only intervals that do not begin on the same date in each year are WEEK and WEEKDAY. A Sunday can occur on any date because the year is not divided evenly into weeks.

**Single-Unit Intervals**

Single-unit intervals begin at the following points on the calendar:

**Table 7.5** Single-Unit Intervals

Single-Unit Interval	Beginning Point on the Calendar
DAY	each day
WEEKDAY	for a standard weekday ■ <i>Start day–End day</i> ■ Monday–Monday ■ Tuesday–Tuesday ■ Wednesday–Wednesday ■ Thursday–Thursday ■ Friday–Sunday
WEEK	each Sunday
TENDAY	the first, eleventh, and twenty-first of each month
SEMIMONTH	the first and sixteenth of each month
MONTH	the first of each month
QTR	the first of January, April, July, and October
SEMIYEAR	the first of January and July
YEAR	the first of January

Single-unit time intervals begin as follows:

**Table 7.6** Single-Unit Time Intervals

Single-Unit Time Intervals	Beginning Point
SECOND	each second
MINUTE	each minute
HOUR	each hour

## Multi-Unit Intervals

### Multi-Unit Intervals Other Than Multi-Week Intervals

Multi-unit intervals, such as MONTH2 or DAY50, also depend on calendar measures, but they introduce a new problem: SAS can find the beginning of a unit (for example, the first of a month), but where does that unit fall in the interval? For

example, does the first of October mark the first or the second month in a two-month interval?

For all multi-unit intervals except multi-week intervals, SAS creates an interval beginning on January 1, 1960, and counts forward from that date to determine where individual intervals begin on the calendar. As a practical matter, when a year can be divided evenly by an interval, think of the intervals as beginning with the current year. Thus, MONTH2 intervals begin with January, March, May, July, September, and November. Consider this example:

*Table 7.7 Month2 Intervals*

SAS statements	Results
howmany1=intck('month2','15feb2000'd, '15mar2000'd);	howmany1=1
count=intck('day50','01oct1998'd, '01jan1999'd);	count=1

In the above example, SAS counts 50 days beginning with January 1, 1960; then another 50 days; and so on. As part of this count, SAS counts one DAY50 interval between October 1, 1998, and January 1, 1999. For example, to determine the date on which the next DAY50 interval begins, use the INTNX function, as follows:

*Table 7.8 Using the INTNX Function*

SAS statements	Results
start=intnx('day50','01oct98'd,1);	SAS date value 14200, or Nov 17, 1998

The next interval begins on November 17, 1998.

Time intervals (those that represent divisions of a day) are aligned with the start of the day, that is, midnight. For example, HOUR8 intervals divide the day into the periods 00:00 to 08:00, 8:00 to 16:00, and 16:00 to 24:00 (the next midnight).

## Multi-Week Intervals

Multi-week intervals, such as WEEK2, present a special case. In general, weekly intervals begin on Sunday, and SAS counts a week whenever it passes a Sunday. However, SAS cannot calculate multi-week intervals based on January 1, 1960, because that date fell on a Friday, as shown:

*Figure 7.2 Calculating Multi-Week Intervals*

Dec	Su	Mo	Tu	We	Th	Fr	Sa	Jan
1959	27	28	29	30	31	1	2	1960

Therefore, SAS begins the first interval on Sunday of the week containing January 1, 1960—that is, on Sunday, December 27, 1959. SAS counts multi-week intervals from that point. The following example counts the number of two-week intervals in the month of August 1998:

**Table 7.9** Counting Two-Week Intervals

SAS Statements	Results
count=intck('week2','01aug98'D, '31aug98'D);	count=3

To see the beginning date of the next interval, use the INTNX function, as shown here:

**Table 7.10** Using INTNX to See the Beginning Date of an Interval

SAS Statements	Results
begin=intnx('week2','01aug1998'd,1);	"Begin" represents SAS date 14093 or August 02, 1998

The next interval begins on August 16.

## Shifted Intervals

### Using Shifted Intervals

Shifting the beginning point of an interval is useful when you want to make the interval represent a period in your data. For example, if your company's fiscal year begins on July 1, you can create a year beginning in July by specifying the YEAR.7 interval. Similarly, you can create a period matching U.S. presidential elections by specifying the YEAR4.11 interval. This section discusses how to use shifted intervals and how SAS creates them.

### How to Use Shifted Intervals

When you shift a time interval by a subperiod, the shift value must be less than or equal to the number of subperiods in the interval. For example, YEAR.12 is valid (yearly periods beginning in December), but YEAR.13 is not. Similarly, YEAR2.25 is not valid because there is no twenty-fifth month in the two-year period.

In addition, you cannot shift an interval by itself. For example, you cannot shift the interval MONTH because the shifting subperiod for MONTH is one month and MONTH contains only one monthly subperiod. However, you can shift multi-unit intervals by the subperiod. For example, MONTH2.2 specifies bimonthly periods starting on the first day of the second month.

### How SAS Creates Shifted Intervals

For all intervals except those based on weeks, SAS creates shifted intervals by creating the interval based on January 1, 1960, by moving forward the required number of subperiods, and by counting shifted intervals from that point. For example, suppose you create a shifted interval called DAY50.5. SAS creates a 50-day interval in which January 1, 1960, is day 1. SAS then moves forward to day 5. (Note that the difference, or amount of movement, is four days.) SAS begins

counting shifted intervals from that point. The INTNX function demonstrates that the next interval begins on January 5, 1960:

**Table 7.11** Using INTNX to Determine When an Interval Begins

SAS Statements	Results
<code>start=intnx('day50.5','01jan1960'd,1);</code>	SAS date value 4, or Jan 5, 1960

For shifted intervals based on weeks, SAS first creates an interval based on Sunday of the week containing January 1, 1960 (that is, December 27, 1959). Then, it moves forward the required number of days. For example, suppose you want to create the interval WEEK2.8 (biweekly periods beginning on the second Sunday of the period). SAS measures a two-week interval based on Sunday of the week containing January 1, 1960, and begins counting shifted intervals on the eighth day of that. The INTNX function shows the beginning of the next interval:

**Table 7.12** Using the INTNX Function to Show the Beginning of the Next Interval

SAS Statements	Results
<code>start=intnx('week2.8','01jan1960'd,1);</code>	SAS date value 2, or Jan 3, 1960

You can also shift time intervals. For example, HOUR8.7 intervals divide the day into the periods 06:00 to 14:00, 14:00 to 22:00, and 22:00 to 06:00.

## Custom Intervals

You can define custom intervals and associate interval data sets with new interval names when you use the INTERVALDS= system option. An interval name cannot be a reserved SAS name. The dates for these intervals are located in a SAS data set that you create. The data set must contain the variable Begin. For each observation, the Begin variable represents the start of an interval. You can specify a second variable, End, to represent the end of the interval, but it is not required. If the End variable is not present in the data set, the end of an interval is inferred by the next Begin variable value. After the custom intervals have been defined, you can use them with the INTCK and INTNX functions just as you would use standard intervals.

The INTERVALDS= system option enables you to increase the number of allowable intervals. In addition to the standard list of intervals (DAY, WEEKDAY, and so on), the names that are listed in INTERVALDS= are valid as well.

**Note:** Nested custom intervals are not supported.

## Retail Calendar Intervals: ISO 8601 Compliant

The retail industry often accounts for its data by dividing the yearly calendar into four 13-week periods, based on one of the following formats: 4-4-5, 4-5-4, and 5-4-4.

The first, second, and third numbers specify the number of weeks in the first, second, and third month of each period, respectively. Retail calendar intervals facilitate comparisons across years, because week definitions remain consistent from year to year.

The intervals that are created from the formats can be used in any of the following functions: INTCINDEX, INTCK, INTCYCLE, INTFIT, INTFMT, INTGET, INTINDEX, INTNX, INTSEAS, INTSHIFT, and INTTEST.

The following table lists calendar intervals that are used in the retail industry and that are ISO 8601 compliant.

**Table 7.13** Calendar Intervals Used in the Retail Industry

Interval	Description
YEARV	specifies ISO 8601 yearly intervals. The ISO 8601 year begins on the Monday on or immediately preceding January 4. Note that it is possible for the ISO 8601 year to begin in December of the preceding year. Also, some ISO 8601 years contain a leap week. The beginning subperiod s is written in ISO 8601 weeks (WEEKV).
R445YR	is the same as YEARV except that in the retail industry the beginning subperiod s is 4-4-5 months (R445MON).
R454YR	is the same as YEARV except that in the retail industry the beginning subperiod s is 4-5-4 months (R454MON).
R544YR	is the same as YEARV except that in the retail industry the beginning subperiod s is 5-4-4 months (R544MON).
R445QTR	specifies retail 4-4-5 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 4-4-5 months (R445MON).
R454QTR	specifies retail 4-5-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 4-5-4 months (R454MON).
R544QTR	specifies retail 5-4-4 quarterly intervals (every 13 ISO 8601 weeks). Some fourth quarters contain a leap week. The beginning subperiod s is 5-4-4 months (R544MON).
R445MON	specifies retail 4-4-5 monthly intervals. The 3rd, 6th, 9th, and 12th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. All other months are four ISO 8601 weeks long. R445MON intervals begin with the 1st, 5th, 9th, 14th, 18th, 22nd, 27th, 31st, 35th, 40th, 44th, and 48th weeks of the ISO year. The beginning subperiod s is 4-4-5 months (R445MON).
R454MON	specifies retail 4-5-4 monthly intervals. The 2nd, 5th, 8th, and 11th months are five ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R454MON intervals begin with the 1st, 5th, 10th, 14th, 18th, 23rd, 27th, 31st, 36th, 40th, 44th, and 49th weeks of the ISO year. The beginning subperiod s is 4-5-4 months (R454MON).

Interval	Description
R544MON	specifies retail 5-4-4 monthly intervals. The 1st, 4th, 7th, and 10th months are five ISO 8601 weeks long. All other months are four ISO 8601 weeks long with the exception that some 12th months contain leap weeks. R544MON intervals begin with the 1st, 6th, 10th, 14th, 19th, 23rd, 27th, 32nd, 36th, 40th, 45th, and 49th weeks of the ISO year. The beginning subperiod s is 5-4-4 months (R544MON).
WEEKV	specifies ISO 8601 weekly intervals of seven days. Each week begins on Monday. The beginning subperiod s is calculated in days (DAY). Note that WEEKV differs from WEEK in that WEEKV.1 begins on Monday, WEEKV.2 begins on Tuesday, and so on.



# Error Processing and Debugging

<b>Types of Errors in SAS .....</b>	<b>155</b>
Summary of Types of Errors That SAS Recognizes .....	155
Syntax Errors .....	156
Semantic Errors .....	158
Execution-Time Errors .....	160
Data Errors .....	163
Macro-related Errors .....	165
<b>Error Processing in SAS .....</b>	<b>165</b>
Syntax Check Mode .....	165
Processing Multiple Errors .....	166
Checkpoint Mode and Restart Mode .....	167
Using System Options to Control Error Handling .....	172
Using Return Codes .....	173
Other Error-Checking Options .....	173
<b>Debugging Logic Errors in the DATA Step .....</b>	<b>174</b>
In the SAS Windowing Environment .....	174
In SAS Studio 5.2 and Later Releases .....	174

---

## Types of Errors in SAS

---

### Summary of Types of Errors That SAS Recognizes

SAS performs error processing during both the compilation and the execution phases of SAS processing. You can debug SAS programs by understanding processing messages in the SAS log and then fixing your code. You can use the DATA Step Debugger to detect logic errors in a DATA step during execution.

SAS recognizes five types of errors.

**Table 8.1** 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 first attempts to correct the error by attempting to interpret what you mean. Then SAS continues processing your program based on its 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 NOAUTOCRECT system option. For more information, see the AUTOCRECT system option in the *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;
  x=1;
run;

proc print data=temp;
run;
```

**Example Code 8.1** 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. This is because SAS statements are free-format (they can begin and end anywhere). 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 8.2 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 of running SAS you use, as well as on 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 165](#).)

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 8.3 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 8.4 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.

```

An example of a semantic error that occurs at execution time is when a "NOTE: SAS went to a new line when input statement reached past the end of a line." is output. This note is written to the SAS log when FLOWOVER is used and all the variables in the INPUT statement cannot be fully read.

Another semantic error is the detection of a variable that is not initialized. 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, SAS stops processing a DATA step and writes an error message to the SAS log.

---

## Execution-Time Errors

### Definition

Execution-time errors are errors that 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.<sup>1</sup> The location of an execution-time error is usually given as line and column numbers in a note or error message.

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 the CLEANUP system option in [SAS System Options: Reference](#), and in the SAS documentation for your operating environment.

### 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

---

1. When you run SAS in noninteractive mode, more serious errors can cause SAS to enter syntax check mode and stop processing the program.

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 8.5** 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 8.1** 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 at the time 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 8.6** 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 8.7 SAS Log: Data Error**

```

234  data age;
235      input Name $ Number;
236      datalines;
NOTE: Invalid data for Number in line 238 5-6.
RULE:     -----+-----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 8.2 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

If you want processing to stop when a statement in a DATA step has a syntax error, you can enable SAS to enter syntax check mode. You do this by setting the SYNTAXCHECK system option in batch or non-interactive mode, or by setting the DMSSYNCHK system option in the windowing environment.

SAS can enter syntax check mode only if your program creates a data set. If you use the DATA \_NULL\_ statement, then SAS cannot enter syntax check mode because no data set is created. In this case, using the SYNTAXCHECK or DMSSYNCHK system option has no effect.

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

You use the SYNTAXCHECK system option to enable syntax check mode when you run SAS in non-interactive or batch mode. You use the DMSSYNCHK system option to enable syntax check mode when you run SAS in the windowing environment. You can use these system options only if your program creates a data set. If you use the DATA \_NULL\_ statement, then these options are ignored.

To disable syntax check mode, use the NOSYNTAXCHECK and NODMSSYNCHK system options.

In an OPTIONS statement, place the OPTIONS statement that enables SYNTAXCHECK or DMSSYNCHK before the step for which you want it to apply. If you place the OPTIONS statement inside a step, then SYNTAXCHECK or DMSSYNCHK does not take effect until the beginning of the next step.

For more information about these system options, see “[DMSSYNCHK System Option](#)” in *SAS System Options: Reference* and “[SYNTAXCHECK System Option](#)” in *SAS System Options: Reference*.

## Processing Multiple Errors

Depending on the type and severity of the error, the method that you use to run SAS, and your operating environment, SAS either stops program processing or flags errors and continues processing. SAS continues to check individual statements in procedures after it finds certain types of errors. In some cases SAS can detect multiple errors in a single statement and might issue more error messages for a given situation. This is likely to occur if the statement containing the error creates an output SAS data set.

The following example illustrates a statement with two errors:

```
data temporary;
   Item1=4;
run;

proc print data=temporary;
   var Item1 Item2 Item3;
run;
```

**Example Code 8.8** 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.

When you are running debugged production programs that are unlikely to encounter errors, you might want to force SAS to abend after a single error occurs. You can use the ERRORABEND system option to do this.

## Checkpoint Mode and Restart Mode

### Overview of Checkpoint Mode and Restart Mode

When used together, checkpoint mode and restart mode create an environment where batch programs that terminate before completing can be resubmitted without rerunning steps or labeled code sections that have already completed. Execution resumes with either the DATA or PROC step or the labeled code section that was executing when the failure occurred.

A labeled code section is the SAS code that begins with *label*: outside of a DATA or PROC step and ends with the RUN statement that precedes the next *label*: that is outside of a DATA or PROC step,. Labels must be unique. Consider using labeled code sections when you want to group DATA or PROC steps that might need to be grouped together because the data for one is dependent on the other.

The following example program has two labeled code sections. The first labeled code section begins with the label readSortData: and ends with the run; statement for proc sort data=mylib.mydata;. The second labeled code section starts with the label report: and ends with the run; statements for proc report data=mylib.mydata;.

```

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.
  - CHKPCTCLEAN 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.
  - STEPCHKPLIB or LABELCHKPLIB specifies the libref of the library where SAS saves the checkpoint-restart data.
  - NOWORKTERM saves the Work library when SAS ends.
  - NOWORKINIT does not initialize the Work library when SAS starts.
  - ERRORCHECK STRICT puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.

- **ERRORABEND** specifies whether SAS terminates for most errors.

In the Windows operating environment, the following SAS command starts a batch program in checkpoint mode using a user-specified checkpoint-restart library:

```
sas -sysin 'c:\mysas\myprogram.sas' -labelchkpt -labelchkptlib mylibref  
-noworkterm -noworkinit -errorcheck strict -errorabend
```

In this case, the first statement in MyProgram.sas is the LIBNAME statement that defines the **MyLibref** libref.

## Restarting Batch Programs

To resubmit a batch SAS session using the checkpoint-restart data that is saved in the Work library, include these system options when SAS starts:

- **SYSIN**, if required in your operating environment, names the batch program.
- **STEPCHKPT** or **LABELCHKPT** continues checkpoint mode.
- **STEPRESTART** or **LABELRESTART** enables restart mode, indicating to SAS to use the checkpoint-restart data.
- **NOWORKINIT** starts SAS using the Work library from the previous SAS session.
- **NOWORKTERM** saves the Work library when SAS ends.
- **ERRORCHECK STRICT** puts SAS in syntax-check mode when an error occurs in the LIBNAME, FILENAME, %INCLUDE, and LOCK statements.
- **ERRORABEND** specifies whether SAS terminates for most errors.
- **CHKPTCLEAN** specifies whether to erase files in the Work library if the batch program runs successfully.

In the Windows operating environment, the following SAS command resubmits a batch program whose checkpoint-restart data was saved to the Work library:

```
sas -sysin 'c:\mysas\mysasprogram.sas' -stepchkpt -steprestart -noworkinit  
-noworkterm -errorcheck strict -errorabend -chkptclean
```

By specifying the **NOWORKTERM** system option 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 only issues 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.

**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 IORCMMSG 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 SYMSG 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.

## Debugging Logic Errors in the DATA Step

### In the SAS Windowing Environment

The DATA step debugger is included with the SAS windowing environment (or SAS display manager) and it is invoked by specifying the [DEBUG option](#) in the SAS [DATA statement](#).

```
data test / debug;
  x=1;
run;
```

For information about how to debug SAS programs when using SAS in the SAS Windowing Environment, see [Using the DATA Step Debugger](#).

### In SAS Studio 5.2 and Later Releases

If you are using the DATA step debugger that is included with SAS Studio 5.2 or later releases, see [Getting Started with SAS Code Debugger](#). The DEBUG option in the DATA statement is not supported in SAS Studio.

# SAS Output

---

<b>Definitions for SAS Output .....</b>	<b>175</b>
<b>Routing and Customizing SAS Output .....</b>	<b>177</b>
Default Output Destination .....	177
Changing the Output Destination .....	178
Customizing Output .....	180
<b>Sample SAS Output .....</b>	<b>182</b>
Default HTML Output in the SAS Windowing Environment .....	182
Traditional SAS LISTING Output in the SAS Windowing Environment .....	183
<b>The SAS Log .....</b>	<b>184</b>
Structure of the Log .....	184
The SAS Log in Interactive Mode .....	187
The SAS Log in Batch, Line, or Objectserver Modes .....	187
Writing to the Log in All Modes .....	190
Customizing the Log .....	191

---

## Definitions for SAS Output

SAS output is the result of executing SAS programs. Most SAS procedures and some DATA step applications produce output. A SAS program can produce some or all of the following types of output:

### 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. 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

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 190.

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 191 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, see the SAS documentation for your operating environment for specific information about the destination of the SAS console log.

### SAS logging facility output

contain log messages that you create using the SAS logging facility. 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.

The logging facility environment consists of loggers, appenders, and log events. A logger defines a message category, references one or more appenders, and specifies the logger's message level threshold. The message level threshold can be one of the following, from lowest to highest: trace, debug, info, warn, error, or fatal. An appender defines the physical location to write log messages and the format of the message. A log event consists of a log message, a message

threshold, and a logger. Log events are initiated by SAS servers and SAS programs.

When SAS processes a logging facility log event, it compares the message level of the log event to the message threshold of the logger that is named in the log event. If the log event message threshold is the same or higher than the logger's message threshold, the message is written to the locations that are specified by the appenders that are referenced in the logger definition. If the log event is not accepted by the logger, the message is discarded.

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](#).

---

## Routing and Customizing SAS Output

---

### Default Output Destination

#### Definition

The *destination* in SAS is a designation 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. The destination of your output depends on the following:

- your operating environment
- your mode of running SAS
- your version of SAS

#### Default Destinations

In SAS 9.3 and later versions, when running SAS in windowing mode in the Microsoft Windows and UNIX operating environments, output is sent by default to the HTML destination (HTML is the default destination). Also, ODS Graphics is turned on by default in the windowing environment under UNIX and Windows for SAS 9.3 and later versions.

For running SAS in batch mode, however, LISTING is the default destination for SAS 9.4 and earlier versions, and ODS Graphics is turned off by default. See [Table 9.1 on page 178](#) for a comparison of output destinations based on SAS version and operating mode. Your defaults might be different due to your registry or configuration file settings.

The following table shows the default destinations for each method of operation based on SAS version:

**Table 9.1** Comparison of Default Destinations for Output

SAS Version	Mode of Running SAS	Viewer	ODS Destination
SAS 9.3 and later	windowing mode	SAS Results Viewer or browser window	HTML
	interactive line mode	terminal display	LISTING
	noninteractive mode	depends on operating environment	
	batch mode	depends on operating environment:	
SAS 9.2	windowing mode	SAS Output Window	LISTING
	interactive line mode	terminal display	LISTING
	noninteractive mode	depends on operating environment	
	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 the 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](#).

---

## Changing the Output Destination

### Overview

With SAS, there are many ways to control where your log, procedure, and DATA step output is sent. The method that you use depends on your operating system and the mode in which you are running SAS. See [Table 9.2 on page 179](#) for a list of commonly used methods for changing the output destination. You can route your output directly to a PC or terminal display, to a printer, or to an external file. Output destinations can be specified using SAS procedures, system options, commands, statements, or global ODS statements.

## Using ODS to Change the Output Destination

Before ODS was introduced in SAS 7, most procedures generated output that was designed for a traditional line-printer and the output went straight to the listing window as a simple list report. If you wanted to change the destination of your output, you used methods such as PROC PRINTTO and the FILENAME statement. Even though these methods are useful, there are many more options available with ODS for controlling output. ODS destination statements enable you to specify a variety of formats and destinations.

The following list describes some of the commonly used ODS statements and other SAS language elements that are used for routing output.

**Table 9.2** *Changing the Output Destination*

Method to Use	Output Result
PRINTTO procedure	routes DATA step, log, or procedure output from the system default destinations to the destination that you choose. The PRINTTO procedure defines non-ODS destinations.
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 and 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.

Method to Use	Output Result
<a href="#">SAS System Options</a>	<p>redefine the destination of log and output for an entire SAS program. These system options are specified when you invoke SAS. Here are the system options used to route output:</p> <p><a href="#">"ALTLOG System Option: Windows" in SAS Companion for Windows</a></p> <ul style="list-style-type: none"> <li>■ ALTLOG= (<a href="#">Windows, UNIX, z/OS</a>)</li> <li>■ ALTPRINT= (<a href="#">Windows, UNIX, z/OS</a>)</li> <li>■ LOG= (<a href="#">Windows, UNIX, z/OS</a>)</li> <li>■ PRINT= (<a href="#">Windows, UNIX, z/OS</a>)</li> </ul>

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: ["Changing the Default Destination" in SAS Companion for z/OS](#)
- UNIX: ["Changing the Default Routings in UNIX Environments" in SAS Companion for UNIX Environments](#)

---

## Customizing 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 9.3 Methods for Making Output Descriptive**

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 top of each page of output the date and time the SAS job started. When you run SAS in interactive mode, the date and time the job started is the date and time that you started your SAS session.

SAS Language Element	Function
FOOTNOTE statement	prints footnotes at the bottom of each output page. You can also use the FOOTNOTES window for this purpose.
FORMCHAR=	specifies the default output formatting characters for some procedures such as CALENDAR, FREQ, REPORT, and TABULATE.
FORMDLIM=	specifies the default output formatting characters for some procedures such as CALENDAR, FREQ, REPORT, and TABULATE.
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 <a href="#">Base SAS Procedures Guide</a> for information about using the LABEL statement with a specific procedure.
LINESIZE= and PAGESIZE= system options	change 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	control 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 output produced by SAS by using the PAGENO= system option.
global ODS statements	enable you to apply styles to your output or to use a style or table definition.
TITLE statement	prints titles at the top of each output page. By default, SAS prints the following title: 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*.

## Sample SAS Output

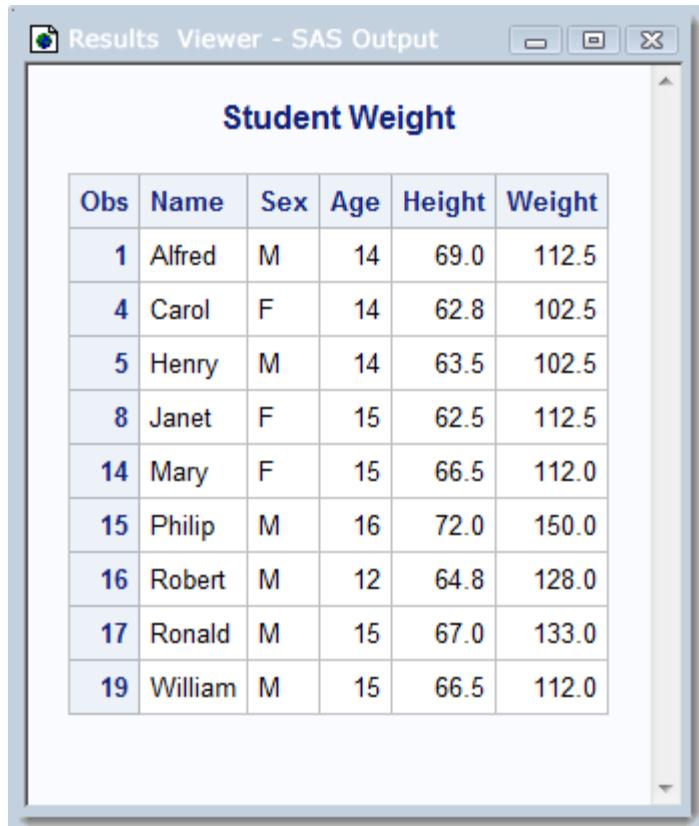
## Default HTML Output in the SAS Windowing Environment

Starting with SAS 9.3, the default destination is HTML when running SAS in the windowing environment for the Windows and UNIX operating environments. The default output is displayed in the SAS Results Viewer Window.

```
title 'Student Weight';
proc print data=sashelp.class;
  where weight>100;
run;
```

```
quit;
```

*Output 9.1 Default HTML Output in the Windowing Environment*



The screenshot shows a Windows application window titled "Results Viewer - SAS Output". The main title of the report is "Student Weight". The data is presented in a tabular format with the following columns: Obs, Name, Sex, Age, Height, and Weight. The data rows are as follows:

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:** At SAS start-up, unless you have previously closed the HTML destination, output is sent to the WORK directory by default. If you close the HTML destination and re-open it in the same SAS session, all output goes to the current directory rather than the WORK directory. You do not have to specify ODS HTML CLOSE; to view your output.

## Traditional SAS LISTING Output in the SAS Windowing Environment

If you are running SAS in windowing mode and want to send your output to the LISTING destination, you can use ODS statements in your SAS programs to change the destination. 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. For information about how to change these settings, see "[Default HTML Output](#)" in *SAS Output Delivery System: User's Guide*.

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;
quit;
ods html;
ods listing close;

```

**Output 9.2** Listing Output in the Windowing Environment

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

See the procedure descriptions in the [Base SAS Procedures Guide](#) for examples of output from SAS procedures. For a discussion and examples of DATA step output, see the “FILE Statement” in [SAS DATA Step Statements: Reference](#) and the “PUT Statement” in [SAS DATA Step Statements: 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. Original program statements are identified by line numbers. SAS messages are interspersed with SAS statements. These messages might begin with the words NOTE, INFO, WARNING, ERROR, or an error number, and they might refer to a SAS statement by its line number in the log.

For example, in the following output, the number 1 is printed to the left of the OPTIONS statement. This means that it is the first line in the program. In interactive mode, SAS continues with the sequence of line numbering until you end your session. If you submit the program again (or submit other programs in your current SAS session), the first program line number is the next consecutive number.

**Operating Environment Information:** The SAS log appears differently depending on your operating environment. See the SAS documentation for your operating environment.

**Example Code 9.1 Sample SAS Log**

```

NOTE: Copyright (c) 2002-2012 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
4     infile
5   ! '\\myserver\my-directory-path\sampleddata.dat'; 7
6     input LastName $ ID $ Gender $ Birth : date7. score1
7     ! score2 score3 score4 score5 score6 score7 score8;
8     format Birth mmddyy8.;

8   run;

NOTE: The infile
      '\\myserver\my-directory-path\sampleddata.dat' is: 8

      Filename=\\myserver\my-directory-path\sampleddata.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\sampleddata.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 system release used to run this program
- 3 name and site number of the computer installation where the program ran
- 4 platform used to run the program
- 5 OPTIONS statement to set a page size of 24, a line size of 64, and to suppress 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 Interactive Mode

In interactive mode, the SAS log is opened when SAS starts. The SAS log is not named until you save it in the active window. The name that you specify must follow the file naming conventions for your operating environment. The SAS log cannot be saved automatically in interactive mode. However, SAS can create a second copy of the SAS log if the ALTLOG= system option is set either at SAS invocation or in a configuration file.

---

## The SAS Log in Batch, Line, or Objectserver Modes

### Overview of 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.

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 perform the following tasks:

- append or replace an existing SAS log
- determine when to write to the SAS log
- start a new SAS log under certain conditions

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](#).

## 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:

**OPEN=APPEND**

appends the SAS log content to the existing SAS log

**OPEN=REPLACE**

replaces the existing SAS log

**OPEN=REPLACEOLD**

replace the existing SAS log if it is older than 24 hours

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

**Overview of Rolling Over the SAS Log:** The SAS log can get very large for long running servers and for batch jobs. By using the LOGPARM= and 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.

**Using Directives to Name the SAS Log:** For the SAS log, a directive is a processing instruction that is used to uniquely name the SAS log. By using directives, you can 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. For example, if you want the SAS log name to include the year, the month, and the day, the LOG= system option might look like this:

```
-log='c:\saslog\#Y#b#dsas.log'
```

When the SAS log is created on February 2, 2009, the name of the log is 2009Feb02sas.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.

For a complete list of directives, see “[LOGPARM= System Option](#)” in *SAS System Options: Reference*.

**Automatically Rolling Over the SAS Log When Directives Change:** 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 follow table 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"
```

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. The log for this SAS session rolls over when the hour changes and when the day changes.

**Table 9.4 Log Names for Rolled Over Logs**

Rollover Time	Log Name
SAS initialization	londonThames0206.log

Rollover Time	Log Name
First rollover	londonThames0207.log
Last log of the day	londonThames0223.log
First log past midnight	londonThames0300.log

**Rolling Over the SAS Log by SAS Session:** To roll over the log at the start of a SAS session, specify the LOGPARM="ROLLOVER=SESSION" option when SAS starts. SAS resolves the system-specific directives by using the system information obtained when SAS starts. No roll over occurs during the SAS session and the log file is closed at the end of the SAS session.

**Rolling Over the SAS Log by the Log Size:** To roll over the log when the log reaches a specific size, specify the LOGPARM="ROLLOVER=*n*" option when SAS starts. *n* is the maximum size of the log, in bytes, and it cannot be smaller than 10K (10,240) bytes. 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 done 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, the log and lock filenames for the second server have a number appended to the names. 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.

**No SAS Log Roll Over:** 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:

### PUT statement

writes selected lines (including text strings and DATA step variable values) to the SAS log in the current iteration of a DATA step. If a FILE statement with the LOG destination executes before a PUT statement, the PUT statement output is directed to a destination that is specified by the FILE statement.

### %PUT statement

enables you to write a text string or macro variable values to the SAS log. %PUT is a SAS macro program statement that is independent of the DATA step and can be used anywhere.

**PUTLOG statement**

writes a user-specified message to the SAS log. Use the PUTLOG statement in a DATA step.

**LIST statement**

writes to the SAS log the input data records for the data line that is being processed. The LIST statement operates only on data that are read with an INPUT statement. It has no effect on data that are read with a SET, MERGE, MODIFY, or UPDATE statement. Use the LIST statement in a DATA step.

**DATA statement with /NESTING option**

writes to the SAS log a note for the beginning and end for each nesting level of DO-END and SELECT-END statements. This enables you to debug mismatched DO-END and SELECT-END statements.

**ERROR statement**

sets the automatic \_ERROR\_ variable to 1 and (OPTIONAL) writes to the log a message that you specify. Use the ERROR statement in a DATA step.

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

### Altering 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. Note that 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:

**CPUID | NOCPUID**

specifies whether hardware information is written to the SAS log.

**ECHO**

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=n**

specifies the maximum number of observations for which data error messages are printed.

**FULLSTATS**

writes expanded statistics to the SAS log. The FULLSTATS system option is valid only under z/OS.

**FULLSTIMER**

writes a subset of system performance statistics to the SAS log.

**ISPNOTES**

specifies whether ISPF error messages are written to the SAS log. The ISPNOTES system option is valid only under the z/OS operating environment.

**HOSTINFOLOG**

writes additional operating environment information to the SAS log when SAS starts.

**LOGPARM "OPEN=APPEND | REPLACE | REPLACEOLD"**

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 REPLACEOLD option specifies to replace logs that are more than one day old.

**MEMRPT**

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**

writes macro nesting execution trace information to the SAS log.

**MPRINT | NOMPRINT**

specifies whether SAS statements that are generated by macro execution are written to the SAS log.

**MSGLEVEL=N | I**

specifies the level of detail in messages that are written to the SAS log. If the MSGLEVEL system option is set to N, the log displays notes, warnings, and error messages only. If MSGLEVEL is set to I, then the log displays additional notes pertaining to index usage, merge processing, HADOOP MapReduce jobs, and sort utilities.

**NEWS=external-file**

specifies whether news information that is maintained at your site is written to the SAS log.

**NOTES | NONOTES**

specifies whether notes (messages beginning with NOTE) are written to the SAS log. NONOTES does not suppress error or warning messages.

**OPLIST**

specifies whether to write to the SAS log the values of all system options that are specified when SAS is invoked.

**OVP | NOOVP**

specifies whether error messages that are printed by SAS are overprinted.

**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**

produces a list of resources that are read during SAS execution 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**

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.

**Operating Environment Information:** See the documentation for your operating environment for other options that affect log output.

## 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.

**DATE system option**

controls whether the date and time that the SAS job began are printed at the top of each page of the SAS log and any output created by SAS.

**DETAILS | NODETAILS**

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 | NODTRESET**

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 the following two options in the FILE statement to customize the log for that report.

**LINESIZE=value**

specifies the maximum number of columns per line for reports and the maximum record length for data files.

**PAGESIZE=value**

specifies the maximum number of lines to be printed on each page of output.

**Note:** FILE statement options apply only to the output specified in the FILE statement, whereas the LINESIZE= and PAGESIZE= SAS system options apply to all subsequent listings.

**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**

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 to a new page in the SAS log and continues printing from there.

**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**

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, VMS, 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.

For more information about how to use these and other SAS system options and statements, see [SAS System Options: Reference](#).

## 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:

**ALTLOG= system option**

specifies the destination for a copy of the SAS log.

**LOG= system option**

specifies the destination for the SAS log when SAS is run in batch mode.

**LOGAPPLNAME**

specifies a SAS session name that can be used for SAS logging.

# 10

## By-Group Processing in SAS Programs

<i>Definition of BY-Group Processing</i> .....	195
<i>References for BY-Group Processing</i> .....	195

### Definition of BY-Group Processing

BY-group processing is a method of processing observations from one or more data sets so that the observations are grouped by common variable values. You can use BY-group processing in both DATA and PROC steps.

The most common use of DATA step BY-group processing is to combine multiple SAS data sets and use the BY statement with one of the following language elements:

- SET statement
- MERGE statement
- MODIFY statement
- UPDATE statement

When you create reports or summaries with SAS procedures, BY-group processing enables you to group information in the output according to values of one or more variables.

### References for BY-Group Processing

- For more information about BY-Group processing, see [Chapter 22, “BY-Group Processing in the DATA Step,” on page 491](#).
- For information about how to use BY-group processing with SAS procedures, see [“BY-Group Processing” in Base SAS Procedures Guide](#) and individual procedures in [Base SAS Procedures Guide](#).
- For information about using BY-group processing to combine information from multiple SAS data sets, see [Chapter 23, “Reading, Combining, and Modifying](#)

["SAS Data Sets," on page 509](#). For even more extensive examples of BY-group processing, see *Combining and Modifying SAS Data Sets: Examples*.

- For information about the BY statement, see Statements in [SAS DATA Step Statements: Reference](#).
- For information about how to use BY-group processing with other software products, see the SAS documentation for those products.

## 11

# WHERE-Expression Processing

<i>Definition of WHERE-Expression Processing</i> .....	197
<i>Where to Use a WHERE Expression</i> .....	198
<i>Syntax of WHERE Expression</i> .....	199
WHERE Expression Contents .....	199
Specifying an Operand .....	199
Specifying an Operator .....	202
<i>Combining Expressions By Using Logical Operators</i> .....	209
Syntax .....	209
Processing Compound Expressions .....	210
Using Parentheses to Control Order of Evaluation .....	210
<i>Improving Performance of WHERE Processing</i> .....	210
<i>Processing a Segment of Data That Is Conditionally Selected</i> .....	211
Applying FIRSTOBS= and OBS= Options .....	211
Applying FIRSTOBS= and OBS= to a Subset of Data .....	212
Processing a SAS View .....	212
<i>Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement</i> ..	214

---

## Definition of WHERE-Expression Processing

### WHERE-expression processing

enables you to conditionally select a subset of observations, so that SAS processes only the observations that meet a set of specified conditions. For example, if you have a SAS data set that contains sales records, you might want to print just the subset of observations for which the sales are greater than \$300,000 but less than \$600,000. In addition, WHERE-expression processing can improve efficiency of a request. For example, if a WHERE expression can be optimized with an index, it is not necessary for SAS to read all observations in the data set in order to perform the request.

### WHERE expression

defines a condition that selected observations must satisfy in order to be processed. You can have a single WHERE expression, referred to as a simple expression, such as the following:

```
where sales gt 600000;
```

Or you can have multiple WHERE expressions, referred to as a compound expression, such as the following:

```
where sales gt 600000 and salary lt 100000;
```

## Where to Use a WHERE Expression

In SAS, you can use a WHERE expression in the following situations:

- WHERE statement in both DATA and PROC steps. For example, the following PRINT procedure includes a WHERE statement so that only the observations where the year is greater than 2001 are printed:

```
proc print data=employees;
  where startdate > '01jan2001'd;
run;
```

- WHERE= data set option. The following PRINT procedure includes the WHERE= data set option:

```
proc print data=employees (where=(startdate > '01jan2001'd));
run;
```

- WHERE clause in the SQL procedure, SCL, and SAS/IML software. For example, the following SQL procedure includes a WHERE clause to select only the states where the murder count is greater than seven:

```
proc sql;
  select state from crime
  where murder > 7;
```

- WHERE command in windowing environments like SAS/FSP software:

```
where age > 15
```

- SAS view (DATA step view, SAS/ACCESS view, PROC SQL view), stored with the definition. For example, the following SQL procedure creates an SQL view named STAT from the data file Crime and defines a WHERE expression for the SQL view definition:

```
proc sql;
  create view stat as
  select * from crime
  where murder > 7;
```

In some cases, you can combine the methods that you use to specify a WHERE expression. That is, you can use a WHERE statement as follows:

- in conjunction with a WHERE= data set option
- along with the WHERE= data set option in windowing procedures, and in conjunction with the WHERE command
- on a SAS view that has a stored WHERE expression

For example, it might be useful to combine methods when you merge data sets. That is, you might want different criteria to apply to each data set when you create a subset of data. However, when you combine methods to create a subset of data, there are some restrictions. For example, in the DATA step, if a WHERE statement and a WHERE= data set option apply to the same data set, the data set option takes precedence. For details, see the documentation for the method that you are using to specify a WHERE expression.

**Note:** By default, a WHERE expression does not evaluate added and modified observations. To specify whether a WHERE expression should evaluate updates, you can specify the WHEREUP= data set option. See the “[WHEREUP= Data Set Option](#)” in *SAS Data Set Options: Reference*.

# Syntax of WHERE Expression

## WHERE Expression Contents

A WHERE expression is a type of SAS expression that defines a condition for selecting observations. A WHERE expression can be as simple as a single variable name or a constant (which is a fixed value). A WHERE expression can be a SAS function, or it can be a sequence of operands and operators that define a condition for selecting observations. In general, the syntax of a WHERE expression is as follows:

**WHERE** *operand* <*operator*> <*operand*>

*operand*

something to be operated on. An operand can be a variable, a SAS function, or a constant. See “[Specifying an Operand](#)” on page 199.

*operator*

a symbol that requests a comparison, logical operation, or arithmetic calculation. All SAS expression operators are valid for a WHERE expression, which include arithmetic, comparison, logical, minimum and maximum, concatenation, parentheses to control order of evaluation, and prefix operators. In addition, you can use special WHERE expression operators. These expression operators include BETWEEN-AND, CONTAINS, IS NULL or IS MISSING, LIKE, sounds-like, and SAME-AND. See “[Specifying an Operator](#)” on page 202.

## Specifying an Operand

### Variable

A variable is a column in a SAS data set. Each SAS variable has attributes like name and type (character or numeric). The variable type determines how you specify the value for which you are searching. For example:

```
where score > 50;
where date >= '01jan2001'd and time >= '9:00't;
where state = 'Texas';
```

In a WHERE expression, you cannot use automatic variables created by the DATA step (for example, FIRST.variable, LAST.variable, \_N\_, or variables created in assignment statements).

As in other SAS expressions, the names of numeric variables can stand alone. SAS treats numeric values of 0 or missing as false; other values as true. In the following example, the WHERE expression returns all rows where EMPNUM is not missing and not zero and ID is not missing and not zero:

```
where empnum and id;
```

The names of character variables can also stand alone. SAS selects observations where the value of the character variable is not blank. For example, the following WHERE expression returns all values not equal to blank:

```
where lastname;
```

## SAS Function

A SAS function returns a value from a computation or system manipulation. Most functions use arguments that you supply, but a few obtain their arguments from the operating environment. To use a SAS function in a WHERE expression, enter its name and arguments enclosed in parentheses. Some functions that you might want to specify include:

- SUBSTR extracts a substring.
- TODAY returns the current date.
- PUT returns a given value using a given format.

The following DATA step produces a SAS data set that contains only observations from data set Customer in which the value of Name begins with Mac and the value of variable City is Charleston OR Atlanta:

```
data testmacs;
  set customer;
  where substr (name,1,3) = 'Mac' and
    (city='Charleston' or city='Atlanta');
run;
```

The OF syntax is permitted in some SAS functions, but it cannot be used when using those functions that are specified in a WHERE clause. In the following DATA step example, OF can be used with RANGE.

```
data abc;
x1=2;
x2=3;
x3=4;
r=range(of x1-x3);
run;
```

When you use the WHERE clause with RANGE and OF, an error is written to the SAS log.

**Example Code 11.1 Output When WHERE Clause Is Used with OF**

```

proc print data=abc;
where range(of x1-x3)=6;
      --
      22
      76
ERROR: Syntax error while parsing WHERE clause.
ERROR 22-322: Syntax error, expecting one of the following: !, !!, &, (, *, **,
+, ',', -, /, <, <=, <>, =, >, >=, ?,
AND, BETWEEN, CONTAINS, EQ, GE, GT, LE, LIKE, LT, NE, OR, ^=, |,
||, ~=.
ERROR 76-322: Syntax error, statement will be ignored.
run;

```

Below is a table of SAS functions that can use the OF syntax:

**Table 11.1 SAS Functions That Use the OF Syntax**

CAT	HARMEANZ	RMS
CATS	KURTOSIS	SKEWNESS
CATT	MAX	STD
CATX	MEAN	STDERR
CSS	MIN	SUM
CV	N	USS
GEOMEAN	NMISS	VAR
GEOMEANZ	ORDINAL	
HARMEAN	RANGE	

**Note:** The SAS functions that are used in a WHERE expression and can be optimized by an index are the SUBSTR function and the TRIM function.

For more information about SAS functions, see [SAS Functions and CALL Routines: Reference](#).

## Constant

A constant is a fixed value such as a number or quoted character string, that is, the value for which you are searching. A constant is a value of a variable obtained from the SAS data set, or values created within the WHERE expression itself. Constants are also called literals. For example, a constant could be a flight number or the name of a city. A constant can also be a time, date, or datetime value.

The value is either numeric or character. Note the following rules regarding whether to use quotation marks:

- If the value is numeric, do not use quotation marks.

```
where price > 200;
```

- If the value is character, use quotation marks.

```
where lastname eq 'Martin';
```

- You can use either single or double quotation marks, but do not mix them. Quoted values must be exact matches, including case.

- It might be necessary to use single quotation marks when double quotation marks appear in the value, or use double quotation marks when single quotation marks appear in the value.

```
where item = '6" decorative pot';
where name ? "D'Amico";
```

- A SAS date constant must be enclosed in quotation marks. When you specify date values, case is not important. You can use single or double quotation marks. The following expressions are equivalent:

```
where birthday = '24sep1975'd;
where birthday = '24sep1975"d;
```

## Specifying an Operator

### Arithmetic Operators

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

*Table 11.2 Arithmetic Operators*

Symbol	Definition	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 with another variable. Comparison operators propose a relationship and ask SAS to determine whether that relationship holds. For example, the following WHERE expression accesses only those observations that have the value 78753 for the numeric variable ZipCode:

```
where zipcode eq 78753;
```

The following table lists the comparison operators:

**Table 11.3** Comparison Operators

Symbol	Mnemonic Equivalent	Definition	Example
=	EQ	equal to	where empnum eq 3374;
^= 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 observations 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.

## IN Operator

The IN operator, which is a comparison operator, searches for character and numeric values that are equal to one from a list of values. The list of values must be in parentheses, with each character value in quotation marks and separated by either a comma or blank.

For example, suppose you want all sites that are in North Carolina or Texas. You could specify:

```
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);`

## Fully Bounded Range Condition

A fully bounded range condition consists of a variable between two comparison operators, specifying both an upper and lower limit. For example, the following expression returns the employee numbers that fall within the range of 500 to 1000 (inclusive):

```
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 observations 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 observations in which the value of a variable falls within an inclusive range of values.

You can specify the limits of the range as constants or expressions. Any range that you specify is an inclusive range, so that a value equal to one of the limits of the range is within the range. The general syntax for using BETWEEN-AND is as follows:

`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 observations 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 observations by searching for a specified set of characters within the values of a character variable. The position of the string within the variable's values does not matter. However, the operator is case sensitive when making comparisons.

The following examples select observations having the values `Mobay` and `Brisbayne` for the variable `Company`, but they do not select observations containing `Bayview`:

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

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

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

You can also use the CONTAINS operator with two variables, that is, to determine whether one variable is contained in another. When you specify two variables, keep in mind the possibility of trailing spaces, which can be resolved using the TRIM function.

```
proc sql;
  select *
  from table1 as a, table2 as b
  where a.fullname contains trim(b.lastname) and
    a.fullname contains trim(b.firstname);
```

In addition, the TRIM function is helpful when you search on a macro variable.

```
proc print;
  where fullname contains trim("&lname");
run;
```

## IS NULL or IS MISSING Operator

The IS NULL or IS MISSING operator selects observations in which the value of a variable is missing. The operator selects observations with both regular or special missing value characters and can be used for both character and numeric variables.

```
where idnum is missing;
where name is null;
```

The following are equivalent for character data:

```
where name is null;
where name = ' ';
```

And the following is equivalent for numeric data. This statement differentiates missing values with special missing value characters:

```
where idnum <= .Z;
```

You can combine the NOT logical operator with IS NULL or IS MISSING to select nonmissing values, as follows:

```
where salary is not missing;
```

## LIKE Operator

The LIKE operator selects observations by comparing the values of a character variable to a specified pattern, which is referred to as pattern matching. The LIKE operator is case sensitive. There are two special characters available for specifying a pattern:

percent sign (%)

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

```
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:

Pattern	Name Selected
like 'D_an'	Dyan
like 'D_an_'	Diana, Diane
like 'D_an__'	Dianna
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%';
```

Because the % and \_ characters have special meaning for the LIKE operator, you must use an escape character when searching for the % and \_ characters in values. An escape character is a single character that, in a sequence of characters, signifies that what follows takes an alternative meaning. For the LIKE operator, an escape character signifies to search for literal instances of the % and \_ characters in the variable's values instead of performing the special-character function.

For example, if the variable X contains the values abc, a\_b, and axb, the following LIKE operator with an escape character selects only the value a\_b. The escape character (/) specifies that the pattern searches for a literal '\_' that is surrounded by the characters a and b. The escape character (/) is not part of the search.

```
where x like 'a/_b' escape '/';
```

Without an escape character, the following LIKE operator would select the values a\_b and axb. The special character underscore in the search pattern matches any single b character, including the value with the underscore:

```
where x like 'a_b';
```

To specify an escape character, include the character in the pattern-matching expression, and then the keyword ESCAPE followed by the escape-character expression. When you include an escape character, the pattern-matching expression must be enclosed in quotation marks, and it cannot contain a column name. The escape-character expression evaluates to a single character. The operands must be character or string literals. If it is a single character, enclose it in quotation marks.

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

## Sounds-like Operator

The sounds-like ( =\*) operator selects observations that contain a spelling variation of a specified word or words. The operator uses the Soundex algorithm to compare the variable value and the operand. For more information, see the SOUNDEX function in [SAS Functions and CALL Routines: Reference](#).

**Note:** Note that the SOUNDEX algorithm is English-biased, and is less useful for languages other than English.

Although the sounds-like operator is useful, it does not always select all possible values. For example, consider that you want to select observations from the following list of names that sound like Smith:

- Schmitt
- Smith
- Smithson
- Smitt
- Smythe

The following WHERE expression selects all the names from this list except Smithson:

```
where lastname=*& 'Smith';
```

You can combine the NOT logical operator with the sounds-like operator to select values that do not contain a spelling variation of a specified word or words, such as:

```
where lastname not =* 'Smith';
```

**Note:** The sounds-like operator cannot be optimized with an index.

## SAME-AND Operator

Use the SAME-AND operator to add more conditions to an existing WHERE expression later in the program without retyping the original conditions. This capability is useful with the following:

- interactive SAS procedures
- full-screen SAS procedures that enable you to enter a WHERE expression on the command line
- any type of RUN-group processing

Use the SAME-AND operator when you already have a WHERE expression defined and you want to insert additional conditions. The SAME-AND operator has the following form:

- where-expression-1;
- ... SAS statements...
- WHERE SAME AND where-expression-2;
- ... SAS statements...
- WHERE SAME AND where-expression-n;

SAS selects observations that satisfy the conditions after the SAME-AND operator in addition to any previously defined conditions. SAS treats all of the existing conditions as if they were conditions separated by AND operators in a single WHERE expression.

The following example shows how to use the SAME-AND operator within RUN groups in the GPLOT procedure. The SAS data set YEARS has three variables and contains quarterly data for the 2009–2011 period:

```
proc gplot data=years;
  plot unit*quar=year;
run;

  where year > '01jan2009'd;
run;

  where same and year < '01jan2012'd;
run;
```

The following WHERE expression is equivalent to the preceding code:

```
where year > '01jan2009'd and year < '01jan2012'd;
```

## MIN and MAX Operators

Use the MIN or MAX operators to find the minimum or maximum value of two quantities. Surround the operators with the two quantities whose minimum or maximum value you want to know.

- The MIN operator returns the lower of the two values.
- The MAX operator returns the higher of two values.

For example, if A is less than B, then the following would return the value of A:

```
where x = (a min b);
```

**Note:** The symbol representation >< is not supported, and <> is interpreted as “not equal to.”

## Concatenation Operator

The concatenation operator concatenates character values. You indicate the concatenation operator as follows:

- || (two OR symbols)
- !! (two exclamation marks)

- `||` (two broken vertical bars).

For example:

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

## Prefix Operators

The plus sign (+) and minus sign (–) can be either prefix operators or arithmetic operators. They are prefix operators when they appear at the beginning of an expression or immediately preceding an open parenthesis. A prefix operator is applied to the variable, constant, SAS function, or parenthetic expression.

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

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

# Combining Expressions By Using Logical Operators

## Syntax

You can combine or modify WHERE expressions by using the logical operators (also called Boolean operators) AND, OR, and NOT. The basic syntax of a compound WHERE expression is as follows:

**WHERE** *where-expression-1 AND | OR | NOT where-expression-n*

AND combines two conditions by finding observations that satisfy both conditions.  
For example:

```
where skill eq 'java' and years eq 4;
```

OR combines two conditions by finding observations that satisfy either condition or both. For example:

```
where skill eq 'java' or years eq 4;
```

NOT modifies a condition by finding the complement of the specified criteria. You can use the NOT logical operator in combination with any SAS and WHERE expression operator. And you can combine the NOT operator with AND and OR. For example:

```
where skill not eq 'java' or years not eq 4;
```

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

**Table 11.4** Logical (Boolean) Operators

Symbol	Mnemonic Equivalent
&	AND

Symbol	Mnemonic Equivalent
! or   or	OR
^ or ~ or $\neg$	NOT

## Processing Compound Expressions

When SAS encounters a compound WHERE expression (multiple conditions), the software follows rules to determine the order in which to evaluate each expression. When WHERE expressions are combined, SAS processes the conditions in a specific order:

- 1 The NOT expression is processed first.
- 2 Then the expressions joined by AND are processed.
- 3 Finally, the expressions joined by OR are processed.

## Using Parentheses to Control Order of Evaluation

Even though SAS evaluates logical operators in a specific order, you can control the order of evaluation by nesting expressions in parentheses. That is, an expression enclosed in parentheses is processed before one not enclosed. The expression within the innermost set of parentheses is processed first, followed by the next deepest, moving outward until all parentheses have been processed.

For example, suppose you want a list of all the Canadian sites that have both SAS/GRAFH and SAS/STAT software, so you issue the following expression:

```
where product='GRAPH' or product='STAT' and country='Canada';
```

The result, however, includes all sites that license SAS/GRAFH software along with the Canadian sites that license SAS/STAT software. To obtain the correct results, you can use parentheses, which causes SAS to evaluate the comparisons within the parentheses first, providing a list of sites with either product licenses, then the result is used for the remaining condition:

```
where (product='GRAPH' or product='STAT') and country='Canada';
```

## Improving Performance of WHERE Processing

Indexing a SAS data set can significantly improve the performance of WHERE processing. An index is an optional file that you can create for SAS data files in order to provide direct access to specific observations.

Processing a WHERE expression without an index requires SAS to sequentially read observations in order to find the ones that match the selection criteria. Without an index, SAS first checks for the sort indicator, which is stored with the data file from a previous SORT procedure or SORTEDBY= data set option. If the sort indicator is validated, SAS takes advantage of it and stops reading the file once it is clear there are no more values that satisfy the WHERE expression. For example, consider a data set that is sorted by Age, without an index. To process the expression `where age > 25`, SAS stops reading observations after it finds an observation that is greater than 25. Note that while SAS can determine when to stop reading observations, without an index, there is no indication where to begin, so SAS always begins with the first observation, which can require reading a lot of observations.

Having an index enables SAS to determine which observations satisfy the criteria, which is referred to as optimizing the WHERE expression. However, by default, SAS decides whether to use the index or read the entire data set sequentially. For details about how SAS uses an index to process a WHERE expression, see “[Using an Index for WHERE Processing](#)” on page 702.

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

*Table 11.5 Constructing Efficient WHERE Expressions*

Guideline	Efficient	Inefficient
Avoid using the LIKE operator that begins with % or _.	<code>where country like 'A%INA';</code>	<code>where country like '%INA';</code>
Avoid using arithmetic expressions.	<code>where salary &gt; 48000;</code>	<code>where salary &gt; 12*4000;</code>

---

## Processing a Segment of Data That Is Conditionally Selected

---

### Applying FIRSTOBS= and OBS= Options

When you conditionally select a subset of observations with a WHERE expression, you can also segment that subset by applying FIRSTOBS=, OBS=, or both processing (as data set options and system options). When used with a WHERE expression,

- FIRSTOBS= specifies the observation number within the subset of data selected by the WHERE expression to begin processing.
- OBS= specifies when to stop processing observations from the subset of data selected by the WHERE expression.

When used with a WHERE expression, the values specified for OBS= and FIRSTOBS= are not the physical observation number in the data set, but a logical number in the subset. For example, obs=3 does not mean the third observation number in the data set. Instead, it means the third observation in the subset of data selected by the WHERE expression.

Applying OBS= and FIRSTOBS= processing to a subset of data is supported for the WHERE statement, WHERE= data set option, and WHERE clause in the SQL procedure.

If you are processing a SAS view that is a view of another view (nested views), applying OBS= and FIRSTOBS= to a subset of data could produce unexpected results. For nested views, OBS= and FIRSTOBS= processing is applied to each SAS view, starting with the root (lowest-level) view, and then filtering observations for each SAS view. The result could be that no observations meet the subset and segment criteria. See “[Processing a SAS View](#)” on page 212.

## Applying FIRSTOBS= and OBS= to a Subset of Data

The following SAS program illustrates how to specify a condition to subset data, and how to specify a segment of the subset of data to process.

```
data A; 1
do I=1 to 100;
  X=I + 1;
  output;
end;
run;

proc print data=work.a (firstobs=2 3 obs=4; 4
  where I > 90; 2
run;
```

- 1** The DATA step creates a data set named Work.A containing 100 observations and two variables: I and X.
- 2** The WHERE expression I > 90 tells SAS to process only the observations that meet the specified condition, which results in the subset of observations 91 through 100.
- 3** The FIRSTOBS= data set option tells SAS to begin processing with the 2nd observation in the subset of data, which is observation 92.
- 4** The OBS= data set option tells SAS to stop processing when it reaches the 4th observation in the subset of data, which is observation 94.

The result of PROC PRINT is observations 92, 93, and 94.

## Processing a SAS View

The following SAS program creates a data set, a SAS view for the data set, then a second SAS view that subsets data from the first SAS view. Both a WHERE statement and the OBS= system option are used.

```
data a; 1
do I=1 to 100;
```

```

X=I + 1;
output;
end;
run;

data viewa/view=viewa; 2

set a;
Z = X+1;
run;

data viewb/view=viewb; 3

set viewa;
where I > 90;
run;

options obs=3; 4

proc print data=work.viewb; 5

run;

```

- 1** The first DATA step creates a data set named Work.A, which contains 100 observations and two variables: I and X.
- 2** The second DATA step creates a SAS view named Work.ViewA containing 100 observations and three variables: I, X (from data set Work.A), and Z (assigned in this DATA step).
- 3** The third DATA step creates a SAS view named Work.ViewB and subsets the data with a WHERE statement, which results in the view accessing ten observations.
- 4** The OBS= system option applies to the previous SET ViewA statement, which tells SAS to stop processing when it reaches the 3rd observation in the subset of data being processed.
- 5** When SAS processes the PRINT procedure, the following occurs:
  - 1** First, SAS applies *obs=3* to Work.ViewA, which stops processing at the 3rd observation.
  - 2** Next, SAS applies the condition *I > 90* to the three observations being processed. None of the observations meet the criteria.
  - 3** PROC PRINT results in no observations.

To prevent the potential of unexpected results, you can specify *obs=max* when creating Work.ViewA to force SAS to read all the observations in the root (lowest-level) view:

```

data viewa/view=viewa;
  set a (obs=max);
  Z = X+1;
run;

```

The PRINT procedure processes observations 91, 92, and 93.

# Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement

To conditionally select observations from a SAS data set, you can use either a WHERE expression or a subsetting IF statement. They both test a condition to determine whether SAS should process an observation. However, they differ as follows:

- The subsetting IF statement can be used only in a DATA step. A subsetting IF statement tests the condition after an observation is read into the Program Data Vector (PDV). If the condition is true, SAS continues processing the current observation. Otherwise, the observation is discarded, and processing continues with the next observation.
- You can use a WHERE expression in both a DATA step and SAS procedures, as well as in a windowing environment, SCL programs, and as a data set option. A WHERE expression tests the condition before an observation is read into the PDV. If the condition is true, the observation is read into the PDV and processed. If the condition is false, the observation is not read into the PDV, and processing continues with the next observation. This can yield substantial savings when observations contain many variables or very long character variables (up to 32K bytes). In addition, a WHERE expression can be optimized with an index, and the WHERE expression enables more operators, such as LIKE and CONTAINS.

**Note:** Although it is generally more efficient to use a WHERE expression and avoid the move to the PDV before processing, if the data set contains observations with very few variables, the move to the PDV could be cheap. However, one variable containing 32K bytes of character data is not cheap, even though it is only one variable.

In most cases, you can use either method. However, the following table provides a list of tasks that require you to use a specific method:

**Table 11.6 Tasks Requiring Either WHERE Expression or Subsetting IF Statement**

Task	Method
Make the selection in a procedure without using a preceding DATA step	WHERE expression
Take advantage of the efficiency available with an indexed data set	WHERE expression
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 expression
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

Task	Method
Make the selection at some point during a DATA step rather than at the beginning	subsetting IF
Execute the selection conditionally	subsetting IF



# 12

## Optimizing System Performance

<i>Definitions for Optimizing System Performance</i> .....	217
<i>Collecting and Interpreting Performance Statistics</i> .....	218
Using the FULLTIMER and TIMER System Options .....	218
Interpreting FULLTIMER and TIMER Statistics .....	219
<i>Techniques for Optimizing I/O</i> .....	219
Overview of Techniques for Optimizing I/O .....	219
Using WHERE Processing .....	220
Using DROP and KEEP Statements .....	221
Using LENGTH Statements .....	221
Using the OBS= and FIRSTOBS= Data Set Options .....	221
Creating SAS Data Sets .....	221
Using Indexes .....	222
Accessing Data through SAS Views .....	222
Using Engines Efficiently .....	222
Setting System Options to Improve I/O Performance .....	223
Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views .....	225
Using the SASFILE Statement .....	225
Using the DATASETS Procedure to Modify Attributes .....	226
Storing Variables as Characters .....	226
<i>Techniques for Optimizing Memory Usage</i> .....	226
System Options .....	226
Using the BY Statement with PROC MEANS .....	227
<i>Techniques for Optimizing CPU Performance</i> .....	227
Reducing CPU Time By Using More Memory or Reducing I/O .....	227
Storing a Compiled Program for Computation-Intensive DATA Steps .....	227
Reducing Search Time for SAS Executable Files .....	228
Specifying Variable Lengths .....	228
Using Parallel Processing .....	228
Reducing CPU Time By Modifying Program Compilation Optimization .....	229
<i>Calculating Data Set Size</i> .....	229

## 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 12.1** 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 12.2** 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 ALIGN=, 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 “[Deciding Whether to Use a WHERE Expression or a Subsetting IF Statement](#)” on page 214.

---

## 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 [Chapter 35, “Cross-Release Compatibility and Migration,”](#) on page 779 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 “[Understanding SAS Indexes](#)” on page 692 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 29, “SAS Views,” on page 721](#) 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 225.

## 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 12.3 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 37, “[SAS Engines](#),” on page 803 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 229](#) 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 225.

## 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 VBUFSIZE= 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 SASSHARE 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 229. 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 [Chapter 30, “Stored Compiled DATA Step Programs,” on page 731](#).

---

## 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 13, “Support for Parallel Processing,” on page 231](#).

---

## 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.



# Support for Parallel Processing

<i>Overview</i> .....	231
<i>What Is Threading Technology in SAS?</i> .....	232
<i>How Is Threading Controlled in SAS?</i> .....	233
<i>Threading in Base SAS</i> .....	233
<i>SAS/ACCESS Engines</i> .....	236
<i>SAS Scalable Performance Data Server</i> .....	237
<i>SAS Intelligence Platform</i> .....	237
<i>SAS High-Performance Analytics Portfolio of Products</i> .....	238
<i>SAS Grid Manager</i> .....	239
<i>SAS In-Database Technology</i> .....	240
<i>SAS In-Memory Analytics Technology</i> .....	240
<i>SAS High-Performance Analytics Product Integration</i> .....	242
<i>SAS Viya</i> .....	244

---

## 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?](#)” on page 433 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?](#)” on page 433 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 THREADMIN and THREADMAX 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 238.)

## 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, the execution times are much shorter than if the data were transferred to the SAS server.

In-Database technology uses existing database functionality and standard SAS functionality to extend that SAS functionality into the database. For example, Base SAS procedures such as SORT or SUMMARY can be executed within the database environment with the inclusion of simple options to the procedure's code. These procedures are "translated" into native database functions for execution within the database environment. As an example of extending SAS functionality into the database, scoring code generated by SAS Enterprise Miner can be exported as functions. Once exported, these functions can be executed by either SAS processes or called from third-party or database-specific applications using standard SQL statements.

SAS In-Database technology is described in the *SAS In-Database Products: Administrators Guide* and *SAS In-Database Products: User's Guide*. SAS In-Database components include scoring and analytic accelerators for most of the supported databases.

---

## SAS In-Memory Analytics Technology

SAS In-Memory Analytics technology takes advantage of the large number of threads and high level of memory that is available in some specially configured DBMS appliances such as Teradata and EMC Greenplum and on commodity hardware using *Hadoop Distributed File System* (HDFS). The Teradata and EMC Greenplum appliances are assembled as computing clusters using specific massively parallel processing (MPP) techniques. With SAS In-Memory Analytics technology, all of the data to be processed is distributed across the cluster and loaded into memory before the analytic procedure begins. This is in distinct contrast to traditional processing where data is loaded in blocks as they are needed. In addition, SAS High-Performance Analytics procedures are engineered to execute on hundreds of threads and each thread is responsible for a small subset of the overall data to be processed. Faster analysis of large data sets results in greater refinement of analytic models.

Several members of the SAS High-Performance Analytics family of products are based on SAS In-Memory Analytics technology, including the SAS High-Performance Analytics Server, SAS Visual Analytics, and SAS High-Performance Risk. For more detail on the array of SAS In-Memory Analytics components, please see the In-Memory Analytics website: [Products & Solutions/In-Memory Analytics](#).

#### SAS High-Performance Analytics Server

The SAS High-Performance Analytics Server is engineered to run in threads and provides high-performance analytic procedures that focus on predictive model development with computationally intensive calculations. These procedures are drawn from the libraries of SAS/STAT, SAS/QC, and SAS/ETS. The procedures execute in the MPP computing environment provided by EMC Greenplum and Teradata appliances and Hadoop clusters. High-performance MPP configurations typically have a minimum 1.5TB of memory and upward of 192 cores with multiple threads per core.

The SAS High-Performance Analytics procedures are invoked on the requesting SAS client where a Base SAS session is executing. This can be the traditional Display Manager System, SAS Enterprise Guide, or through the SAS High-Performance Data Mining tab in SAS Enterprise Miner. In the MPP environment, the SAS client communicates with the SAS High-Performance Analytics Server nodes where a thin SAS environment executes a copy of the requested SAS procedure or DS2 code. Once completed, the analytic results are returned to the requesting application on the SAS client.

The PERFORMANCE statement in the SAS High-Performance Analytics procedures enables you to specify parameters to control threading and the mode of processing, SMP (client mode) or MPP (distributed mode). In SMP mode (which is a SAS session on the client machine), the CPUCOUNT default is the number of CPUs on the client machine. CPUCOUNT and NOTHREADS options can override the SAS system options. In this environment, if the procedure executes in MPP mode, then the CPUCOUNT option default is that the number of threads is determined by the number of CPUs on the appliance nodes. The NTHREADS option available only in the PERFORMANCE statement throttles the number of threads.

The SAS High-Performance Analytics Server and procedures are documented in the [SAS High-Performance Server Administration Guide](#). The SAS High-Performance Analytics Server relies on the SAS LASR Analytic Server to provide a highly scalable and reliable analytics infrastructure that is optimized for large volumes of data and complex computations.

#### SAS Visual Analytics

SAS Visual Analytics runs on SAS High-Performance Analytics Server and on a SAS Intelligence Platform if it is configured as an MPP environment. SAS Visual Analytics provides the ability for business users and business analysts to perform a wide variety of tasks using visualizations specifically designed for exploring big data and deriving value. This enables you to go beyond descriptive statistics and use more specialized analytics in a very scalable way. This provides more accurate insights into the future and aids decision making. For example, users can visually explore data, execute analytic correlations on billions of rows of data in just seconds, and visually present results. This helps quickly identify patterns, trends, and relationships in data that were not evident before.

SAS Visual Analytics can be combined with the SAS High-Performance Analytics Server and SAS In-Database to provide a high-performance model lifecycle. For example, use SAS Visual Analytics to explore the data and decide which information to use. Use SAS High-Performance Analytics Server to build

your models, and then use SAS In-Database to push the models into an appropriate database for scoring.

SAS Visual Analytics requires a dedicated and specialized configuration of blade hardware such as Teradata or EMC Greenplum appliances or Hadoop HDFS configured as an MPP cluster. This environment is always threaded. SAS options CPUCOUNT and NOTHREADS have no effect. Instead, the NTHREADS option in the PERFORMANCE statement provides a way to throttle thread usage. See the *SAS Visual Analytics: User's Guide* for product information.

SAS Visual Analytics relies on the SAS LASR Analytic Server to provide a highly scalable analytics infrastructure that is optimized for large volumes of data and complex computations.

#### SAS LASR Analytic Server

The SAS LASR Analytic Server is an analytic platform that provides a secure environment for concurrent access to data. It loads the data into memory across the computing nodes of a SAS High-Performance Analytics Server. The SAS LASR Analytic Server executes on the SAS High-Performance Analytics Server root node with worker nodes across the appliance that read data into memory in parallel very fast. If the data is not from a *co-located data provider*, then the data is read from the DBMS appliance or Hadoop cluster and transferred to the root node of the SAS High-Performance Analytics Server. Then, it is loaded into the memory of the *worker nodes*. The SAS LASR Analytic Server is not influenced by CPUCOUNT or NOTHREADS. Instead, the NTHREADS option in the PERFORMANCE statement throttles thread usage. Refer to the *SAS LASR Analytic Server: Administration Guide* for details.

For more SAS In-Memory Analytics products, see [Products & Solutions/In-Memory Analytics](#).

## SAS High-Performance Analytics Product Integration

The SAS High-Performance Analytics portfolio supports many SAS solutions and products. Some SAS products are integrated with SAS Grid Manager and (to some extent) other high-performance analytics products in order to take advantage of *parallel processing*.

#### SAS Enterprise Guide:

SAS Enterprise Guide executes as a SAS client and provides a front end to SAS servers to execute SAS programs. Users create programs that take advantage of the parallel processing capabilities in Base SAS (for example, thread-enabled procedures and stored processes). SAS Enterprise Guide can detect whether SAS Grid Manager is managing the environment to provide workload balancing, resource assignment, and job prioritization. SAS Enterprise Guide can run Process Flow branches in parallel on different grid nodes. It enables parallel execution of tasks on the same server, and you can run tasks at the project or individual level in a SAS Grid Manager environment. See the *SAS Enterprise Guide* for details.

#### SAS Data Integration Studio

SAS Data Integration Studio runs on the SAS Data Integration Server and automatically takes advantage of grid computing if SAS Grid Manager is

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. NOTREADS and CPUCOUNT options have no effect. 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 THREADES as the default. NOTREADS 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](#).

# The SAS Registry

---

<b>Introduction to the SAS Registry .....</b>	<b>245</b>
What Is the SAS Registry? .....	245
Who Should Use the SAS Registry? .....	246
Where the SAS Registry Is Stored .....	246
How Do I Display the SAS Registry? .....	246
Definitions for the SAS Registry .....	247
<b>Managing the SAS Registry .....</b>	<b>248</b>
Primary Concerns about Managing the SAS Registry .....	248
Backing Up the Sasuser Registry .....	248
Recovering from Registry Failure .....	250
Using the SAS Registry to Control Color .....	252
Using the Registry Editor .....	253
<b>Configuring Your Registry .....</b>	<b>258</b>
Configuring Universal Printing .....	258
Configuring SAS Explorer .....	258
Configuring Libraries and File Shortcuts with the SAS Registry .....	259
Fixing Library Reference (Libref) Problems with the SAS Registry .....	260

---

## 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 registry.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 registry.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**  $\Rightarrow$  **Accessories**  $\Rightarrow$  **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 14.1** Section of the Registry Editor Showing Value Names and Value Data for the Subkey 'HTML'

Contents of 'HTML'	
Name	Data
ab coco	"D2,71,1E"
oioi 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<sup>1</sup> 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

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.

- modifying printer settings from the default printer settings that your system administrator provides for you
- changing localization settings
- altering translation tables with TRANTAB

## When SAS Resets to the Default Settings

When SAS starts up, it automatically scans the registry file. SAS restores the registry to its original settings under two conditions:

- If SAS detects that the registry is corrupted, then SAS rebuilds the file.
- If you delete the registry file called registry.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 registry.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**  $\Rightarrow$  **Explorer**.
- 2 Select **Tools**  $\Rightarrow$  **Options**  $\Rightarrow$  **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 `Registry.Itemstor` file.
  - 8 Select **Copy** from the pop-up menu and copy the `Registry` file. SAS assigns the name `Registry_copy` to the file.

**Operating Environment Information:** You can also use a copy command from your operating environment to make a copy of your registry file for backup purposes. When viewed from outside SAS Explorer, the filename is `registry.sas7bitm`. Under z/OS, you cannot use the environment copy command to copy your registry file unless your Sasuser library is assigned to an HFS directory.

## Using the Registry Editor to Back Up the SAS Registry

Using the Registry Editor to back up the SAS registry is generally the preferred backup method, because it retains any new keys or values in case you must restore the registry from the backup.

To use the Registry Editor to back up the SAS Registry:

- 1 Open the Registry Editor with the REGEDIT command.
- 2 Select the top-level key in the left pane of the registry window.
- 3 From the Registry Editor, select **File**  $\Rightarrow$  **Export Registry File**.  
A Save As window appears.
- 4 Enter a name for your registry backup file in the filename field. (SAS applies the proper file extension name for your operating system.)
- 5 Click **Save**.

This saves the registry backup file in Sasuser. You can control the location of your registry backup file by specifying a different location in the Save As window.

## Recovering from Registry Failure

This section gives instructions for restoring the registry with a backup file, and shows you how to repair a corrupt registry file.

To install the registry backup file that was created using SAS Explorer or an operating system copy command:

- 1 Change the name of your corrupt registry file to something else.

- 2 Rename your backup file to *registry.sas7bitm*, which is the name of your registry file.
- 3 Copy your renamed registry file to the Sasuser location where your previous registry file was located.
- 4 Restart your SAS session.

To restore a registry backup file created with the Registry Editor:

- 1 Open the Registry Editor with the REGEDIT command.
- 2 Select **File**  $\Rightarrow$  **Import Registry File**.
- 3 Select the registry file that you previously exported.
- 4 Click **Open**.
- 5 Restart SAS.

To restore a registry backup file created with PROC REGISTRY:

- 1 Open the Program editor and submit the following program to import the registry file that you created previously.

```
proc registry import=<registry file specification>;  
run;
```

This imports the registry file to the Sasuser library.

- 2 If the file is not already properly named, then use Explorer to rename the registry file to *registry.sas7bitm*:
- 3 Restart SAS.

To attempt to repair a damaged registry:

- 1 Rename the damaged registry file to something other than “registry” (for example, *temp*).
- 2 Start your SAS session.
- 3 Define a library pointing to the location of the *temp* registry.

```
libname here '..'
```

- 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**  $\Rightarrow$  **Accessories**  $\Rightarrow$  **Registry Editor**  $\Rightarrow$  **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**  $\Rightarrow$  **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**  $\Rightarrow$  **Accessories**  $\Rightarrow$  **Registry Editor**.

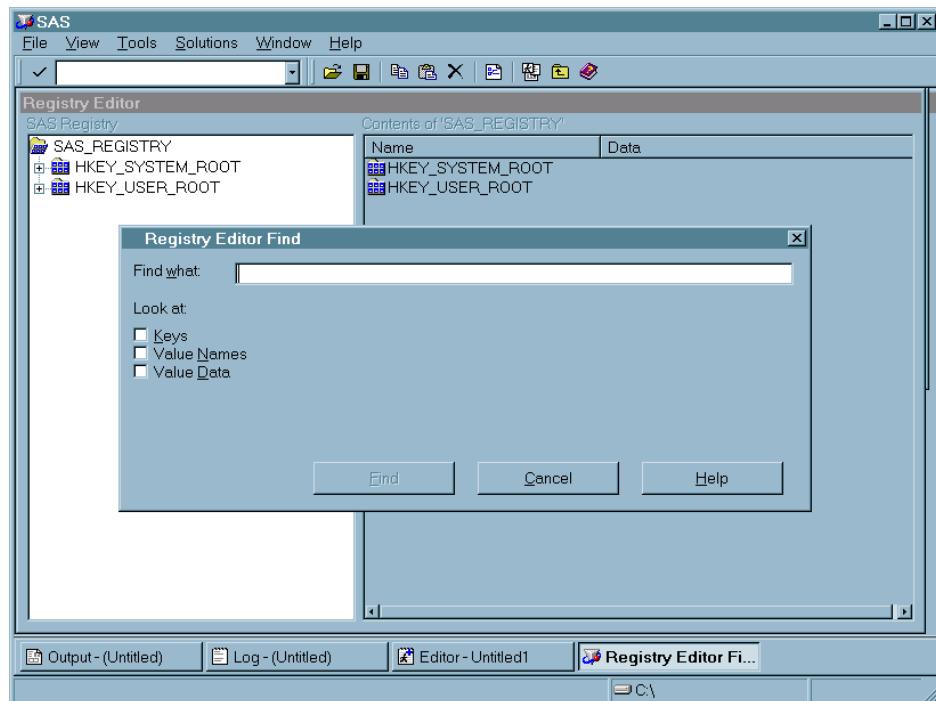
### Finding Specific Data in the Registry

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**  $\Rightarrow$  **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 14.2 The Registry Editor Find Utility

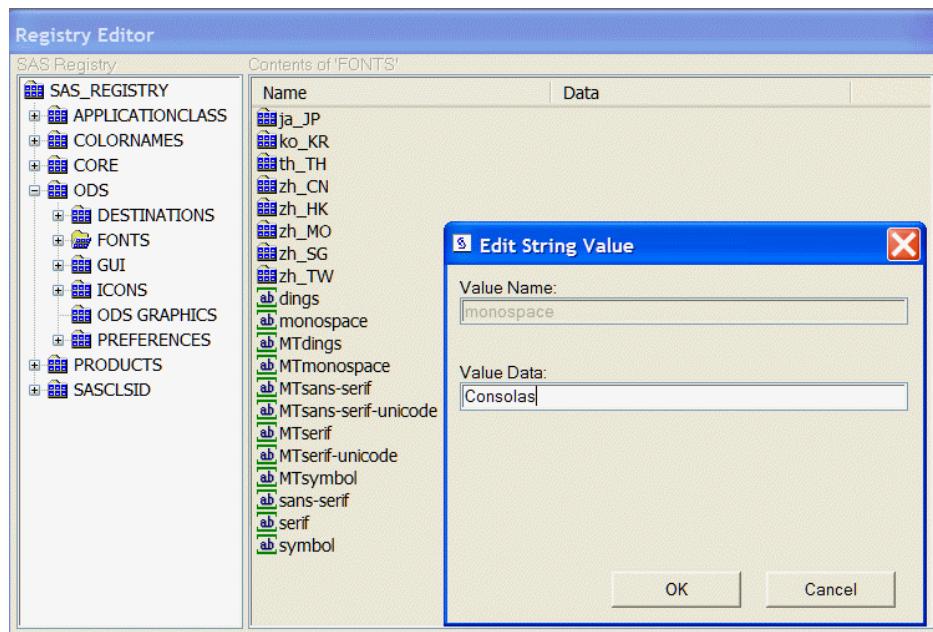


## Changing a Value in the SAS Registry

**CAUTION!** Before modifying registry values, always back up the `registry.sas7bitm` file from `Sasuser`.

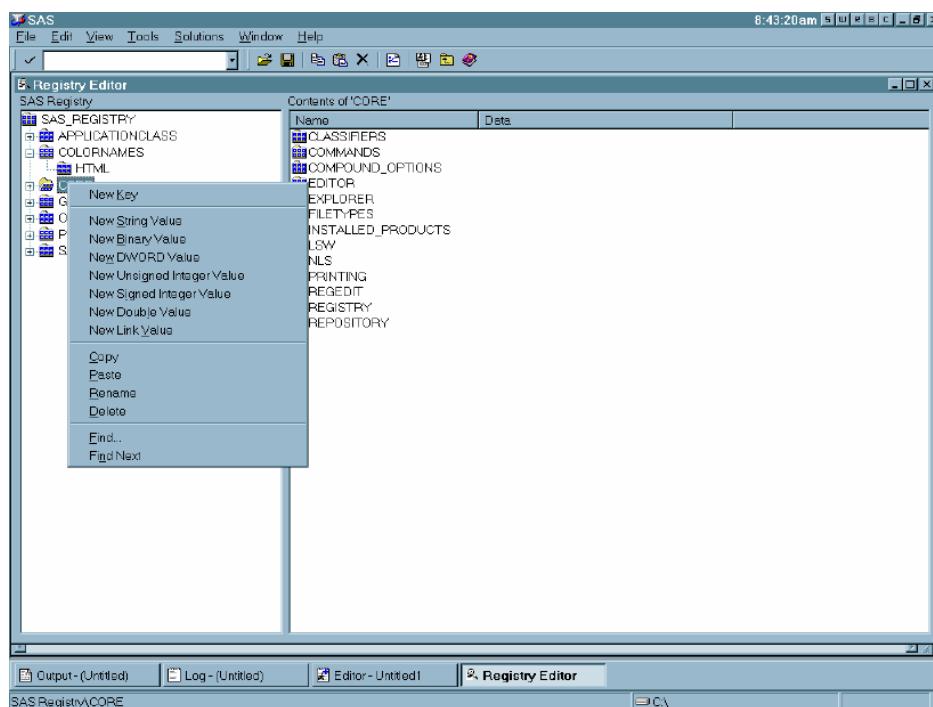
- 1 In the left pane of the Registry Editor window, click the key that you want to change. The values contained in the key appear in the right pane.
- 2 Double-click the value.

The Registry Editor displays several types of windows, depending on the type of value that you are changing.

**Figure 14.3** 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 14.4** 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

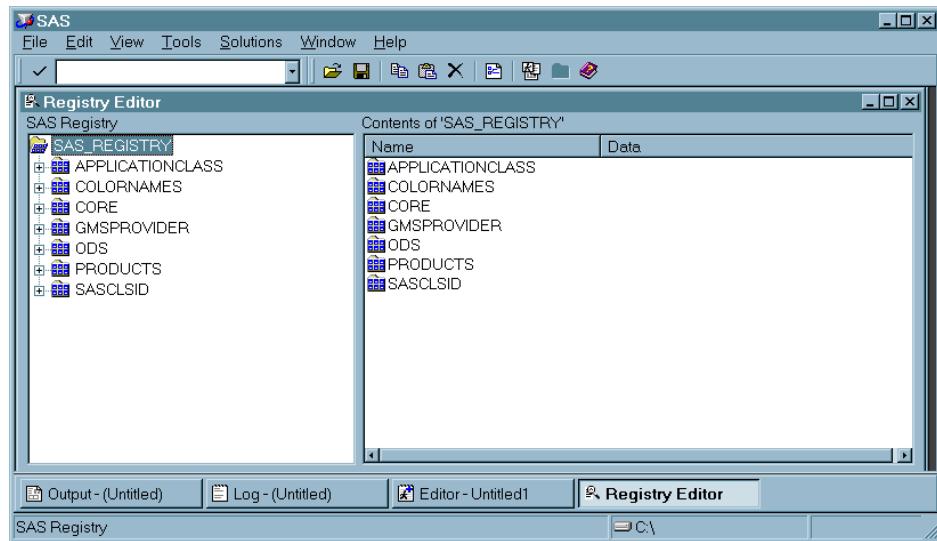
From the SAS Registry Editor:

- 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**  $\Rightarrow$  **Options**  $\Rightarrow$  **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 14.5** 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**  $\Rightarrow$  **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**  $\Rightarrow$  **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**  $\Rightarrow$  **Options**  $\Rightarrow$  **Explorer** menu when you click on the SAS Explorer window. All the Explorer configuration data is stored in the registry under COREExplorer. The following table outlines the location of the most commonly used Explorer configuration data.

*Table 14.1 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 <b>File</b> $\Rightarrow$ <b>New</b> 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**  $\Rightarrow$  **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**  $\Rightarrow$  **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**  $\Rightarrow$  **New** from the Explorer window.

**CAUTION! You can correct many libref assignment errors in the SAS Registry Editor.**

**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**  $\Rightarrow$  **Accessories**  $\Rightarrow$  **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.



# Printing with SAS

---

<b><i>Universal Printing</i></b> .....	<b>265</b>
What Is Universal Printing? .....	265
Setting Up the Universal Printing Interface and the Default Printing Environment .....	265
Universal Printing Output Formats .....	266
Viewing Universal Printers and Printer Prototypes .....	268
Viewing Universal Printer Settings .....	269
Modifying Universal Printing Printer Settings .....	270
Universal Printing and ODS .....	270
Specifying the Page Orientation for Universal Printing Documents .....	271
Color Support for Universal Printers .....	273
Embedding Non-Viewable Comments in Universal Printing Output .....	282
<b><i>Configuring Universal Printing Using the Windowing Environment</i></b> .....	<b>283</b>
Overview of the Universal Printing Menu .....	283
Setting Up Printers .....	284
Printing with Universal Printing .....	292
Working with Previewers .....	294
Set Page Properties .....	297
<b><i>System Options That Control Universal Printing</i></b> .....	<b>300</b>
<b><i>Managing Universal Printers Using the PRTDEF Procedure</i></b> .....	<b>302</b>
About Using the PRTDEF Procedure .....	302
Examples of Creating New Printers and Previewers Using the PRTDEF Procedure .....	303
<b><i>Forms Printing</i></b> .....	<b>308</b>
Overview of Forms Printing .....	308
Creating or Editing a Form .....	309
<b><i>Using Fonts with Universal Printers and SAS/GRAF Devices</i></b> .....	<b>309</b>
Rendering Fonts .....	309
The FONTEMBEDDING and FONTRENDERING System Options .....	312
ODS Styles and TrueType Fonts .....	313
Portability of TrueType Fonts .....	313
International Character Support .....	313
TrueType Fonts Supplied by SAS .....	313
Registering Fonts .....	317
Listing the Registered Fonts for a Device .....	318
Using Fonts .....	320
Examples of Specifying Fonts and Printing International Characters .....	322
<b><i>Creating EMF (Enhanced Metafile Format) Graphics Using Universal Printing</i></b> .....	<b>328</b>
EMF Graphics in SAS .....	328
Creating an EMF Graphic .....	330
Example of Creating an EMF Graphic Using the ODS PRINTER Statement .....	330

<b><i>Creating GIF Images Using Universal Printing</i></b> .....	<b>331</b>
GIF Images in SAS .....	331
Creating a GIF Image .....	332
Example of Creating a GIF Image Using the ODS PRINTER Statement .....	332
<b><i>Creating PCL (Printer Command Language) Files Using Universal Printing</i></b> .....	<b>333</b>
PCL Files in SAS .....	333
Creating a PCL File .....	334
<b><i>Creating PDF Files Using Universal Printing</i></b> .....	<b>335</b>
PDF Files in SAS .....	335
Creating a PDF File .....	335
Example of Creating a PDF Using the ODS PDF Statement .....	336
System Options That Affect PDF Output .....	337
<b><i>Creating PNG (Portable Network Graphics) Files Using Universal Printing</i></b> .....	<b>338</b>
Portable Network Graphics in SAS .....	338
The PNG Universal Printers .....	338
Creating a PNG Image .....	339
Example of Creating a PNG File Using the ODS PRINTER Statement .....	339
Web Browsers and Viewers That Support PNG Files .....	340
<b><i>Creating PostScript Files Using Universal Printing</i></b> .....	<b>341</b>
PostScript Files in SAS .....	341
Creating a PostScript File .....	341
Example of Creating a PostScript File Using the ODS PS Statement .....	342
<b><i>Creating SVG (Scalable Vector Graphics) Files Using Universal Printing</i></b> .....	<b>343</b>
Overview of Scalable Vector Graphics in SAS .....	343
Web Server Content Type for SVG Documents .....	345
The SVG Universal Printers and the Output That They Create .....	345
How to Create SVG Documents .....	346
Browser Support for Viewing SVG Documents .....	349
Images in SVG Documents .....	350
Setting the Environment to Create Stand-alone SVG Documents .....	352
Creating Stand-alone SVG Documents Using the ODS PRINTER Destination .....	358
SVG Documents in HTML Files .....	367
Printing an SVG Document from a Browser .....	372
<b><i>Creating TIFF Images Using Universal Printing</i></b> .....	<b>372</b>
TIFF Images in SAS .....	372
The TIFF Universal Printers .....	373
Creating a TIFF Image .....	373
Example of Creating a TIFF Image Using the ODS PRINTER Statement .....	374
<b><i>Creating Animated GIF Images and SVG Documents</i></b> .....	<b>375</b>
About Animated GIF Images and SVG Documents .....	375
Animation System Options .....	376
Example: Creating an Animated SVG Document .....	377

---

# Universal Printing

---

## What Is Universal Printing?

Universal Printing is a system that provides both interactive and batch printing capabilities to create variety of document and graphic output formats. For example, you can use Universal Printing to create HTML, PDF, documents, and PNG, GIF, and SVG graphics. For a complete list of supported document and graphic types, see [Table 15.1 on page 266](#).

Universal Printing enables you to define printers and print previewers, and to 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.

**Windows Specifics:** By default, the Windows operating environment uses Windows printing and not Universal Printing. For more information about using Universal Printing under Windows, see [“Setting Up the Universal Printing Interface and the Default Printing Environment” on page 265](#).

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. For more information about these system options, see [“System Options That Control Universal Printing” on page 300](#).

**Note:** Before the introduction of Universal Printing, SAS supported a utility for print jobs known as Forms. Forms printing is still available if you select **File**  $\Rightarrow$  **Print Setup** from the menu in the windowing environment. Then check the **Use Forms** check box. This turns off Universal Printing menus and functionality. For more information, see [“Forms Printing” on page 308](#).

---

## Setting Up the Universal Printing Interface and the Default Printing Environment

### Universal Printing in UNIX and z/OS

Universal Printing is enabled when SAS starts in the UNIX and z/OS operating environments. No further action is required.

### Universal Printing in Windows

Under Windows, Windows printing is enabled when SAS starts. To use Universal Printing in Windows, you must set the UNIVERSALPRINT system option to enable the Universal Printing environment, menus, and dialog boxes, and to set up the printing defaults. If you use the SAS windowing environment, you can also use the UPRINTMENUSWITCH system option to enable the print commands on the **File**

menu. These options can be set only in a SAS configuration file or at start-up. You cannot enable or disable Universal Printing menus and dialog boxes after SAS starts.

Include the following system options when you start SAS:

```
-uprint -uprintmenuswitch
```

**UPRINT** is an alias for the **UNIVERSALPRINT** system option.

If you start SAS with only the UPRINT option, you need to close the HTML destination, which is open by default. Use the PRINTERPATH= option to specify the output format type. Then, use an ODS PRINTER statement and ODS PRINTER CLOSE statement around the code that you want to execute. Here is an example:

```
ods html close;
options printerpath=pdf;
ods printer;
proc print data=sashelp.class;
run;
ods printer close;
```

## Return to the Default Printer

When you use the PRINTERPATH= system option to specify a printer, the print job is controlled by Universal Printing. To return to the default Universal Printer (the PostScript printer) set the PRINTERPATH= option to a null value (double quotation marks with no space between them):

```
options printerpath="" ;
```

In Windows, when Universal Printing is not enabled, setting PRINTERPATH= to a null value returns printing to Windows printing.

## Universal Printing Output Formats

In addition to sending print jobs to a printer, you can also direct output to external files that are widely recognized by different types of printers and software programs. You can use Universal Printing to produce the following commonly recognized file types.

*Table 15.1 Available Print Output Formats*

Type	Full Name	Description
GIF	Graphics Interchange Format	An image format designed for the online transmission and interchange of graphic data. The format is widely used to display images on the World Wide Web because of its smaller size and portability.

Type	Full Name	Description
EMF	Enhanced Metafile Format	A metafile format that is a collection of graphic drawing commands, configuration properties, and graphic objects to create true color, scalable, device-independent graphics. Applications that support EMF run on Windows. Universal Printing currently supports EMF, EMFPlus, and EMFDual levels of the metafile format. The EMF Universal Printer uses the EMFPlus level of metafile formatting, which is the default EMF printer.
PCL	Printer Control Language	Developed by Hewlett-Packard as a language that applications use to control a wide range of printer features across a number of printing devices. Universal Printing currently supports PCL4, PCL5, PCL5e, and PCL5c levels of the language.
PDF	Portable Document Format	A file format developed by Adobe Systems for viewing and printing a formatted document. To view a file in PDF format, you need Adobe Reader, a free application distributed by Adobe Systems.  Note: Adobe Acrobat is not required to produce PDF files with Universal Printing.
PNG	Portable Network Graphics	An image format that was designed as a replacement for the older simple GIF format and the more complex TIFF format. As with GIF, one of the major uses of PNG is to display images on the web. PNG has these major advantages over GIF on the web: gamma correction, two-dimensional interlacing, variable transparency (alpha channel), setting the resolution, and more than 256 colors.
PS	PostScript	A page description language developed by Adobe Systems. This is the default Universal Printer.
SVG	Scalable Vector Graphics	A vector format that is a language for describing two-dimensional graphics and graphical applications in XML.
TIFF	Tagged Image File Format	An Adobe raster image format that supports both image and data in a single file. The TIFF Universal Printer supports RGBA color printing and transparency. The TIFFk Universal Printer supports CYMK color printing.

You set the value of the PRINTERPATH= system option to a Universal Printer or use ODS statements to create output in one of the above formats. When the PRINTERPATH= system option is set to a printer that prints to a file, the default filename is sasprt.*extension*. *extension* is the printer format type. Here are some example filenames: sasprt.pdf, sasprt.emf, sasprt.png, and sasprt.gif. The file is written to the current directory.

You can use the PRINTERPATH= system option to change the location and the name of the file. Here is an example:

```
options printerpath=(svg out);
filename out 'c:\myimages\graph1.svg';
```

## Viewing Universal Printers and Printer Prototypes

SAS provides Universal Printers and printer prototypes that you can use to create your own printers. You can access the list of available printers from the Print dialog box. You can also use the QDEVICE procedure to create a data set of printers and then print the printer information using the PRINT procedure.

To create a table of printers and print the list with a description of each printer, submit this code:

```
proc qdevice out=printers;
  printer _all_;
run;
```

```
proc print data=printers;
  var name desc;
run;
```

For more information, see “[QDEVICE Procedure](#)” in *Base SAS Procedures Guide*.

To print a list of printer prototypes to the SAS log, submit this SAS program:

```
filename registry temp;
proc printto log=registry;
run;

proc registry list keysonly levels=1 startat="core\printing\prototypes";
proc printto;
run;

data prototypes;
  keep prototype;
  infile registry lrecl=300 pad;
  length line $300;
  input line $300.;
  if substr(line,1,1) = "["
    then do;
      prototype = strip(substr(line,2,length(line)-2));
      if index(prototype,'core\printing\prototypes') ne 0
        then delete;
      else
        output;
    end;
  run;
```

```
proc print label;
  label prototype = "Prototype";
run;
```

For more information, see “[REGISTRY Procedure](#)” in *Base SAS Procedures Guide*.

## Viewing Universal Printer Settings

You can use the QDEVICE procedure or the Print dialog box to view the settings of a Universal Printer. To view printer settings using the QDEVICE procedure, submit this code:

```
proc qdevice;  
  printer printer-name;  
  run;
```

Here are the printer settings for the GIF printer:

```
19  proc qdevice;  
20  printer gif;  
21  run;  
  
      Name: GIF  
      Description: Graphics Interchange Format RGB Color/Alpha Blending  
      Module: SASPDGIF  
      Type: Universal Printer  
      Registry: SASHELP  
      Prototype: GIF  
Default Typeface: Cumberland AMT  
Typeface Alias: Courier  
      Font Style: Regular  
      Font Weight: Normal  
      Font Height: 8 points  
      Font Version: Version 1.03  
Maximum Colors: 16777216  
Visual Color: True Color  
Color Support: RGBA  
Destination: sasprt.gif  
I/O Type: DISK  
Data Format: GIF  
      Height: 6.25 inches  
      Width: 8.33 inches  
      Ypixels: 600  
      Xpixels: 800  
Rows(vpos): 50  
Columns(hpos): 114  
      Left Margin: 0 inches  
Minimum Left Margin: 0 inches  
      Right Margin: 0 inches  
Minimum Right Margin: 0 inches  
      Bottom Margin: 0 inches  
Minimum Bottom Margin: 0 inches  
      Top Margin: 0 inches  
Minimum Top Margin: 0 inches  
XxY Resolution: 96x96 pixels per inch  
Compression Enabled: Always  
Compression Method: LZW  
Font Embedding: Never  
Animation: Enabled
```

The QDEVICE procedure does not report all printer settings. For a description of the printer settings that can be reported, see “[QDEVICE Procedure](#)” in *Base SAS Procedures Guide*.

## Modifying Universal Printing Printer Settings

You modify printer settings using the Universal Printer dialog boxes, by setting SAS system options, or by using the PRTDEF procedure. See the following topics:

- “[Configuring Universal Printing Using the Windowing Environment](#)” on page 283
- “[System Options That Control Universal Printing](#)” on page 300
- “[Managing Universal Printers Using the PRTDEF Procedure](#)” on page 302

## Universal Printing and ODS

The ODS PRINTER statement can use Universal Printing whether the UNIVERSALPRINT or NOUNIVERSALPRINT system option is set. The PRINTER destinations used by the ODS PRINTER statement are described in the “[ODS PRINTER Statement](#)” in *SAS Output Delivery System: User’s Guide*.

The Output Delivery System (ODS) uses Universal Printing for the following ODS statements.

**Table 15.2** ODS Statements That Use Universal Printing

ODS Statement	Description
Document Formats	
ODS PRINTER PRINTER= option	Uses the selected printer.
ODS PDF statement	Uses the Universal Printing PDF printer.*
ODS PS statement	Uses the Universal Printing PostScript Level 1 printer.
ODS PCL statement	Uses the Universal Printing PCL5 printer.
Graphic Formats	
ODS LISTING	Uses the selected printer.
ODS HTML	Use with ODS Graphics and SAS/GRAFH.
ODS HTML5	Use with ODS Graphics and SAS/GRAFH.
ODS RTF	Use with ODS Graphics and SAS/GRAFH.
ODS EPUB	Use with ODS Graphics and SAS/GRAFH.

\* You must have SAS/GRAFH installed to create drill-down regions in a graph created by the PDF Universal Printer. For more information, see “[Adding Drill-Down Graphs in Your PDF File](#)” in *SAS/GRAFH: Reference*.

**Windows Specifics:** In the Windows operating environment, the ODS PRINTER destination uses the Windows system printers unless SAS is started with the UNIVERSALPRINT system option, or when you specify a printer with the PRINTERPATH= system option. If Universal Printing is enabled in Windows, SAS overrides the use of the Windows system printer and causes ODS to use Universal Printing. To return to Windows printing, set the PRINTERPATH= system option to a null string: PRINTERPATH="" (double quotation marks with no space between them).

For more information about ODS, see [SAS Output Delivery System: User's Guide](#).

---

## Specifying the Page Orientation for Universal Printing Documents

You can specify the page orientation for each page of a multiple-page document that is created by a Universal Printer. You can also use page orientation for documents that are created for the ODS LISTING, PCL, PDF, PRINTER, and PS destinations.

The ORIENTATION= system option has four values: PORTRAIT, LANDSCAPE, REVERSEPORTRAIT, and REVERSELANDSCAPE. To change the orientation of a document page, specify the OPTIONS statement, using the ORIENTATION= system option, between the steps that create output to change the page orientation.

**Note:** The EMF, GIF, PNG, and TIFF Universal Printers do not support multiple-page documents. These printers also do not support the REVERSELANDSCAPE and the REVERSEPORTRAIT orientations.

The following example creates a four-page SVG document. The orientation is changed between landscape and portrait for each page in the document. The OPTIONS statements are highlighted:

```
options nodate nonumber;
ods printer printer=svgview file='orientation.svg' style=Ocean;
title 'Demonstration of Page Orientation Changes in a Document';
footnote 'PROC SGPLOT in Landscape Orientation';
options orientation=landscape;
proc sgplot data=sashelp.class;
vbar age;
run;

options orientation=portrait;
footnote 'PROC PRINT in Portrait Orientation';
proc print data=sashelp.class;
run;

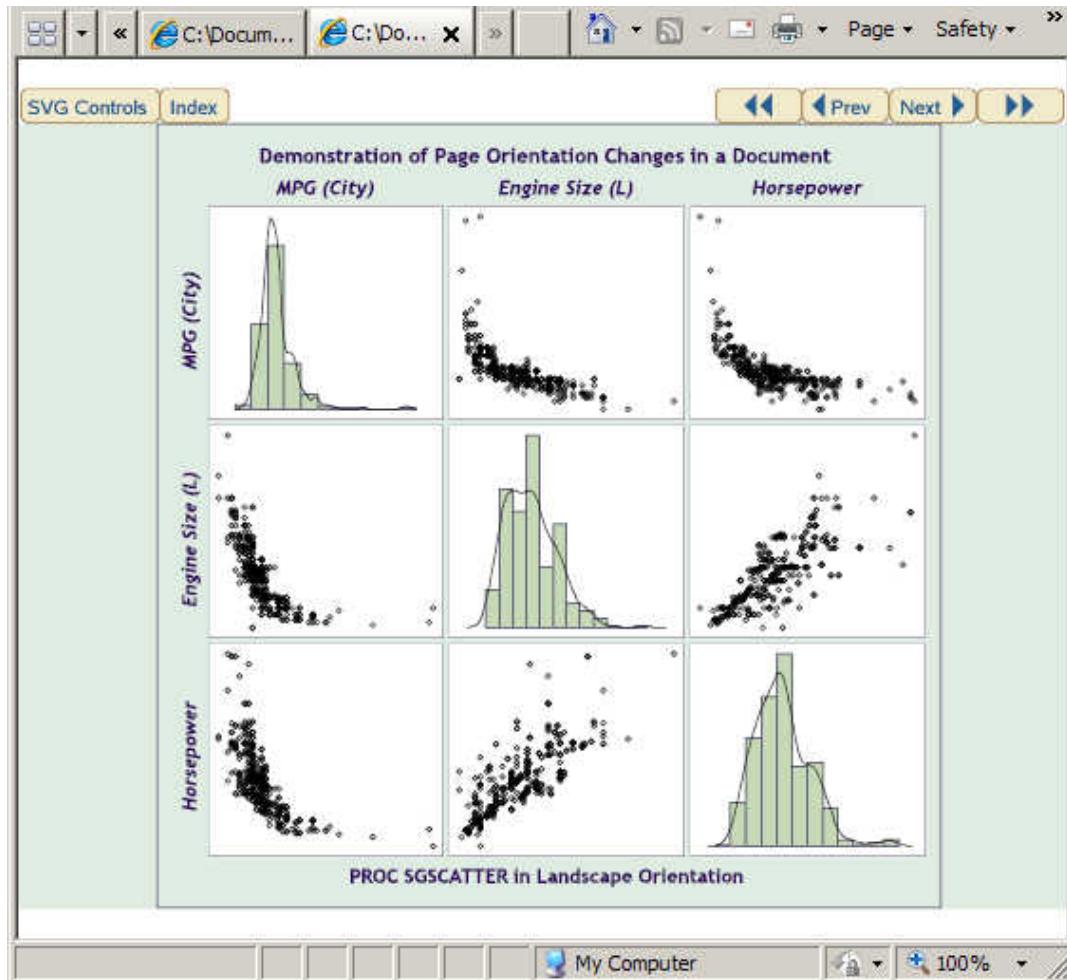
options orientation=landscape;
footnote 'PROC SGSCATTER in Landscape Orientation';
proc sgscatter data=sashelp.cars;
matrix mpg_city enginesize horsepower /
diagonal=(histogram kernel);
run;

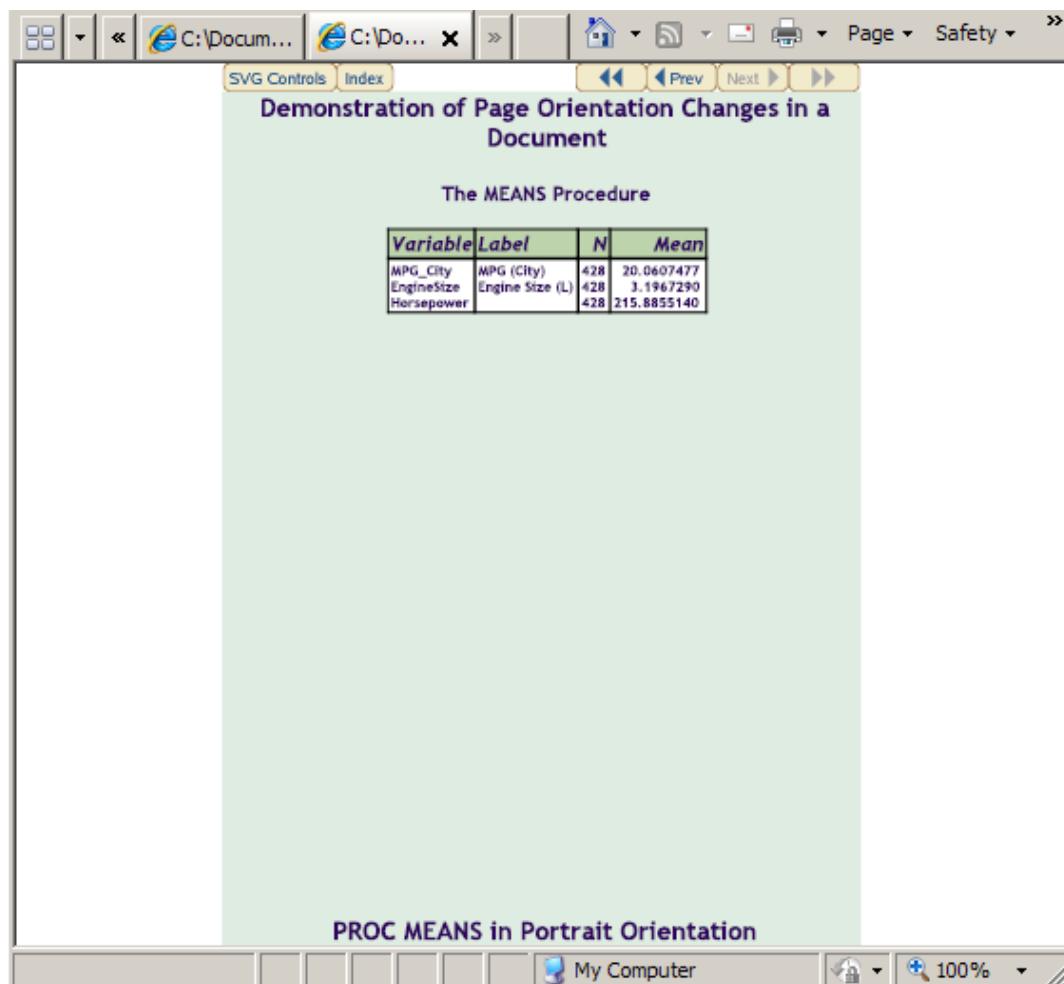
options orientation=portrait;
footnote 'PROC MEANS in Portrait Orientation';
proc means data=sashelp.cars n mean;
```

```
var mpg_city enginesize horsepower;  
run;  
  
ods printer close;
```

The following output shows the third and fourth pages of the document. The third page has a landscape orientation and the fourth page has a portrait orientation.

Figure 15.1 Page Three of an SVG Document Showing the Landscape Orientation



**Figure 15.2** Page Four of an SVG Document Showing the Portrait Orientation

For more information, see “[ORIENTATION= System Option](#)” in *SAS System Options: Reference*.

## Color Support for Universal Printers

### Universal Printers and the Color Spaces They Support

All Universal Printers support 24-bit RGB colors. Most printers support 32-bit CMYK colors or 32-bit RGBA (transparency) colors. The following table lists the Universal Printers and their respective color support.

**Table 15.3** Color Support for Universal Printers

Universal Printer	Color Support	Supports Transparency
EMF	RGBA (32-bit)	Yes

Universal Printer	Color Support	Supports Transparency
EMFDual	If the EMF viewer supports the EMFPlus format, the color support is RGBA If the EMF viewer does not support EMFPlus format, the color support is RGBA for bitmap images and RGB for vector elements.	Yes Yes, for bitmap images
SASEMF	RGBA only for bitmap images. RGB for vector elements.	Yes, for bitmap images
GIF	RGBA (24-bit with support for transparent backgrounds)	Yes
PCL5c*	RGBA	Yes
PDF	CMYK and RGBA	Yes, for RGBA colors
PNG	RGBA (32-bit)	Yes
PostScript	CMYK and RGB RGBA for GIF images **	No No
SVG	RGBA (32-bit)	Yes
TIFF	RGBA	Yes
TIFFk	CYMK	No

\* PCL4 and PCL5 Universal Printers support only monochrome printing.

\*\* The PostScript Universal Printer recognizes RGBA colors but transparency is not supported.

For information about CYMK, RGB, and RGBA colors, see “[CYMK Colors](#)” on page 274 and “[RGB and RGBA Colors](#)” on page 276.

## CMYK Colors

CMYK colors setting specify eight hexadecimal characters with a value of 0–255 to specify the amount of cyan, magenta, yellow, and black ink. Use your printer’s Pantone Color Lookup table to find the CMYK values for your printer. If you specify an unsupported color, such as a CMYK color with an EMF printer, the color is converted to a color that is supported.

You can specify CMYK colors where ever colors can be set (for example, in the PROC PRINT statement STYLE option or in the TITLE statement).

Preface the hexadecimal number with a CMYK or a K. Here are some examples of CMYK colors that you can set in SAS:

**Table 15.4** Example CMYK Colors

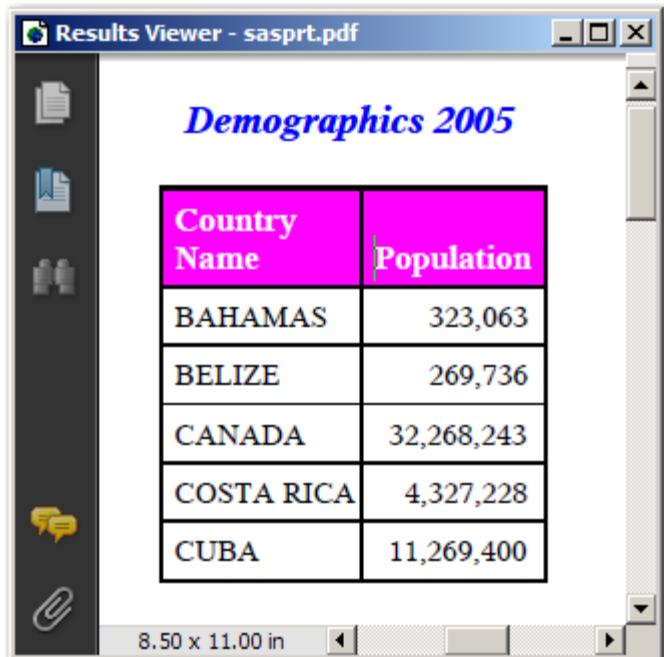
Hexadecimal Representation	Color
cmykFF000000	cyan
k00FF0000	magenta
cmyk0000FF00	yellow
kFFFF0000	blue
cmykFF00FF00	green
k00FFFF00	red
cmykFFFFFF00	process black, using cyan, magenta, and yellow
k000000FF	black

The first byte of the hexadecimal number represents cyan. The second byte represents magenta. The third byte represents yellow. The fourth byte represents black.

This example uses the STYLE option to set the column background color to magenta and sets the foreground color to white. The TITLE statement sets the output title to blue.

```
options obs=5 nodate;
ods html close;
ods pdf;
proc print data=sashelp.demographics label
    style(header)={background=cmyk00ff0000 foreground=k00000000} noobs;
var name pop;
label name=Country Name pop=Population;
title color=kfffff0000 'Demographics 2005';
run;
ods pdf close;
ods html;
```

Figure 15.3 CMYK Color Specified in the STYLE Option



**Note:** When you use the TIFFk Universal Printer, specify the UPRINTCOMPRESSION system option to avoid very large TIFF files.

## RGB and RGBA Colors

RGB and RGBA colors combine red, green, and blue colors in different ratios to create colors. The A is the alpha channel, which represents a percentage of opacity.

You specify RGB colors as a triple of hexadecimal numbers, ranging from 00–FF. Each hexadecimal number indicates how much of the red, green, or blue is included in the color. RGBA color includes an additional hexadecimal number for the alpha channel that indicates how transparent the color is. FF is opaque and 00 is transparent. In both RGB and RGBA color specifications, the first hexadecimal number is red, the second, is green, and the third is blue. In RGBA colors, the fourth hexadecimal number is the alpha channel specification.

You can specify RGB and RGBA colors wherever colors can be set (for example, as an option in the VBAR statement in the SGLOT procedure or in the TITLE statement). For RGB colors, preface the hexadecimal number with a CX. For RGBA colors, preface the hexadecimal number with RGBA or A.

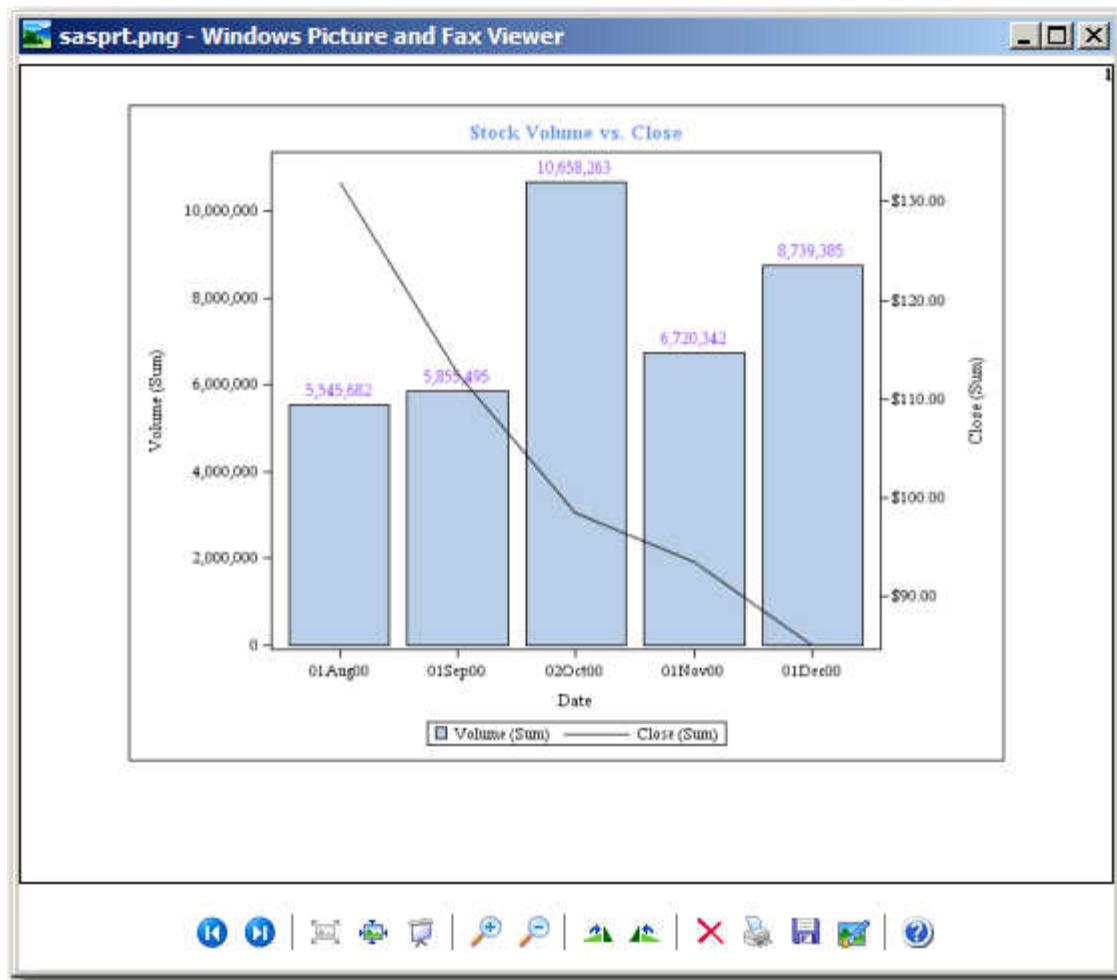
The following SGLOT procedure uses an RGBA color to create the bar labels:

```
ods html close;
ods printer printer=png;
proc sgplot data=sashelp.stocks (where=(date >= "01jan2000"d
                                         and date <= "01jan2001"d
                                         and stock = "IBM"));
  title color=a6495edff "Stock Volume vs. Close";
  vbar date / response=volume
    datalabel
      datalabelatrrs=(color=a8a44ff8a size=10);
  vline date / response=close y2axis;
```

```
run;
title;
ods printer close;
ods html;
```

Here is the PNG file with bar labels:

**Figure 15.4** *RGB Color Specified for the Bar Labels*



### Example: Static and Varying Background Color in a Table Using RGBA Colors

This example program does the following:

- Creates the format PCT. using a DATA \_NULL\_ statement. The DATA step defines salary ranges of \$3,000.00 and calculates an RGBA color value for each salary range. The CALL EXECUTE statement is used to output the FORMAT procedure code as it is generated.
- Creates a data set.
- The PRINT procedure uses an RGBA color value for the background of the table header and formats the salary variable using the PCT. format.

```
options nodate;
```

```

/* Create the PCT format. */  

/* The color variable is a concatenation of calculated */  

/* hexadecimal values. */  

data _null_ ;  

  call execute('proc format fmtlib ; value pct');  

  max=10000;  

  maxloop=255;  

  do i=1 to maxloop by 10;  

    color='RGBA'||put(((maxloop)/(maxloop+i)*200),hex2.)  

      ||put(((maxloop)/(maxloop+i)*235),hex2.)  

      ||put(((maxloop)/(maxloop+i)*255),hex2.)||'95';  

    from=max;  

    to=(max+3000);  

    max=max+3000;  

    /* Create salary ranges of $3000.00 equal to the calculated RGBA color value. */  

    call execute(put(from,best.)||'-'||put(to,best.)||'='||quote(color));  

  end;  

/* Create RGBA values for missing values and values outside the salary ranges. */  

call execute('.="RGBAF7F5F0480" other="RGBAFF2A2A88"; run;');
run;  

data staff;  

  infile datalines dlm='#';  

  input Name $16. IdNumber $ Salary  

    Site $ HireDate date7.;  

  format hiredate date7.;  

  datalines;  

Capalleti, Jimmy# 2355# 21163# BR1# 30JAN09  

Chen, Len# 5889# 20976# BR1# 18JUN06  

Davis, Brad# 3878# 19571# BR2# 20MAR84  

Leung, Brenda# 4409# 34321# BR2# 18SEP94  

Martinez, Maria# 3985# 49056# US2# 10JAN93  

Orfali, Philip# 0740# 50092# US2# 16FEB03  

Patel, Mary# 2398# 35182# BR3# 02FEB90  

Smith, Robert# 5162# 40100# BR5# 15APR66  

Sorrell, Joseph# 4421# 38760# US1# 19JUN11  

Zook, Carla# 7385# 22988# BR3# 18DEC10  

;  

run;
ods html close;
ods pdf file='outpdf.pdf';
proc print data=staff noobs label
  style(H HEADER)={background=rgbac7eafe95 fontstyle=italic}
  style(D DATA)={foreground=black};
var name IdNumber ;
var salary /style(DATA)={background=pct.};
label IdNumber='Employee Number' salary='Salary in U.S. Dollars';
format salary dollar7.;
title 'Generated Colors for the Variable Salary';
run;
ods pdf close;

```

**Example Code 15.1 Static and Varying Background Color in a Table Using RGBA Colors**

```
501 options nodate;
502
503 /* Create the PCT format. */ 
504 /* The color variable is a concatenation of calculated */
505 /* hexadecimal values. */ 
506
507 data _null_ ;
508   call execute('proc format fmtlib ; value pct');
509   max=10000;
510   maxloop=255;
511   do i=1 to maxloop by 10;
512     color='RGBA'||put(((maxloop)/(maxloop+i)*200),hex2.)||put((maxloop)/
(maxloop+i)*235),
512! hex2.)
513     ||put((maxloop)/(maxloop+i)*255),hex2.)||'95';
514   from=max;
515   to=(max+3000);
516   max=max+3000;
517
518   /* Create salary ranges of $3000.00 equal to the calculated RGBA color
value.*/
519   call execute(put(from,best.)||'-'||put(to,best.)||'='||quote(color));
520 end;
521
522 /* Create RGBA values for missing values and values outside the salary
ranges. */
523 call execute('.="RGBAF7F5F0480" other="RGBAFF2A2A88"; run;');
524 run;

NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time          0.00 seconds
```

```

NOTE: CALL EXECUTE generated line.
1   + proc format fmtlib ;
1   +           value pct
2   +             10000-13000="RGBAC7EAFE95"
3   +             13000-16000="RGBABFE1F495"
4   +             16000-19000="RGBAB8D9EB95"
5   +             19000-22000="RGBAB2D1E395"
6   +             22000-25000="RGBAAACCADB95"
7   +             25000-28000="RGBAA6C3D495"
8   +             28000-31000="RGBAA1BDCD95"
9   +             31000-34000="RGBAA9CB7C795"
10  +            34000-37000="RGBAA97B2C195"
11  +            37000-40000="RGBAA93ADBB95"
12  +            40000-43000="RGBAA8FA8B695"
13  +            43000-46000="RGBAA8BA3B195"
14  +            46000-49000="RGBAA879FAC95"
15  +            49000-52000="RGBAA849BA895"
16  +            52000-55000="RGBAA8097A495"
17  +            55000-58000="RGBAA7D93A095"
18  +            58000-61000="RGBAA7A909C95"
19  +            61000-64000="RGBAA778C9895"
20  +            64000-67000="RGBAA74899595"
21  +            67000-70000="RGBAA72869195"
22  +            70000-73000="RGBAA6F838E95"
23  +            73000-76000="RGBAA6D808B95"
24  +            76000-79000="RGBAA6B7D8895"
25  +            79000-82000="RGBAA687B8595"
26  +            82000-85000="RGBAA66788395"
27  +            85000-88000="RGBAA64768095"
28  + .="RGBAF7F5F0480" other="RGBAFF2A2A88";
NOTE: Format PCT has been output.
28  +                                     run;

NOTE: PROCEDURE FORMAT used (Total process time):
      real time          0.03 seconds
      cpu time          0.01 seconds

525
526 data staff;
527   infile datalines dlm='#';
528   input Name $16. IdNumber $ Salary
529     Site $ HireDate date7. ;
530   format hiredate date7. ;
531   datalines;

NOTE: The data set Work.Staff has 10 observations and 5 variables.
NOTE: DATA statement used (Total process time):
      real time          0.01 seconds
      cpu time          0.01 seconds

542 ;
543 run;
544

```

```
545 /* Close the HTML destination and open the PDF destination.*/
546 /* Format the header background using an RGBA color.          */
547 /* Use the PCT. format to format the salary variable.        */
548
549 ods html close;
550 ods pdf file='outpdf.pdf';
NOTE: Writing ODS PDF output to DISK destination "c:\public\mySASPrograms
\outpdf.pdf",
      printer "PDF".
551 proc print data=staff noobs label
      style(H HEADER)={background=rgbac7eafe95 fontstyle=italic}
      style(D DATA)={foreground=black};
554 var name IdNumber ;
555 var salary /style(DATA)={background=pct.};
556 label IdNumber='Employee Number' salary='Salary in U.S. Dollars';
557 format salary dollar7.;
558 title 'Generated Colors for the Variable Salary';
559 run;

NOTE: There were 10 observations read from the data set Work.Staff.
NOTE: PROCEDURE PRINT used (Total process time):
      real time          0.03 seconds
      cpu time          0.03 seconds

560 ods pdf close;
NOTE: ODS PDF printed 1 page to c:\public\mySASPrograms\outpdf.pdf.
561 ods html;
NOTE: Writing HTML Body file: sashmtl7.htm
```

Here is the formatted PDF output:

**Output 15.1 PDF Formatted Using RGBA Color Values**

The screenshot shows the SAS Results Viewer interface with a PDF document titled "Generated Colors for the Variable Salary". The document contains a table with the following data:

Name	Employee Number	Salary in U.S. Dollars
Capalleti, Jimmy	2355	\$21,163
Chen, Len#	5889	\$20,976
Davis, Brad#	3878	\$19,571
Leung, Brenda#	4409	\$34,321
Martinez, Maria#	3985	\$49,056
Orfali, Philip#	0740	\$50,092
Patel, Mary#	2398	\$35,182
Smith, Robert#	5162	\$40,100
Sorrell, Joseph#	4421	\$38,760
Zook, Carla#	7385	\$22,988

---

## Embedding Non-Viewable Comments in Universal Printing Output

You can embed a comment in Universal Printer output that does not appear in the output when the file is displayed or printed. The comment can be a text string up to 4,000 characters that you specify using the COLOPHON= system option. You might want to use the comment as a digital signature or to identify the image, vector graphic, or PDF file. You can use a text editor or a third-party application to view the text string in the file.

This example adds text to an SVG document using the COLOPHON= option::

```
options printerpath=svg colophon='Colophon text: SVG SGPLOT for sashelp.class';
ods html close;
ods printer;
proc sgplot data=sashelp.class;
  reg x=height y=weight / CLM CLI;
run;
ods printer close;
ods html;
```

Here is the comment in the SVG document on the second line:

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink">
<!-- Colophon text: SVG SGPLOT for sashelp.class -->
```

For more information, see “[COLOPHON= System Option](#)” in [SAS System Options: Reference](#).

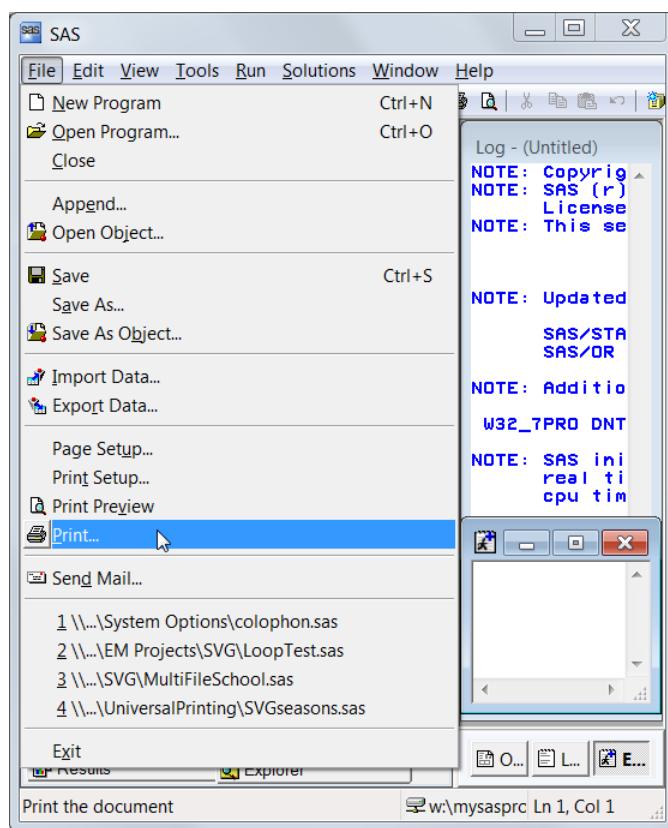
## Configuring Universal Printing Using the Windowing Environment

### Overview of the Universal Printing Menu

SAS Universal Printing windows are accessible from the **File** menu.

The following display shows the **File** menu containing the Universal Printing choices of **Page Setup**, **Print Setup**, **Print Preview**, and **Print**.

*Figure 15.5 File Menu Displaying Universal Printing Options*



**Table 15.5** Menu Choices or Commands to Open Universal Printing Windows

Menu Choice	Equivalent Command
Page Setup	DMPAGESETUP
Print Setup	DMPRINTSETUP
Print Preview	DMPRTPREVIEW
Print	DMPRINT

**Windows Specifics:** In the Windows operating environment, SAS uses the Windows print windows as the default. To access the Universal Printing user interface, the UNIVERSALPRINT system option must be set. To do this, include the following line of code in the string that you use to invoke SAS in Windows:

```
-uprint
```

UPRINT is an alias for the UNIVERSALPRINT system option.

You can open Universal Printing windows by entering commands at the command line or into the command box in the menu bar. The following table lists the commands for the most common tasks.

**Table 15.6** Commands to Open Universal Printing Windows

Action	Command
Print the current window	DMPRINT
Change the default printer	DMPRINT or DMPRINTSETUP
Create a new printer or previewer definition	DMPRTC CREATE PRINTER or DMPRTC CREATE PREVIEWER
Modify, add, remove, or test printer definitions	DMPRINTSETUP
Show default printer properties sheet	DMPRINTPROPS
Show page properties sheet	DMPAGESETUP
Print preview the current window	DMPRTPREVIEW

## Setting Up Printers

### Print Setup Window

The **File**  $\Rightarrow$  **Print Setup** menu selection opens the Print Setup window, where you can perform the following tasks:

- Change the default printer.
- Remove a printer from the selection list.
- Print a test page.
- Open the Printer Properties window.
- Launch the New Printer wizard.

Alternatively, you can issue the DMPRINTSETUP command.

## Change the Default Printer

To change the default printer device for this SAS session and future SAS sessions, follow these steps:

- 1 Select **File**  $\Rightarrow$  **Print Setup**. The Print Setup window appears.
- 2 Select the new default device from the list of printers in the **Printer** field.
- 3 Click **OK**.

Alternatively, you can issue the DMPRINTSETUP command.

## Remove a Printer from the Selection List

To remove a printer from the selection list:

- 1 Select **File**  $\Rightarrow$  **Print Setup**. The Print Setup window appears.
- 2 Select the printer that you want to delete from the list of printers in the **Printer** field
- 3 Click **Remove**.

**Note:** Only your system administrator can remove printers that the administrator has defined for your site. If you select a printer that was defined by your system administrator, the **Remove** button is unavailable.

Alternatively, you can issue the DMPRINTSETUP command.

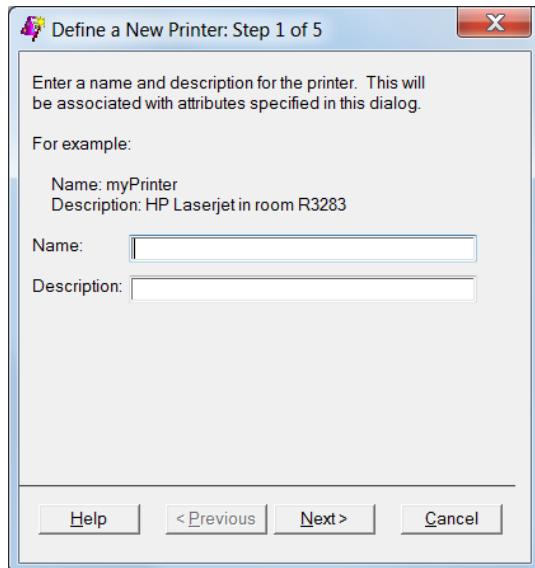
## Define a New Printer

While Universal Printing provides you with predefined printers, you can also add your own printers with the Define a New Printer wizard. This wizard guides you step-by-step through the process of installing a printer.

To start the New Printer wizard and define a new printer, follow these steps:

- 1 Select **File**  $\Rightarrow$  **Print Setup** and click **New**.

The following window appears.

**Figure 15.6** Printer Definition Window to Enter Name and Description

Alternatively, you can either issue the DMPRTCREATE PRINTER command or issue the DMPRINTSETUP command and click **New**.

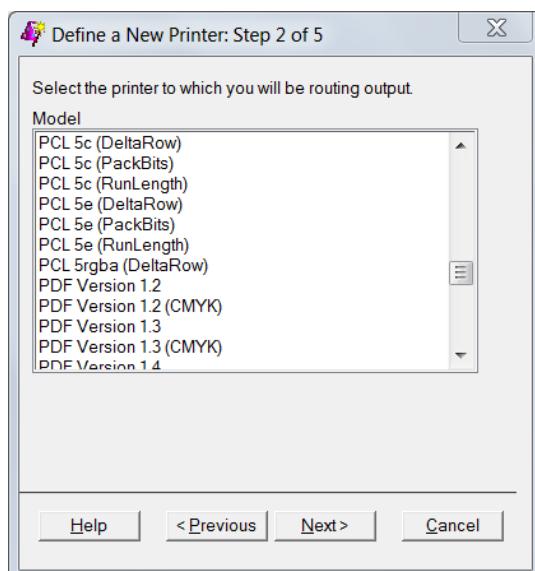
- 2 Enter the name and a description for the new printer (127-character maximum, no backslash characters, not case sensitive).

The printer name is required. The description is optional.

- 3 Click **Next** to proceed to Step 2 of the wizard.

Select a printer model. If your exact printer model is not available, select a general model that is compatible with your printer. For example, for the HP LaserJet printer series, select PCL5 for monochrome printers or PCL5c for color printers.

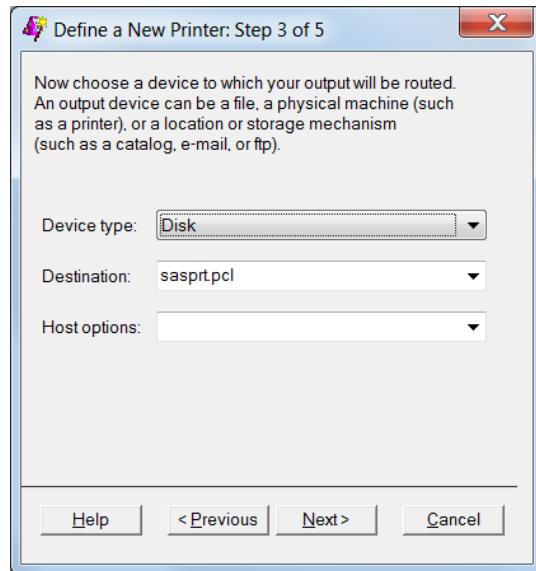
**Note:** General models might provide fewer options than specific models.

**Figure 15.7** Printer Definition Window to Select Printer Model

- 4 Click **Next** to proceed to Step 3 of the wizard.

The following window appears:

**Figure 15.8** Printer Definition Window to Select Output Device



- 5 Select the **Device type** for your print output. Put this sentence in a paragraph under the numbered item

The device type selections are host-dependent.

If you select **Catalog**, **Disk**, **Ftp**, **Socket**, or **Pipe** as the device type, then you must specify a destination.

If you select a device type of **Printer**, then a destination might not be required, because some operating environments use the Host options box to route output.

**Note:** Examples for your operating system of Device Type, Destination, and Host options are also provided in “[Sample Values for the Device Type, Destination, and Host Options Fields](#)” on page 307.

- 6 Enter the **Destination** for your file.

The destination is the target location for your device type. For example, if your device type is **disk**, then your destination is an operating environment-specific filename. With some system device types, the destination might be blank and you can specify the target location using the **Host options** box.

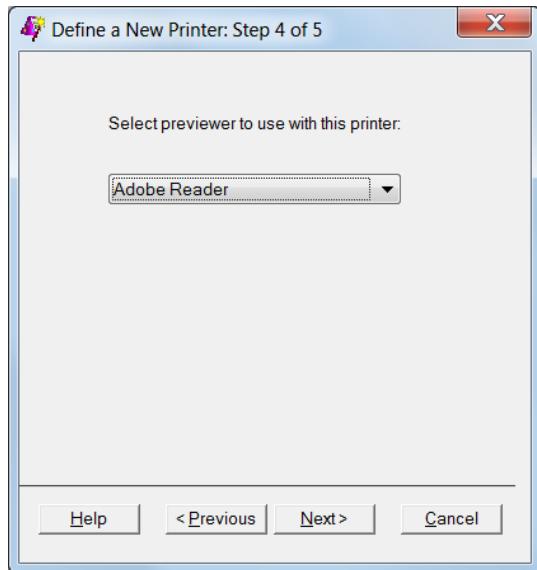
- 7 Select or enter any host-specific options for the device that you chose.

This field might be optional for your operating environment. For a list of host options, see the FILENAME statement information for your operating environment.

**Note:** The **Destination** and **Host Options** lists can also be populated using the REGISTRY procedure. Click the **Help** button in step 3 to see the “[Populating Destination and Host Option Lists](#)” topic, which contains more details.

- 8 Click **Next** to proceed to Step 4 of the wizard, in which you select from a list of installed print previewers.

If no previewers are defined, proceed to the next step of the wizard.

**Figure 15.9** Printer Definition Window to Select Previewer

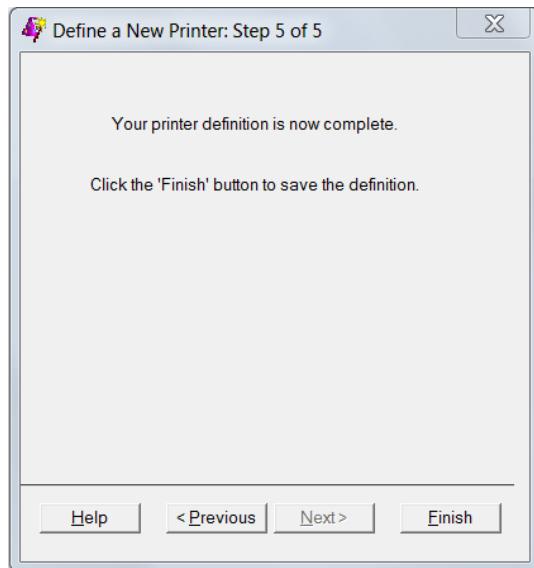
If the previewer selection box appears, select the previewer for this printer. If you do not need a previewer, choose **None** or leave the field blank.

**Note:** You can add a previewer to any printer through the DMPRTCREATE PREVIEWER command. For more information, see “[Define a New Previewer](#)” on page 294.

**Note:** It is not required that printers and print previewers share a common language.

- 9 Click **Next** to proceed to Step 5 of the wizard.

The following window appears:

**Figure 15.10** Printer Definition Window to Complete Process

- 10 Click **Previous** to change any information. Click **Finish** when you have completed your printer definition.

You have now finished setting your default printer.

After you have returned to the Print Setup window, you can test your default printer by clicking **Print Test Page**.

**Note:** You can also use the PRTDEF procedure to define a printer programmatically. For more information, see “[Managing Universal Printers Using the PRTDEF Procedure](#)” on page 302.

## Set Printer Properties for Your Default Printer

Printer properties that you can change include the following:

- the printer name and description
- the printer destination device and its properties
- the default font for the printer
- advanced features such as translation tables, printer resolution, and the print previewer associated with the printer

To change printer properties for your default printer, follow these steps:

- 1 Select **File**  $\Rightarrow$  **Print Setup** and choose **Properties**.

The Printer Properties window appears.

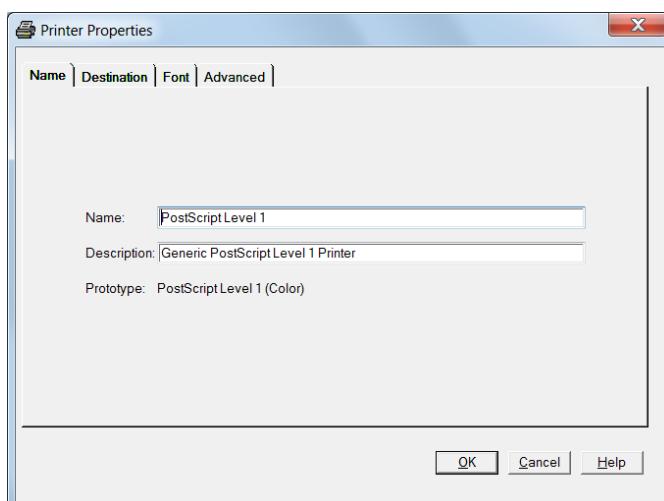
Alternatively, you can issue the DMPRTPROPS command.

- 2 From the Printer Properties window, select the tab that contains the information that you need to modify.

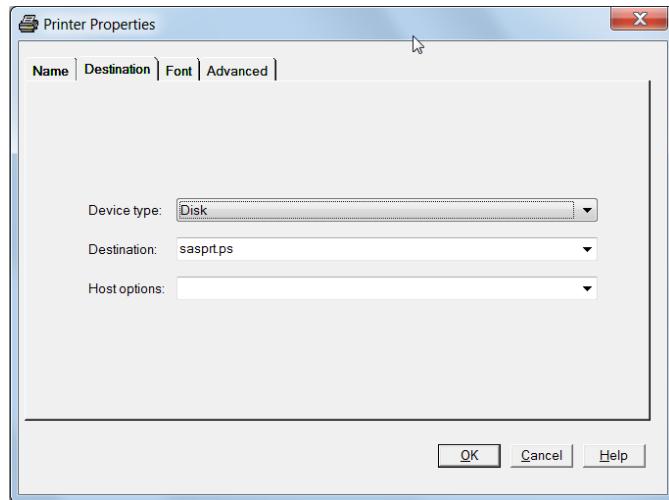
- In the **Name** tab, you can modify the printer name and the printer description.

**Note:** The printer name is not case sensitive. If you change only the casing, the printer name change fails. To change the case of the printer name, you can delete the printer and re-create it with the new casing. You can also modify the name of the printer, save the modifications, and then change the name again to the name and casing that you want.

*Figure 15.11 Printer Properties Window Displaying Name Tab*

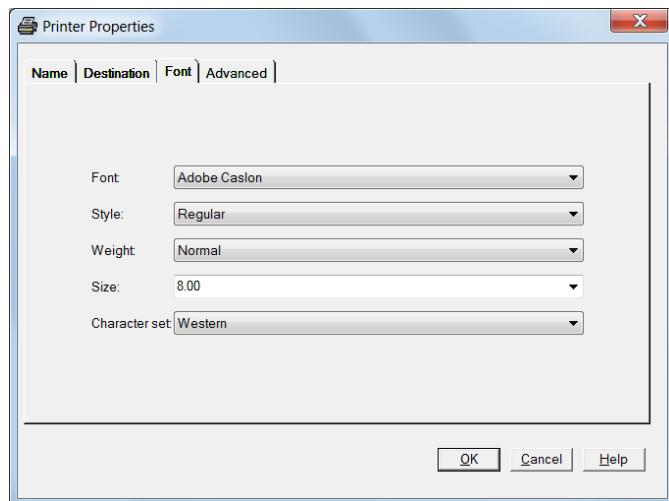


- The **Destination** tab enables you to designate the device type, destination, and host options for the printer. See “[Sample Values for the Device Type, Destination, and Host Options Fields](#)” on page 307 for examples.

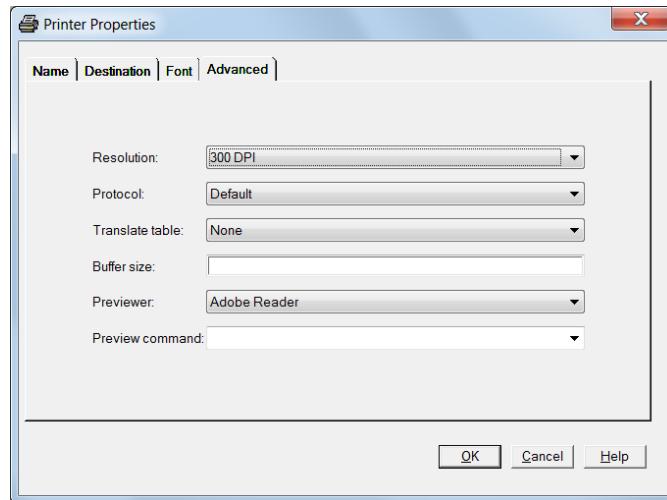
**Figure 15.12** Printer Properties Window Displaying Destination Tab

- The **Font** tab controls the available font options. The selections available in the drop-down boxes are printer specific. The font size is in points.

**Note:** This window enables you to set attributes for the default fonts. Typically, procedure output is controlled by the fonts specified by the ODS style or by program statements that specify font attributes.

**Figure 15.13** Printer Properties Window Displaying Font Tab

- The **Advanced** tab lists the Resolution, Protocol, Translate table, Buffer size, Previewer, and Preview command options for the printer. The information in the drop-down fields is printer specific.

**Figure 15.14** Printer Properties Window Displaying Advanced Tab**Resolution**

specifies the resolution for the printed output in dots per inch (dpi).

**Protocol**

provides the mechanism for converting the output to a format that can be processed by a protocol converter that connects the EBCDIC host mainframe to an ASCII device. Protocol is required in the z/OS operating environment, and if you must use one, select one of the protocol converters that are listed.

**Translate table**

manages the transfer of data between an EBCDIC host and an ASCII device. Normally, the driver selects the correct table for your locale; the translate table needs to be specified only when you require nonstandard translation.

**Buffer size**

controls the size of the output buffer or record length. If the buffer size is left blank, a default size is used.

**Previewer**

specifies the Previewer definition to use when Print Preview is requested. The Previewer box contains the previewer application that you have defined. See “[Define a New Previewer](#)” on page 294.

**Preview command**

is the command that is used to open an external printer language viewer. For example, if you want Ghostview as your previewer, type `ghostview %s`. When a Preview Command is entered into a Printer definition, the printer definition becomes a previewer definition. The Preview Command must a valid command. When the command is executed as part of the preview process the `%s` are replaced with the name of a temporary file that contains the input for the preview command.

**Note:** The **Previewer** and **Preview Command** fields are mutually exclusive. When you enter a command path into the **Preview Command** field, the **Previewer** box is dimmed.

## How to Specify a Printer for Your Session

The PRINTERPATH= system option enables you to specify a Universal Printer to use for the current SAS session. This printer specification is not retained across SAS sessions. The PRINTERPATH= system option is primarily used in batch mode, when there is no windowing environment in which to set the default printer. This option accepts a character string as its value. For example:

```
options printerpath=myprinter;
options printerpath="Print PostScript to disk";
```

**Note:** If the printer name contains blanks, you must enclose it in quotation marks.

You can get a list of printers that are currently defined from two places:

- The list of printers in the **Printer** field of the Print Setup window.
- Submit this code:

```
proc qdevice out=printers;
  printer _all_;
run;

proc print data=printers;
  var name desc;
  where nametype contains "Printer";
run;
```

You can also override the printer destination by specifying a fileref with the PRINTERPATH= system option:

```
options printerpath= (myprinter printout);
filename printout path;
```

## Printing with Universal Printing

### Print a Test Page

To print a test page, follow these steps:

- 1 Select **File**  $\Rightarrow$  **Print Setup** and choose **Print Test Page** to open the Print Setup window.
- 2 Select the printer for which you would like a test page from **Printer** list view.
- 3 Click **Print Test Page**.

Alternatively, you can issue the DMPRINTSETUP command.

### Print the Contents of an Active SAS Window

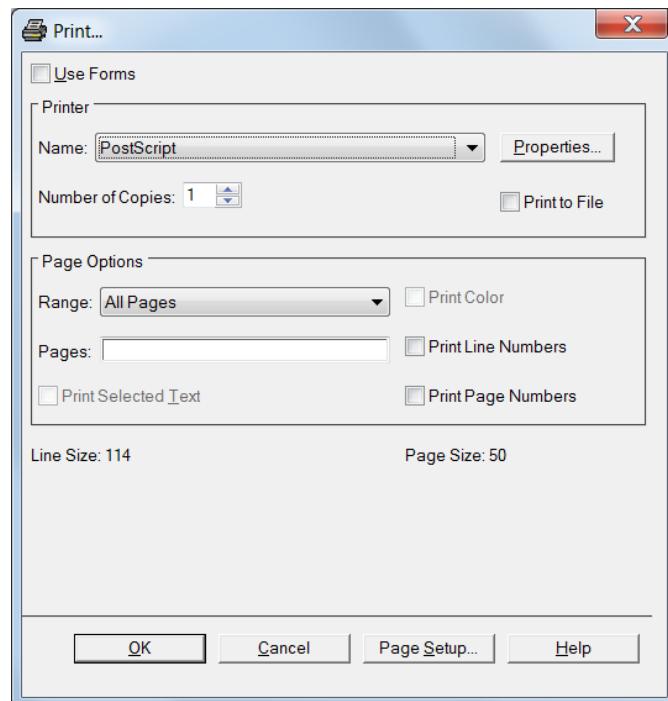
To print the contents of a window in SAS, follow these steps:

- 1 Click inside the window to make it active.
- 2 Select **File**  $\Rightarrow$  **Print**.

A print window appears. Your print window might differ from the window that follows.

Alternatively, you can issue the DMPRINT command.

**Figure 15.15 Print Window**



**3** If the **Use Forms** check box is visible, clear it in order to use Universal Printing.

**4** From the Printer group box, select the name of the printer definition.

**5** Enter the number of copies that you want.

**6** If you want to save your print job to a file, follow these steps:

**a** Select **Print to File**.

**b** Select **OK**. The **File Selection** window appears.

**c** Select an existing file or enter a new filename.

**Note:** If you print to an existing file, the contents of the file are either overwritten or appended, depending on whether you choose **replace** or **append** from the open print window. Most viewers for EMF, GIF, PNG, SVG, and TIFF files do not view appended files. When **append** is selected with a PDF printer, a merged PDF file is not produced.

**7** Set additional printing options.

The fields in the Page Options area provide choices according to the content in the SAS window that you are trying to print. By default, SAS prints the entire contents of the selected window.

**Table 15.7** Page Options

Item to Print	Do This
Selected lines of text in a window <b>Note:</b> not available on z/OS	Select the text that you want to print, and then open the <b>Print</b> window. In the <b>Page Options</b> box, check the <b>Print Selected Text</b> box.
The page that is currently displayed in the window	Select <b>Current page</b> .
A range of pages or other individual pages	Select <b>Range</b> and enter the page numbers in the <b>Pages</b> field. Separate individual page numbers and page ranges with either a comma (,) or a blank. You can enter page ranges in any of these formats: <ul style="list-style-type: none"> <li>■ n–m prints all pages from n to m, inclusive.</li> <li>■ –n prints all pages from page 1 to page n.</li> <li>■ n– prints all pages from page n to the last page.</li> </ul>
In color	Check the <b>Print Color</b> box.
Line numbers	Check the <b>Print Line Numbers</b> box.
Page numbers	Check the <b>Print Page Numbers</b> box.
A graph	Use the DMPRINT command, or select <b>File</b> $\Rightarrow$ <b>Print</b> . Verify that the <b>Use SAS/GRAFH Drivers</b> check box is deselected in order to use Universal Printing.

8 Click **OK** to print.

## Working with Previewers

### Define a New Previewer

Previewers enable you to preview a print job. SAS does not set a default previewer application. To use the Print Preview feature in SAS, you or your system administrator must first define a previewer for your system.

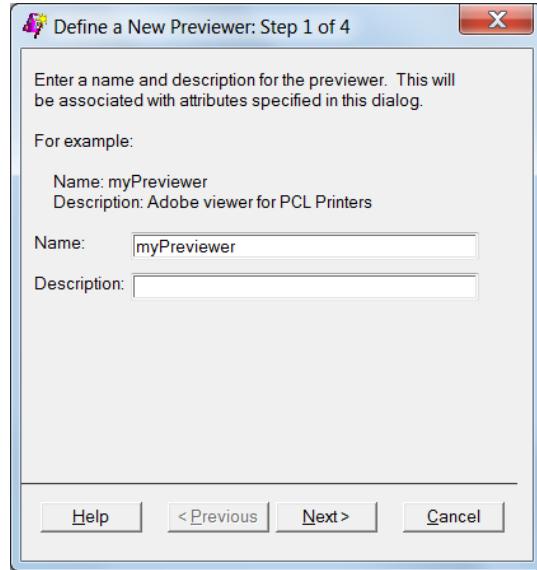
**z/OS Specifics:** Print Previewers are not supported on z/OS.

Previewers can be defined using the New Previewer wizard. To use the New Previewer wizard to define a new print previewer, follow these steps:

1 Issue the DMPRTC CREATE PREVIEWER command.

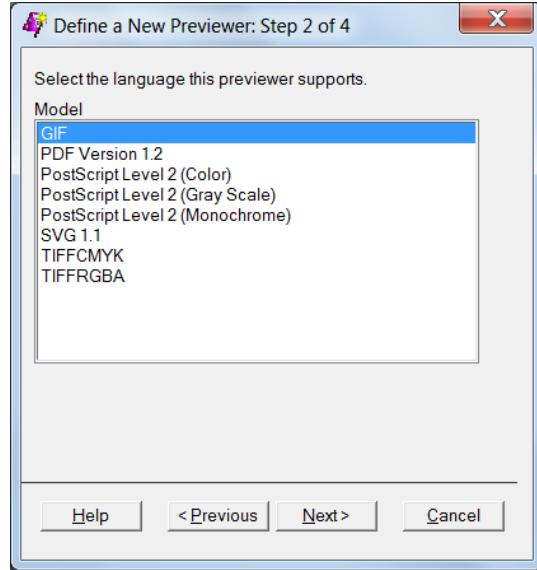
The following window appears:

**Figure 15.16** Previewer Definition Window to Enter Name and Description



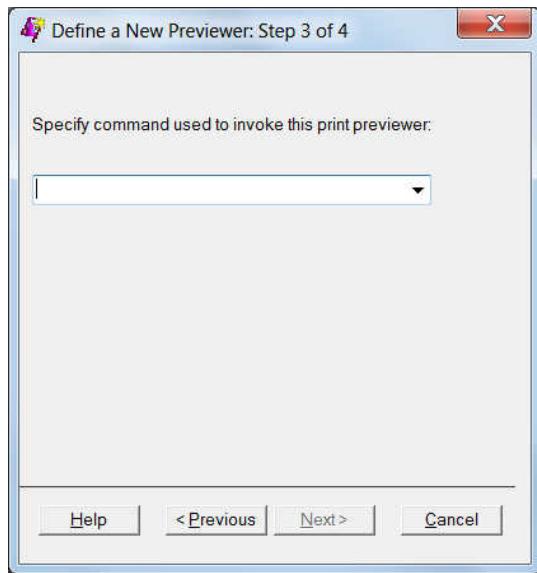
- 2 Enter the name and a description for the new previewer (127-character maximum, no backslashes, not case sensitive).
- The previewer name is required. The description is optional.
- 3 Click **Next** to proceed to Step 2 of the wizard.

**Figure 15.17** Previewer Definition Window to Enter Previewer Language



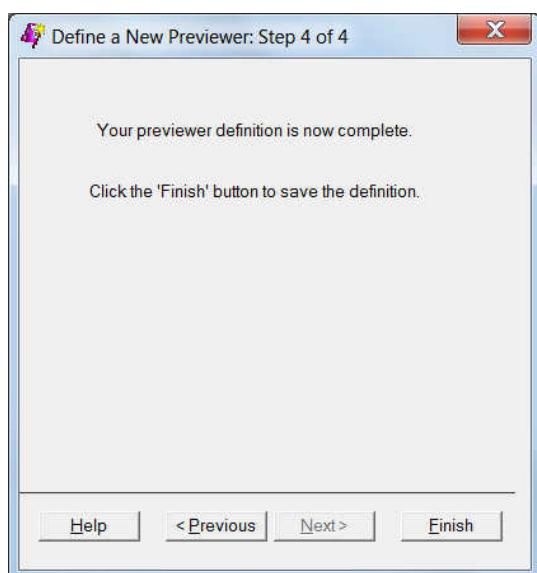
- 4 Select the printer model that you want to associate with your previewer definition.
- The PostScript, PCL, or PDF language generated for the model must be a language that your external viewer package supports. For best results, select the generic models such as PostScript Level 1 (Color) or PCL 5.
- 5 Click **Next** to proceed to Step 3 of the wizard.

**Figure 15.18** Previewer Definition Window to Enter Command to Open Previewer Application



- 6 Enter the command or commands used to open the previewer application, followed by %s where you would normally put the input filename.  
For example, if the command for starting your previewer is “ghostview,” then you would enter `ghostview %s` in the text field.  
You can populate or seed a list of commands used to invoke a print preview application. For more information, see “[Seeding the Print Previewer Command Box](#)” on page 297.  
**Note:** The %s directive can be used as many times as needed in the commands for starting the viewer. However, the start command needs to be the fully qualified command if it is not in the machine’s command path.
- 7 Click **Next** to proceed to Step 4 of the wizard.

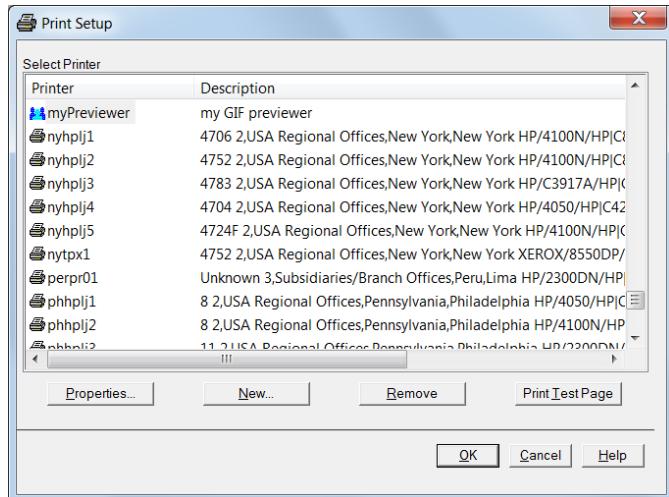
**Figure 15.19** Previewer Definition Window to Complete Process



- 8** Click **Previous** to correct any information. Click **Finish** when you have finished defining your default previewer.

The newly defined previewer displays a previewer icon in the Print Setup window.

**Figure 15.20** Print Setup Window Displaying New Previewer



This previewer application can be tested with the **Print Test Page** button on the Print Setup window.

## Seeding the Print Previewer Command Box

Print Preview is supported by print previewer applications such as Ghostview, gv, and Adobe Reader. The Preview command box that appears in the Previewer Definition wizard ([Figure 15.18 on page 296](#)) and on the **Advanced** tab of the Printer Properties window ([Figure 15.14 on page 291](#)) can be pre-populated or seeded with a list of commands. These commands are used to invoke print previewer applications that are available at your site. Users and administrators can manually update the registry, or define and import a registry file that contains a list of previewer commands. This is an example of a registry file.

```
[CORE\PRINTING\PREVIEW COMMANDS]
"1"="/usr/local/gv %s"
"2"="/usr/local/ghostview %s"
```

## Previewing Print Jobs

You can use the print preview feature if a print viewer is installed for the designated printer. Print Preview is always available from the File menu in SAS. You can also issue the DMPRTPREVIEW command.

## Set Page Properties

You can customize how your printed output appears in the Page Setup window. Depending on which printer you have currently set, some of the Page Setup options that are described in the following steps might be unavailable.

To customize your printed output, follow these steps:

**1 Select File → Page Setup.**

The Page Setup window appears.

Alternatively, you can issue the DMPAGESETUP command.

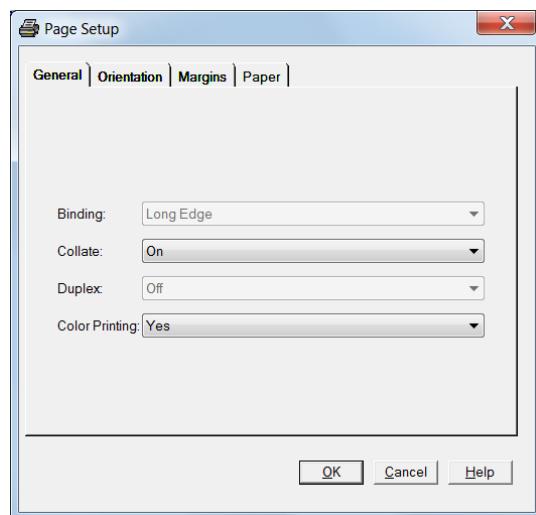
**2 Select a tab to open windows that control various aspects of your printed output.**

Descriptions of the tabbed windows follow.

The Page Setup window consists of four tabs: **General**, **Orientation**, **Margins**, and **Paper**.

- The **General** tab enables you to change the options for **Binding**, **Collate**, **Duplex**, and **Color Printing**.

*Figure 15.21 Page Setup Window Displaying the General Tab*



#### **Binding**

specifies the binding edge (Long Edge or Short Edge) to use with duplexed output. This sets the Binding option.

#### **Collate**

specifies whether the printed output should be collated. This sets the Collate option.

#### **Duplex**

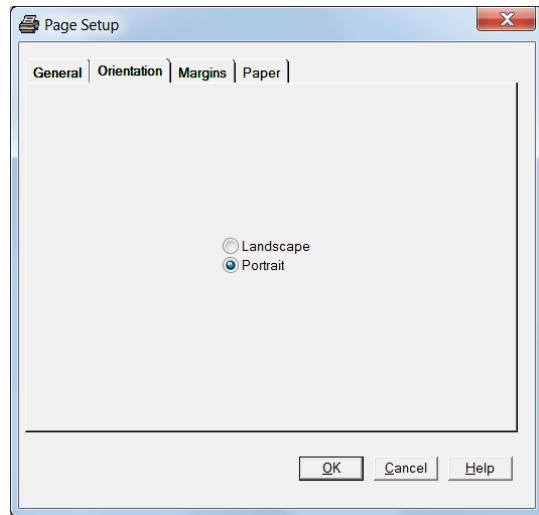
specifies whether the printed output should be single-sided or double-sided. This sets the Duplex option.

#### **Color Printing**

specifies whether output should be printed in color. This sets the COLORPRINTING option.

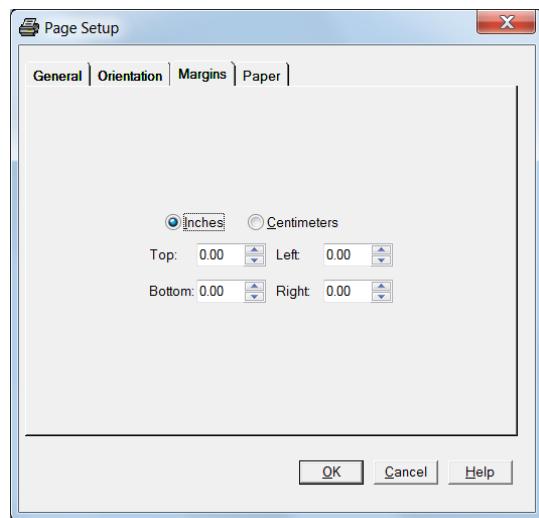
- The **Orientation** tab enables you to change the output's orientation on the page. The default is **Portrait**. This tab sets the ORIENTATION option.

**Figure 15.22** Page Setup Window Displaying the Orientation Tab

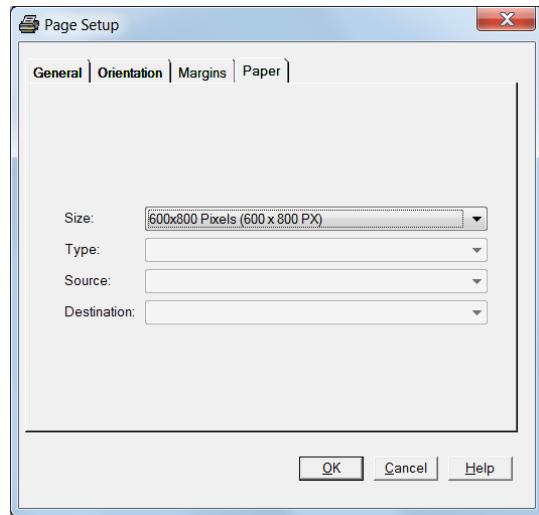


- The **Margin** tab enables you to change the top, bottom, left, and right margins for your pages. The value range depends on the type of printer that you are using. The values that are specified on this tab set the TOPMARGIN, BOTTOMMARGIN, LEFTMARGIN, and RIGHTMARGIN options.

**Figure 15.23** Page Setup Window Displaying the Margins Tab



- The **Paper** tab specifies the Size, Type, Source, and Destination of the paper used for the printed output.

**Figure 15.24** Page Setup Window Displaying Paper Tab**Size**

specifies the size of paper to use by setting the PAPERSIZE option. Paper sizes include Letter, Legal, A4, and so on.

**Type**

specifies the type of paper to use. Examples of choices include Standard, Glossy, and Transparency. This sets the PAPERTYPE option.

**Source**

designates which input paper tray is to be used. This sets the PAPERSOURCE option.

**Destination**

specifies the bin or output paper tray that is to be used for the resulting output. This sets the PAPERDEST option.

**Note:** Page settings are stored in the SAS registry. Although your page settings should remain in effect from one SAS session to another, changing default printers could lose, change, or disable some of the settings. If you change printers during a SAS session, check the Page Setup window to ensure that all of your settings are valid for your new default printer.

---

## System Options That Control Universal Printing

The following system options control Universal Printing.

**Table 15.8** System Options That Control Universal Printing

System Option	Description
BINDING=	Specifies the binding edge for the printer.
BOTTOMMARGIN=	Specifies the size of the margin at the bottom of the page for printing.

System Option	Description
COLLATE	Specifies the collation of multiple copies for output for the printer.
COLORPRINTING	Specifies color printing, if it is supported.
COPIES=	Specifies the number of copies to make when printing.
DUPLEX	Specifies double-sided printing, if it is supported.
LEFTMARGIN=	Specifies the size of the margin on the left side of the page.
ORIENTATION=	Specifies the paper orientation to use (either portrait, landscape, reverse-portrait, or reverse-landscape) for the whole document or for changing the orientation of individual pages in a document.
PAPERDEST=	Specifies the bin or output paper tray to receive printed output.
PAPERSIZE=	Specifies the paper size to use when printing.
PAPERSOURCE=	Specifies the input paper tray to use for printing.
PAPERTYPE=	Specifies the type of paper to use for printing.
PRINTERPATH=	Specifies a printer for Universal Printing print jobs.
RIGHTMARGIN=	Specifies the size of the margin on the right side of the page.
SYSPRINTFONT=	Specifies the default font to use when printing.
TOPMARGIN=	Specifies the size of the margin at the top of the page.

**Note:** The PRINTERPATH= system option specifies which printer is used.

- If the PRINTERPATH= system option is blank, then the default printer is used.
- If the PRINTERPATH= system option is not blank, then Universal Printing is used.

**Note:** In the Windows environment, the default printer is the current Windows system printer or the printer specified by the SYSPRINT system option. Therefore, Universal Printing is not used.

---

# Managing Universal Printers Using the PRTDEF Procedure

---

## About Using the PRTDEF Procedure

Printer definitions can be created for an individual or for all SAS users at a site by using the PRTDEF procedure. The PRTDEF procedure can be used to do many of the same printer management activities that you can do with the Universal Printing windows. The PRTDEF procedure can be used in any execution mode, but it is especially useful if you use SAS in batch mode, where the Universal Printing windows are unavailable.

To define or modify one or more printers with the PRTDEF procedure, you first create a SAS data set that contains variables that correspond to printer attributes. These four variables must be specified for every printer destination:

**DEST**

specifies the printer destination.

**DEVICE**

specifies the device name.

**MODEL**

specifies the name of a printer prototype. For a list of printer prototypes, open the SAS registry to this key: \CORE\PRINTING\PROTOTYPES.

**NAME**

specifies the name of the printer.

For a list of optional variables, see “[Input Data Set Variables That Are Used To Create Printer Definitions](#)” in *Base SAS Procedures Guide*. The PRTDEF procedure reads the data set and converts the variable attributes into one or more printer definitions in the SAS registry.

After you create the printer definition data set, you run the PRTDEF procedure to create the printer.

Only system administrators or others who have Write permission to the Sashelp library can use the PRTDEF procedure to create printer definitions for all SAS users at a site. Individuals have Write permission to their Sasuser library and can use the PRTDEF procedure to create their own printers. However, the printer definition is stored in the Sasuser library and is lost if the Sasuser library is deleted. Printer definitions that are created by individuals are available only when the directory where the printer definition is stored is specified as the Sasuser library. For information about assigning the Sasuser library, see “[SASUSER= System Option](#)” in *SAS System Options: Reference*.

For more information see, “[PRTDEF Procedure](#)” in *Base SAS Procedures Guide*.

---

## Examples of Creating New Printers and Previewers Using the PRTDEF Procedure

### Introduction

These examples show you how to use the PRTDEF procedure to define new printers and to manage your installed printers and previewers.

After a program statement containing the PRTDEF procedure runs successfully, the printers or previewers that have been defined appear in the Print Setup window. A complete set of all available printers and previewers appear in the **Printer name** list. Printer definitions can also be viewed in the Registry Editor window under CORE \PRINTING\PRINTERS.

### Creating a Data Set That Defines Multiple Printers

When you create a data set to use with the PRTDEF procedure to define a printer, you must specify the name, model, device and, destination variables.

See the “[PRTDEF Procedure](#)” in *Base SAS Procedures Guide* in *Base SAS Procedures Guide* for the names of the optional variables that you can also use.

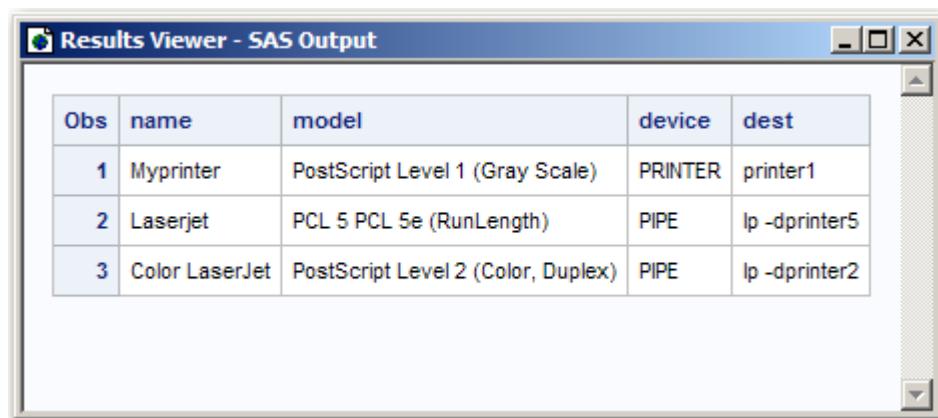
The following code creates a data set to use with the PRTDEF procedure:

```
data printers;
  input name $15. model $35. device $8. dest $14. ;
  datalines;
Myprinter      PostScript Level 1 (Gray Scale)      PRINTER printer1
Laserjet       PCL 5 PCL 5e (RunLength)            PIPE     lp -dprinter5
Color LaserJet PostScript Level 2 (Color, Duplex)  PIPE     lp -dprinter2
;
run;

proc print data=printers;
run;
```

Here is the output:

*Output 15.2 The Printer Data Set*



The screenshot shows a Windows-style window titled "Results Viewer - SAS Output". Inside the window, there is a table with five columns: Obs, name, model, device, and dest. The data is as follows:

Obs	name	model	device	dest
1	Myprinter	PostScript Level 1 (Gray Scale)	PRINTER	printer1
2	Laserjet	PCL 5 PCL 5e (RunLength)	PIPE	lp -dprinter5
3	Color LaserJet	PostScript Level 2 (Color, Duplex)	PIPE	lp -dprinter2

After you create the data set containing the variables, you run the PRTDEF procedure. The PRTDEF procedure creates the printers that are named in the data set by creating the appropriate entries in the SAS registry.

```
proc prtdef data=printers usesashelp replace;
run;
```

The USESASHELP option specifies that the printer definitions are to be placed in the Sashelp library, where they are available to all users. If the USESASHELP option is not specified, then the printer definitions are placed in the current Sasuser library, where they are available to the local user only. The printers that are defined are available only in the local Sasuser directory. However, to use the USESASHELP option, you must have permission to write to the Sashelp library.

The REPLACE option specifies that the default operation is to modify existing printer definitions. Any printer name that already exists is modified by using the information in the printer attributes data set. Any printer name that does not exist is added.

## Creating a Printer for Multiple Users

This example creates a Tektronix Phaser 780 printer definition that specifies to use Ghostview as the preview application and to store the printer definition in the Sashelp library. The bottom margin is set to two centimeters, the font size to 14 point, and the paper size to ISO A4.

```
data tek780;
  name = "Tek780";
  desc = "Test Lab Phaser 780P";
  model = "Tek Phaser 780 Plus";
  device = "PRINTER";
  dest = "testlab3";
  preview = "Ghostview";
  units = "cm";
  bottom = 2;
  fontsize = 14;
  papersiz = "ISO A4";
run;

proc prtdef data=tek780 usesashelp;
run;
```

**Note:** To preview output for this printer, you must create a Ghostview printer definition. You can do this either in the Preview Definition Wizard ([Figure 15.14 on page 291](#)), on the **Advanced** tab of the Printer Properties window ([Figure 15.18 on page 296](#)) or by using the PRTDEF procedure.

Here is a Ghostview printer definition using the PRTDEF procedure:

```
data gsview;
  name = "Ghostview";
  desc = "Print Preview with Ghostview";
  model= "Tek Phaser 780 Plus";
  viewer = 'gv %s';
  device = "dummy";
  dest = " " ;

proc prtdef data=gsview list replace usesashelp;
run;
```

The PROC PRTDEF statement LIST option specifies to write the printer definition to the log.

**Note:** You must specify a preview command either in the Preview Definition Wizard ([Figure 15.14 on page 291](#)) or on the **Advanced** tab of the Printer Properties window ([Figure 15.18 on page 296](#)). An example of a preview command is  
ghostview -bg white -fg black -magstep -2 -nolabel %s

For more information about print previewers see, “[Creating PostScript Previewer Definitions](#)” on page 306.

## Adding, Modifying, and Deleting Printers

This example uses the Printers data set to add, modify, and delete printer definitions. See the “[PRTDEF Procedure](#)” in *Base SAS Procedures Guide* for more variables that you can use to define a printer. The following list describes the variables used in the example:

- The MODEL variable specifies the printer prototype to use when defining this printer.
- The DEVICE variable specifies the type of I/O device to use when sending output to the printer.
- The DEST variable specifies the output destination for the printer.
- The OPCODE variable specifies what action (Add, Delete, or Modify) to perform on the printer definition.
- The first Add operation creates a new printer definition for Color PostScript in the registry and the second Add operation creates a new printer definition for ColorPS in the registry.
- The Mod operation modifies the existing printer definition for LaserJet 5 in the registry.
- The Del operation deletes the printer definitions for printers named “Gray PostScript” and “test” from the registry.

The following example creates a printer definition in the Sashelp library. Because the definition is in Sashelp, the definition becomes available to all users. Special system administration privileges are required to write to the Sashelp library. An individual user can create a personal printer definition by specifying the Sasuser library instead.

```
data printers;
  infile datalines dlm='#';
  length name $ 80
  model $ 80
  device $ 8
  dest $ 80
  opcode $ 3;
  input opcode $ name $ model $ device $ dest $ ;
datalines;
add#  Color PostScript F2#  PostScript Level 2 (Color)#
      DISK#  sasprt.ps
mod#  LaserJet 5#          PCL 5c (DeltaRow)#
      DISK#  sasprtpcl
del#  Gray PostScript#     PostScript Level 2(Gray Scale)#
      DISK#  sasprt.ps
del#  test#                PostScript Level 2 (Color)#
      DISK#  sasprt.ps
add#  ColorPS#             PostScript Level 2 (Color)#
      DISK#  sasprt.ps
;
```

```
proc prtdef data=printers list;
run;
```

**Note:** If the end user modifies and saves new attributes for an administrator-defined printer in the Sashelp library, the printer becomes a user-defined printer in the Sasuser library. Values that are specified by the user override the values that were set by the administrator. If the user-defined printer definition is deleted, the administrator-defined printer reappears.

## Creating PostScript Previewer Definitions

These examples show how to create the Adobe Acrobat Reader print previewer and the Ghostview print previewer in order to preview PDF output in both formats. The variables in the data sets have values that the PRTDEF procedure uses to produce the print previewer definition in the SAS registry.

- The NAME variable specifies the printer name that is associated with the rest of the attributes in the printer definition data record.
- The DESC variable specifies the description of the printer.
- The MODEL variable specifies the printer prototype to use when defining this printer.
- The VIEWER variable specifies the host system commands for print preview.

**Note:** The ghostview %s command needs to be the fully qualified command if it is not in the machine's command path.

**Note:** You must specify a preview command either in the Preview Definition Wizard ([Figure 15.14 on page 291](#)) or on the **Advanced** tab of the Printer Properties window ([Figure 15.18 on page 296](#)). An example of a preview command is ghostview -bg white -fg black -magstep -2 -nolabel %s and c:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe' %s.pdf.

- The DEVICE variable should always be DUMMY.
- DEST should be blank to specify that output is not returned.

The following program creates a print previewer definition for using Adobe Acrobat Reader:

```
data adobeR;
  name = "myAdobeReader";
  desc = "Adobe Reader Print Preview";
  model= "PDF Version 1.2";
  viewer = "'c:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe' %s.pdf";
  device = "dummy";
  dest = " ";
run;
proc prtdef data=adobeR list replace;
run;
```

The following program creates a print previewer definition for using Ghostview:

```
data gsview;
  name = "MyGhostview";
  desc = "Print Preview with Ghostview";
  model= "PostScript Level 2 (Color)";
  viewer = 'ghostview %s';
  device = "dummy";
  dest = " ";
```

```

run;
proc prtdef data=gsview list replace;
run;

```

## Exporting and Backing Up Printer Definitions

The PRTEXP procedure enables you to back up your printer definitions as a SAS data set that can be restored with the PRTDEF procedure.

The PRTEXP procedure has the following syntax.

```
PROC PRTEXP <USESASHELP> <OUT=dataset>
  <SELECT | EXCLUDE> printer_1 printer_2 ... printer_n;
```

The following example shows how to back up four printer definitions (named PDF, postscript, PCL5, and PCL5c) using the PRTEXP procedure:

```
proc prtexp out=printers;
  select PDF postscript PCL5 PCL5c;
run;
```

For more information, see “[PRTEXP Procedure](#)” in *Base SAS Procedures Guide*.

## Sample Values for the Device Type, Destination, and Host Options Fields

The following list provides examples of the printer values for device type, destination, and host options. Because these values can be dependent on each other, and the values can vary by operating environment, several examples are shown. You might want to refer to this list when you are installing a printer or when you change the destination of your output.

- Device Type: Printer
  - z/OS
    - Device type: Printer
    - Destination: (leave blank)
    - Host options: sysout=*class-value* dest=*printer-name*
  - UNIX and Windows
    - Device type: Printer
    - Destination: *printer name*
    - Host options: (leave blank)
- Device Type: Pipe
 

**Note:** A sample command to send output to an lp-defined printer queue on a UNIX host is *lp -ddest*

  - UNIX
    - Device Type: Pipe
    - Destination: *command*
    - Host options: (leave blank)
- Device Type: FTP
 

**Note:** An example of a node name is *pepper.unx*

- z/OS
  - Device type: FTP
  - Destination: ftp.out
  - Host options: host='*nodename*' recfm=vb prompt
  - Device type: Printer
  - Destination: *printer name*
  - Host options: (leave blank)
- Windows
  - Device type: FTP
  - Destination: ftp.out
  - Host options: host='*nodename*' prompt
- UNIX
  - Device type: FTP
  - Destination: filename.ext
  - Host options: host='*nodename*' prompt
- Device Type: Socket
 

**Note:** An example of an lp destination queue is *lp286nc0.prt:9100*

  - UNIX
    - Device type: Socket
    - Destination: *destination-queue*
    - Host options: (leave blank)

## Forms Printing

### Overview of Forms Printing

Before Universal Printing was introduced, SAS provided a utility for print jobs called a form. A form was a standard template that let you control such things as line size and margin information for pages in a print job. Universal Printing is easier to use and has more features than the simple controls offered in forms printing, but SAS still supports forms.

Printing with forms is still available through the Print window. You can switch to forms print mode by selecting **File** ⇒ **Print** and selecting **Use Forms**.

**Note:** Forms printing is not available in batch mode.

---

## Creating or Editing a Form

If your organization has legacy reports that need to be printed using forms, you might have to use the FORM window to create or edit a form. SAS still supports the ability to create or edit forms, though Universal Printing provides more features, and is the recommended method of printing.

You can create or edit a form by entering the FSFORM command:

**FSFORM<catalog-name.>form-name**

If you do not specify a catalog-name, SAS uses the SASUSER.PROFILE catalog. If the form name that you specify does not exist, SAS creates a new form.

If you are creating a new form, SAS displays the Printer Selection frame. If you are editing an existing form, SAS displays the Text Body and Margin Information frame.

To move between the FORMS frames, you can do the following:

- Use the NEXTSCR command to scroll to the next frame and the PREVSCR command to scroll to the previous frame.
- Enter an equal sign (=) and the number of the frame that you want to go to. For example, =1 displays the Text Body and Margin Information frame, and =2 displays the Carriage Control Information frame.
- Select the name of the frame from the **Tools** menu.
- Select **Next Screen** or **Previous Screen** from the **Tools** menu.

You can move between fields on a frame with the TAB key.

After you finish defining or editing your form, issue the END command to save your changes and exit the FORM window.

**Note:** Turning on Forms by checking the **Use Forms** check box in the print window turns Universal Printing off for printing non-graphic windows.

**Operating Environment Information:** For more information about printing with Forms, see the documentation for your operating environment.

---

## Using Fonts with Universal Printers and SAS/GPGRAPH Devices

---

### Rendering Fonts

Universal printing uses the following two methods to generate and display fonts in SAS output.

- the FreeType library
- the font-rendering capabilities of the host

Universal printing supports the following font formats:

- TrueType fonts
- Type1 fonts

**Note:** Universal Printing and SAS/GRAFH do not support double-byte Type1 fonts.

The output methods in the following table are recommended because they use the FreeType library to render fonts. This means that they can render fonts in all of the operating environments that SAS supports.<sup>1</sup>

**Table 15.9 Recommended Devices (because they use the FreeType library to render fonts)**

Output Method	Device
SAS/GRAFH devices	GIF, GIFANIM, HTML, WEBFRAME
	TIFFB, TIFFP, TIFFG3, TIFFB300, TIFFP300
	JPEG
	PCL5, PCL5C, PCL5E
	PDF, PDFC, PDFA
	PNG, PNGT, PNG300
	PSL, PSCOLOR, PSLEPSF, PSLEPSFC
	SVG, SVGT, SVGVIEW, SVGANIM, and SVGZ*
	SASEMF, SASWMF**
	SASPRTC, SASPRTG, SASPRTM printer interface devices

---

1. The FreeType library is used to perform two distinct operations in SAS: measuring the text and rendering the font. Depending on the output devices specified, the FreeType library can perform one or both of these operations. to render fonts.

Output Method	Device
ODS printing and Universal Printing	EMF, EMFDUAL** GIF PCL5, PCL5C, PCL5E PDF, PDFA PNG, PNGT, PNG300 PostScript SVG, SVGT, SVGZ, SVGView, and SVGnotip SVGANIM
ODS RTF	PNG, SASEMF* , EMF
ODS HTML	PNG, PNGT, PNG300, GIF, JPEG, SVG, SVGT*

\* If the NOFONTRENDERING option is set, the device driver uses only the FreeType library for measuring the text. See “[FONTRENDERING= System Option](#)” in *SAS System Options: Reference*

\*\* These devices use the FreeType library only for measuring text. The final font rendering is done by an application such as Microsoft Word, which displays the output using system installed fonts.

You can specify the [QDEVICE](#) procedure to see a list of supported fonts. For a more detailed example, see [Example 5: Generate a Font Report](#).

```
proc qdevice;
run;
```

**Table 15.10** Devices That Use Host Font-Rendering Only

Output Method	Device
SAS/GPGRAPH devices	ACTIVEX, ACTXIMG, JAVA, JAVAIMG <b>Note:</b> These are client devices.
	BMP
	EMF, WMF
	ZGIF

**Table 15.11** Devices That Use Either FreeType Font-Rendering or Host Font-Rendering

Output Method	Device	By Default, uses ...
SAS/GRAFH devices	ZGIF, ZPNG	Host font-rendering
	HTML, WEBFRAME	FreeType font-rendering
	TIFFB, TIFFP, TIFFB300, TIFFP300	FreeType font-rendering
	JPEG	FreeType font-rendering

**Note:** You can set `OPTIONS FONTRENDERING= FREETYPE_POINTS` or `OPTIONS FONTRENDERING=HOST_PIXELS` to change the rendering method for devices that support both the FreeType library and host rendering.

**UNIX Specifics:** With devices that use host rendering in a UNIX operating environment, the TrueType fonts must be installed on the X server that is being used. This is usually specified by the `DISPLAY` environment variable. For more information, see the [Configuration Guide for SAS 9.4 Foundation for UNIX Environments](#).

## The FONTEMBEDDING and FONTRENDERING System Options

### Embedding Fonts in Your Output

Font embedding (using the “[FONTEMBEDDING System Option](#)” in [SAS System Options: Reference](#). ) allows fonts used in the creation of output to travel with that output, ensuring that it is displayed or printed exactly as you intended. Here are some important points to know about font embedding:

- Fonts are included in the output files that are created by the Universal Printer and SAS/GRAFH.
- Output files with embedded fonts do not rely on fonts being installed on the computer that is used to view or print the output file.
- When NOFONTEMBEDDING is set, the output files rely on the fonts being installed on the computer that is used to view or print the font.
- File size is increased for vector output for printers such as PDF and PostScript.
- Not all printers support font embedding. To determine whether the printer that you are using supports font embedding, use the [QDEVICE procedure](#) . If Font Embedding is listed in the SAS log with a value of `Option` or `Always`, then the printer supports font embedding.

```
proc qdevice report=general;
  printer pdf;
run;
```

For more information about the FONTEMBEDDING system option see “[FONTEMBEDDING System Option](#)” in [SAS System Options: Reference](#).

## Measuring Fonts in Pixels or in Points

The [FONTRENDERING system option](#) specifies whether SAS/GPGRAPH devices that are based on the SASGDGIF, SASGDTIF, and SASGDIMG modules render fonts by using the operating system or by using the FreeType engine.

- If the operating system is used (that is, you specify options `fontrendering=host_pixels;`), then font size is requested in pixels.
- If the FreeType engine is used (that is, you specify options `fontrendering=freetype_points;`), then font size is requested in points.

---

## ODS Styles and TrueType Fonts

By default, many SAS/GPGRAPH device drivers and all Universal Printers generate output by using ODS styles, and these ODS styles use TrueType fonts. If no style is specified, the default style is used. If you want the appearance of graphs to be compatible with graphs that were generated prior to SAS 9.2, set the GSTYLE system option to specify NOGSTYLE. For information about the GSTYLE System Option, see “[Using Fonts with Universal Printers and SAS/GPGRAPH Devices](#)” on page 309.

---

## Portability of TrueType Fonts

TrueType fonts are portable across operating environments and are always available in Microsoft Windows environments. A few TrueType fonts are included with some versions of UNIX X Windows.

---

## International Character Support

TrueType fonts support a wide range of international characters. For more information about SAS code that uses TrueType fonts, see “[Examples of Specifying Fonts and Printing International Characters](#)” on page 322.

---

## TrueType Fonts Supplied by SAS

When you install SAS, a number of TrueType fonts are available based on choices that were made during the installation.

TrueType fonts that are supplied by SAS are categorized into four groups:

- Windows Glyph List (WGL) Pan-European character set fonts that are compatible with Microsoft
- graphic symbol
- multilingual

- monolingual Asian

Windows Glyph List (WGL) fonts are also called Pan-European Character Set Fonts. These fonts are about the same shape and size as the Microsoft fonts and can be substituted for the Microsoft fonts without changing formatting or paging. The following table shows the SAS font and the compatible Microsoft font.

**Table 15.12** Windows Glyph List (WGL) and Compatibility with Microsoft

Font Name	Font Description	Compatibility with Microsoft Font
Albany AMT	sans-serif	Arial
Thorndale AMT	serif	Times New Roman
Cumberland AMT	serif fixed <i>(fixed</i> refers to uniform spacing)	Courier New

**Table 15.13** Graphic Symbol TrueType Fonts

Font Name	Font Description	Compatibility with Microsoft Font	Compatibility with Adobe Type1 Font
Symbol MT	192 symbols	Symbol	Symbol
Monotype Sorts*	205 Wingdings characters	Not applicable*	Not applicable*
Arial Symbol	42 symbols**	Not applicable	Not applicable
Arial Symbol Bold	42 symbols**	Not applicable	Not applicable
Arial Symbol Bold Italic	42 symbols**	Not applicable	Not applicable
Arial Symbol Italic	42 symbols**	Not applicable	Not applicable
Times New Roman Symbol	42 symbols**	Not applicable	Not applicable
Times New Roman Symbol Bold	42 symbols**	Not applicable	Not applicable
Times New Roman Symbol Bold Italic	42 symbols**	Not applicable	Not applicable
Times New Roman Symbol Italic	42 symbols**	Not applicable	Not applicable

\* SAS Monotype Sorts is an ornamental font consisting of shapes, symbols, and decorative glyphs that have no one-to-one mapping to Microsoft TrueType or Adobe Type1 fonts. However, the SAS Monotype Sorts font closely resembles Microsoft "Wingdings" TrueType and Adobe "ITC Zapf Dingbats" Type1 fonts.

\*\* These fonts have special glyphs for the Latin characters 0, <, =, C, D, L, M, N, P, R, S, U, V, W, X, Z, and a-z. All other characters are undefined and might be rendered as a rectangle. For example, in

the HTML destination, the rectangle is replaced with the matching Latin1 character when it is displayed in Internet Explorer.

**Table 15.14 Multilingual TrueType Fonts**

Font Name	Language Supported	Font Description
Arial Unicode MS*	Arabic, Armenian, Basic Latin, Bengali, Bopomofo, Cyrillic, Devanagari, Georgian, Greek and Coptic, Gujarati, Gurmukhi, hangul jamo, Hebrew, hiragana, Kanbun, Kannada, katakana, Lao, Malayalam, Oriya, Tamil, Telugu, Oriya, Tamil, Telugu, Thai Tibetan.	sans-serif
Times New Roman Uni*	Arabic, Basic Latin, Bopomofo, Cyrillic, Devanagari, Georgian, Greek and Coptic, Gujarati, hangul jamo, Hebrew, hiragana, Kanbun, katakana, Lao, Mongolian, Tamil, Telugu, Thai, Tibetan.	serif

\* In SAS 9.4, the Arial Unicode MS and Times New Roman fonts replace the Monotype Sans WT and Thorndale Duospace WT fonts.

In SAS 9.4M5, the following new *AvenirNextforSAS* replaces the Avenir Next fonts that were added in a previous maintenance release.

New Font	Replaces
AvenirNextforSAS	These replace the Avenir Next Fonts that were added in SAS 9.4M3.
AvenirNextforSASItalic	
AvenirNextforSASBold	
AvenirNextforSASBoldItalic	
AvenirNextforSASLight	
AvenirNextforSASLightItalic	

In SAS 9.4M5, the following new *HelveticaNeueforSAS* replace the Helvetica fonts that were added in a previous maintenance release.

New Font	Replaces
HelveticaNeueforSAS	These replace the Helvetica LT Pro Fonts that were added in SAS 9.4M4.
HelveticaNeueforSASItalic	
HelveticaNeueforSASBold	
HelveticaNeueforSASBoldItalic	
HelveticaNeueforSASLightItalic	
HelveticaNeueforSASLight	

**Table 15.15** Monolingual Asian TrueType Fonts

Language Supported	Font Name	Character Set
Japanese	MS Gothic, MS UI Gothic, MS PGothic	Shift JIS
	MS Mincho, MS PMincho	Shift JIS
Korean	Gulim, GulimChe, Dotum, DotumChe	KSC5601
	Batang, BatangChe, Gungsuh, GungsuhChe	KSC5601
Simplified Chinese	MYingHei_18030_C-Medium	GB18030 and GB2312
	MYingHei_18030_C-MediumHWL	
	CSongGB18030C-Light	GB18030 and GB2312
	CSongGB18030C-LightHWL	
Traditional Chinese*	HeiT	Big5
	MingLiU, MingLiU_HKSCS, PMingLiU	Big5

\* HeiT, MingLiU, MingLiU\_HKSCS, and PMingLiU support HKSCS2004 (Hong Kong Supplemental Character Set) characters.

The fonts that are supplied by SAS and the fonts that are already installed on Windows are automatically registered in the SAS registry when you install SAS. Fonts already installed on UNIX and z/OS must be registered manually in the SAS registry after you install SAS. To register other TrueType Fonts, see “[Registering Fonts](#)” on page 317.

---

## Registering Fonts

### Fonts Supported by SAS

In addition to the TrueType fonts that come installed with SAS, SAS supports PostScript Type1 fonts. The following table shows the font prefix and file extension for TrueType and Type1 fonts:

*Table 15.16 Supported Font Types*

Type	Tag	File Extension
TrueType	<ttf>	.ttf
Type1	<at1>	.pfb *

\* Data from a .pfm file is used to generate output using the SAS/GRAFH SASEMF and SASWMF devices on Windows. On UNIX and z/OS, data from a .pfm file is used to generate output using the WMF device and the EMF universal printer. This file is not required to register Type1 fonts using PROC FONTREG. If you do not register a .pfm file, you might not have the desired results.

### Registering Fonts with the SAS Registry

To use TrueType or Type1 fonts that are not registered when SAS is installed, use the FONTREG procedure to register these fonts in the SAS registry. For more information, see “[FONTREG Procedure](#)” in *Base SAS Procedures Guide*.

**Note:** The fonts that are supplied by SAS and the fonts that are installed by Microsoft are automatically registered in the SAS registry when you install SAS. Fonts that are installed after you install SAS must be registered manually in the SAS registry.

### Registering Fonts for UNIX, Windows, or the z/OS HFS File System

Submit the following SAS program. The FONTPATH statement specifies the directory that contains the fonts and `pathname` is the directory path of the fonts.

```
proc fontreg;
  fontpath 'pathname';
run;
```

For more information about adding fonts to the SAS Registry, see “[FONTREG Procedure](#)” in *Base SAS Procedures Guide*.

**Note:** In Microsoft Windows environments, TrueType fonts are usually located in either the `C:\WINNT\Fonts` or `C:\Windows\Fonts` directory. For all other operating environments, contact your system administrator for the location of the TrueType font files.

For more information, see “[FONTREG Procedure](#)” in *Base SAS Procedures Guide*.

## Registering Fonts for z/OS

On z/OS systems that do not use the HFS file system, you can use the FONTFILE statement instead of the FONTPATH statement to specify the fixed block sequential file that contains a font. Because the default value of MODE= option is ALL, the PROC statement below adds new fonts that do not already exist in the SAS registry and replaces existing fonts.

```
proc fontreg;
  fontfile 'filename';
run;
```

**z/OS Specifics:** When you add fonts to a z/OS system, the font file must be allocated as a sequential data set with a fixed block record format and a record length of 1.

For more information, see “[FONTREG Procedure](#)” in *Base SAS Procedures Guide*.

## Listing the Registered Fonts for a Device

You can use the QDEVICE procedure to view the list of fonts that have been registered in the SAS registry, including fonts that you registered with the FONTREG procedure. You can submit the following program to view fonts for a device or universal printer.

```
/* Macro FONTLIST - Report fonts supported by a device */

%macro fontlist(type, name);
proc qdevice report=font out=fonts;
  &type &name;
  var font ftype fstyle fweight;
run;

data;
  set fonts;
  drop ftype;
  length type $16;
  if ftype = "System"
    then do;
      if substr(font,2,3) = "ttf" then type = "TrueType";
      else if substr(font,2,3) = "at1" then type = "Adobe Type1";
      else if substr(font,2,3) = "cff" then type = "Adobe CFF/Type2";
      else if substr(font,2,3) = "pfr" then type = "Bitstream PFR";
      else type = "System";
      if type ^= "System" then font = substr(font,7,length(font)-6);
      else if substr(font,1,1) = "@" then font = substr(font, 2,length(font)-1);
    end;
    else type = "Printer Resident";
run;

proc sort;
  by font;
run;
```

```

title "Fonts Supported by the %upcase(&name) &type";

proc print label;
  label fstyle="Style" fweight="Weigth" font="Font" type="Type";
run;

%mend fontlist;

%fontlist(printer, pdf)
%fontlist(device, pdf)
%fontlist(device, win)
%fontlist(printer, png)
%fontlist(device, pcl5c)

```

Here is the output for the first 25 fonts in the output data set::

*Output 15.3 List of Fonts Supported by the PDF Printer (Partial Output)*

<b>Fonts Supported by the PDF printer</b>				
<b>Obs</b>	<b>Font</b>	<b>Style</b>	<b>Weigth</b>	<b>Type</b>
<b>1</b>	Adobe Caslon	Italic	Bold	Printer Resident
<b>2</b>	Adobe Caslon	Italic	Normal	Printer Resident
<b>3</b>	Adobe Caslon	Italic	Semi Bold	Printer Resident
<b>4</b>	Adobe Caslon	Roman	Bold	Printer Resident
<b>5</b>	Adobe Caslon	Roman	Normal	Printer Resident
<b>6</b>	Adobe Caslon	Roman	Semi Bold	Printer Resident
<b>7</b>	Adobe Caslon Oldstyle	Italic	Bold	Printer Resident
<b>8</b>	Adobe Caslon Oldstyle	Italic	Normal	Printer Resident
<b>9</b>	Adobe Caslon Oldstyle	Italic	Semi Bold	Printer Resident
<b>10</b>	Adobe Caslon Oldstyle	Roman	Bold	Printer Resident
<b>11</b>	Adobe Caslon Oldstyle	Roman	Normal	Printer Resident
<b>12</b>	Adobe Caslon Oldstyle	Roman	Semi Bold	Printer Resident
<b>13</b>	Adobe Caslon Small Caps	Roman	Semi Bold	Printer Resident
<b>14</b>	Adobe Caslon Small Caps Oldstyle	Roman	Normal	Printer Resident
<b>15</b>	Adobe Garamond	Italic	Bold	Printer Resident
<b>16</b>	Adobe Garamond	Italic	Normal	Printer Resident
<b>17</b>	Adobe Garamond	Italic	Semi Bold	Printer Resident
<b>18</b>	Adobe Garamond	Roman	Bold	Printer Resident
<b>19</b>	Adobe Garamond	Roman	Normal	Printer Resident
<b>20</b>	Adobe Garamond	Roman	Semi Bold	Printer Resident
<b>21</b>	Adobe Garamond Oldstyle	Italic	Bold	Printer Resident
<b>22</b>	Adobe Garamond Oldstyle	Italic	Normal	Printer Resident
<b>23</b>	Adobe Garamond Oldstyle	Roman	Bold	Printer Resident
<b>24</b>	Adobe Garamond Oldstyle	Roman	Normal	Printer Resident
<b>25</b>	Adobe Garamond Small Caps Oldstyle	Roman	Normal	Printer Resident

For more information, see “[QDEVICE Procedure](#)” in *Base SAS Procedures Guide*.

## Using Fonts

### Specifying Fonts with the Display Manager

After you update the SAS Registry, the newly registered fonts are available for use within SAS. To access the fonts interactively when Universal Printing is enabled, follow these steps:

- 1** Select **File**  $\Rightarrow$  **Print**.
- 2** Select an appropriate printer, such as PDF or PCL5.
- 3** Click the **Properties** button.
- 4** Click the **Font** tab.

This window contains drop-down boxes for Font, Style, Weight, Size (in points), and Character Set.

- 5** Click the arrow on the right side of the **Font** box and scroll through the list of available fonts.

TrueType fonts are indicated by the letters `ttf` enclosed in angle brackets (`<>`), and Type1 fonts are indicated by the letters `at1` enclosed in angle brackets (`<>`). For example, the TrueType font Albany AMT is listed as `<ttf> Albany AMT` and the Type1 Font Times is listed as `<at1> Times`. The three-character tag enclosed in angle brackets makes the distinction between the different types of fonts with the same name, such as `<ttf> Symbol` and a `Symbol` font that resides on a physical printer. Fonts that do not have a `<ttf>` tag or an `<at1>` tag reside in the printer's memory. To ensure that you are using SAS fonts when you specify a font that has different types, use only the font syntax with the angled brackets. For example, you can specify the `Symbol` font as follows: `<ttf> Symbol`.

- 6** Select the font that you want to use.
- 7** Click **OK** to return to the Print dialog box.
- 8** Click **OK** to create your output.

### Specifying Fonts with SAS Program Statements

You can specify a font in the TITLE statement. For example, if you want to use the TrueType font Albany AMT in a TITLE statement, include the following line of code in your SAS program.

```
Title1 f="Albany AMT" "Text in Albany AMT";
```

You can also specify attributes such as style or weight in the TITLE statement by using the forward slash (/) as a delimiter.

```
Title1 f="Albany AMT/Italic/Bold" "Text in Bold Italic Albany AMT";
```

For ODS templates, the attributes are specified after the text size parameter. See “[Specifying a Font with PROC PRINT and a User-Defined ODS Template](#)” on page 324 for a complete example.

**Note:** You should use the <ttf> tag only when it is necessary (for example, to distinguish between a TrueType font and another type of font with the same name).

## Specifying a Font with the SYSPRINTFONT Option

The SYSPRINTFONT= system option sets the default font that you want to use for printing from windows such as the Program Editor, the Log, and Output windows. For example, you could use the SYSPRINTFONT= system option to print your output in the Albany AMT font by submitting the following OPTIONS statement.

```
options sysprintfont="Albany AMT";
```

You can also use the SYSPRINTFONT= system option to specify the weight and size of a font. For example, the following code specifies an Arial font that uses bold face, is italicized, and has a size of 14 points.

```
options sysprintfont="Arial" bold italic 14;
```

You can override the default font by explicit font specifications or ODS styles.

For more information, see the “[SYSPRINTFONT= System Option](#)” in *SAS System Options: Reference*.

## Slanting and Emboldening Fonts

Some font families do not have italic or bold fonts. If you specify italic or bold on these fonts, SAS will automatically generate the font as slanted or emboldened on universal printers that support this feature. The following universal printers support font slanting and emboldening:

**Table 15.17** Universal Printers That Support Font Slanting and Emboldening

GIF	PCL
TIFF	PNG
SVG*	

\*Font slanting and emboldening is not supported on Internet Explorer and Firefox. However, it is supported on Chrome, Opera, and Safari browsers.

The following universal printers do not support font slanting and emboldening:

- PDF
- EMF
- PostScript

## Changing the Slant Factor

You can adjust the degree of slanting on fonts that support glyph slanting. These are typefaces that do not already have italic style fonts. True italic fonts are not subject

to font slanting. To change the slant factor for all universal printers, follow these steps:

- open the Registry Editor by entering `regedit` in the command bar or by selecting **Tools**  $\Rightarrow$  **Options**  $\Rightarrow$  **Registry Editor** from the Application Toolbar.
- in the SAS Registry panel of the Registry Editor window, expand the `CORE/PRINTING/FREETYPE` folder
- right-click and choose **New Double Value** from the pop-up menu
- enter `SlantFactor` in the **Value Name** field of the Edit Double Value window
- enter the desired slant factor value in the **Value Data** field. The default is value `.25`.

To change the slant factor for a specific font, perform the following tasks using the SAS Registry Editor:

- in the left SAS Registry panel of the SAS Registry Editor, expand the `CORE/PRINTING/FREETYPE/FONTS/<ttf>font-name/Attributes` folder
- right-click and choose **New Double Value** from the pop-up menu
- enter `SlantFactor` in the **Value Name** field of the Edit Double Value window
- enter the desired slant factor value in the **Value Data** field. The default is value `.25`.

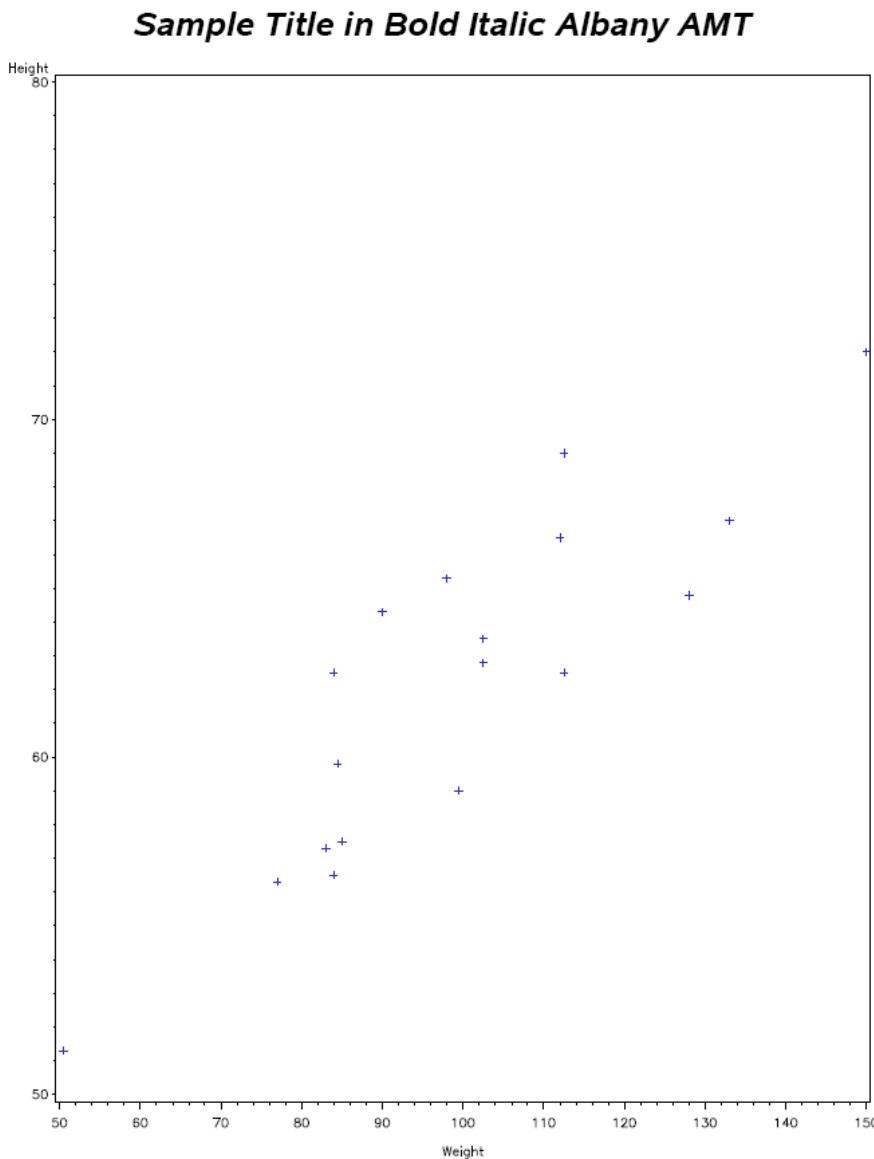
## Examples of Specifying Fonts and Printing International Characters

### Specifying a Font with SAS/GRAFH

The following example creates an output file, `sasprt.pdf`, with a title in the Albany AMT font that uses bold face and is italicized.

```
options printerpath=pdf device=sasprtc;
ods printer;
title1 color=black f="Albany AMT/Italic/Bold" "Sample Title in Bold Italic Albany AMT";
proc gplot data=sashelp.class;
plot height*weight;
run;
quit;
ods printer close;
```

**Note:** The DEVICE= option defaults to either SASPRTM, SASPRTG, or SASPRTC, depending on the type of printer. If PRINTERPATH=PCL5, which is a monochrome printer, ODS PRINTER defaults to SASPRTM.

**Figure 15.25** GPLOT Output

## Specifying a Font with PROC PRINT

The following example produces an output file `.print1.pdf`, with the titles in the Albany AMT, Thorndale AMT, and Cumberland AMT fonts.

```

filename new 'print1.pdf';
options printerpath=(PDF new) device=sasprtc obs=5;
ods printer;
proc print data=sashelp.class;
  title1 f='Albany AMT' h=2 'TrueType Albany AMT';
  title2 f='Thorndale AMT' h=3 'Thorndale AMT';
  title3 f='Cumberland AMT' 'Cumberland AMT ';
run;
ods printer close;

```

Figure 15.26 PDF Output Using PROC PRINT

**TrueType Albany AMT**  
**Thorndale AMT**  
**Cumberland AMT**

Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5

## Specifying a Font with PROC PRINT and a User-Defined ODS Template

The following example creates a template of font styles and then produces a PDF file.

```

filename out 'print2.pdf';

options printerpath=(pdf out) device=sasprtc;
proc template;
  define style New_style / store = SASUSER.TEMPLAT;
  parent = styles.printer;
  style fonts /
    'docFont'           = ("Cumberland AMT", 12pt)
    'headingFont'        = ("Albany AMT", 10pt, bold)
    'headingEmphasisFont' = ("Albany AMT", 10pt, bold italic)
    'TitleFont'          = ("Albany AMT", 12pt, italic bold)
    'TitleFont2'         = ("Albany AMT", 11pt, italic bold)
    'FixedFont'          = ("Cumberland AMT", 11pt)
    'BatchFixedFont'     = ("Cumberland AMT", 6pt)
    'FixedHeadingFont'   = ("Cumberland AMT", 9pt, bold)
    'FixedStrongFont'    = ("Cumberland AMT", 9pt, bold)
    'FixedEmphasisFont'  = ("Cumberland AMT", 9pt, italic)
    'EmphasisFont'       = ("Albany AMT", 10pt, italic)
    'StrongFont'         = ("Albany AMT", 10pt, bold);
  end;
run;

ods printer style=New_style;
proc print data=sashelp.class;
  title1 'Proc Print';
  title2 'Templated ODS output';
run;
ods printer close;

```

**Figure 15.27 PDF Output Using PROC PRINT and a User-Defined ODS Template**

***Proc Print***  
***Templated ODS output***

<b>Obs</b>	<b>Name</b>	<b>Sex</b>	<b>Age</b>	<b>Height</b>	<b>Weight</b>
<b>1</b>	Alfred	M	14	69.0	112.5
<b>2</b>	Alice	F	13	56.5	84.0
<b>3</b>	Barbara	F	13	65.3	98.0
<b>4</b>	Carol	F	14	62.8	102.5
<b>5</b>	Henry	M	14	63.5	102.5

## Printing International Characters

### Example 1

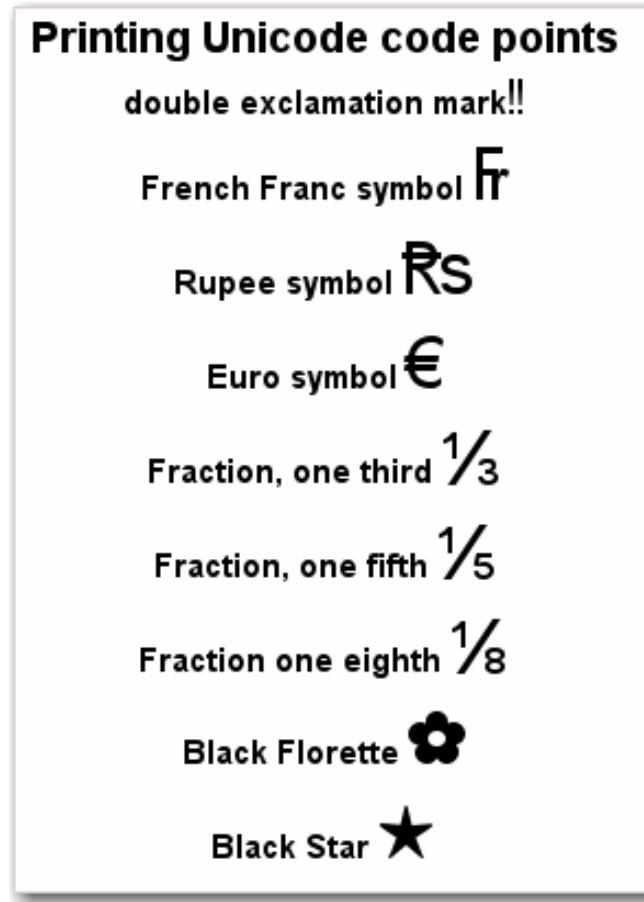
The following example produces an output file, **titles.png**. It is printed ten Unicode characters.

```

filename new 'titles.png';
options printerpath=(png new) device=sasprtc;
ods printer;

proc gslide;
  title1 "Printing Unicode code points";
  title2 "double exclamation mark" f="Arial Unicode MS/Unicode" h=2 '203C'x;
  title3 "French Franc symbol " f="Arial Unicode MS/Unicode" h=3 '20A3'x;
  title4 "Lira symbol " f="Arial Unicode MS/Unicode" h=3 '20A4'x;
  title4 "Rupee symbol " f="Arial Unicode MS/Unicode" h=3 '20A8'x;
  title5 "Euro symbol " f="Arial Unicode MS/Unicode" h=3 '20AC'x;
  title6 "Fraction, one third " f="Arial Unicode MS/Unicode" h=3 '2153'x;
  title7 "Fraction, one fifth " f="Arial Unicode MS/Unicode" h=3 '2155'x;
  title8 "Fraction one eighth " f="Arial Unicode MS/Unicode" h=3 '215B'x;
  title9 "Black Florette " f="Arial Unicode MS/Unicode" h=3 '273F'x;
  title10 "Black Star " f="Arial Unicode MS/Unicode" h=3 '2605'x;
run;
quit;
ods printer close;
```

Figure 15.28 Output of Ten Unicode Characters

**Example 2**

The following example produces an output file, `utf8.gif`. It must be run with a UTF-8 server and requires a TrueType font that contains the characters that are used. The table of character names and the associated codes can be found on the Unicode website at <http://www.unicode.org/charts>.

```
proc template;
  define style utf8_style / store = SASUSER.TEMPLAT;
  parent = styles.printer;
  style fonts /
    'docFont'           = ("Arial Unicode MS", 12pt)
    'headingFont'        = ("Arial Unicode MS", 10pt, bold)
    'headingEmphasisFont' = ("Arial Unicode MS", 10pt, bold italic)
    'TitleFont'          = ("Arial Unicode MS", 12pt, italic bold)
    'TitleFont2'         = ("Arial Unicode MS", 11pt, italic bold)
    'FixedFont'          = ("Times New Roman Uni", 11pt)
    'BatchFixedFont'     = ("Times New Roman Uni", 6pt)
    'FixedHeadingFont'   = ("Times New Roman Uni", 9pt, bold)
    'FixedStrongFont'    = ("Times New Roman Uni", 9pt, bold)
    'FixedEmphasisFont'  = ("Times New Roman Uni", 9pt, italic)
    'EmphasisFont'       = ("Arial Unicode MS", 10pt, italic)
    'StrongFont'         = ("Arial Unicode MS", 10pt, bold);
  end;
run;
```

```
%macro utf8chr(ucs2);
  kcvt(&ucs2, 'ucs2b', 'utf8');
%mend utf8chr;
%macro namechar(name, char);
  name=&name"; code=upcase("&char"); char=%utf8chr("&char"x); output;
%mend namechar;

data uft8char;
length name $40;
%namechar(Registered Sign, 00AE);
%namechar(Cent Sign, 00A2);
%namechar(Pound Sign, 00A3);
%namechar(Currency Sign, 00A4);
%namechar(Yen Sign, 00A5);
%namechar(Rupee Sign, 20A8);
%namechar(Euro Sign, 20Ac);
%namechar(Dong Sign, 20Ab);
%namechar(Euro-currency Sign, 20A0);
%namechar(Colon Sign, 20A1);
%namechar(Cruzeiro Sign, 20A2);
%namechar(French Franc Sign, 20A3);
%namechar(Lira Sign, 20A4);
run;

options printerpath=(gif out) device=sasprtc;
filename out 'utf8.gif';

ods printer style=utf8_style;
proc print;
run;
ods printer close;
```

**Note:** If you get an “unable to write to template store” error when running this code, place the following before the proc step:

```
PROC TEMPLATE step:
  ODS PATH work.templat(update) sasuser.templat(read)
    sashelp.tmplmst(read);
```

This statement causes the templates to be written to the WORK library where the server has Read and Write access.

Figure 15.29 Output of International Characters

Obs	name	code	char
1	Registered Sign	00AE	®
2	Cent Sign	00A2	¢
3	Pound Sign	00A3	£
4	Currency Sign	00A4	¤
5	Yen Sign	00A5	¥
6	Rupee Sign	20A8	₹
7	Euro Sign	20AC	€
8	Dong Sign	20AB	₫
9	Euro-currency Sign	20A0	₠
10	Colon Sign	20A1	₡
11	Cruzeiro Sign	20A2	₲
12	French Franc Sign	20A3	₣
13	Lira Sign	20A4	₺

---

## Creating EMF (Enhanced Metafile Format) Graphics Using Universal Printing

---

### EMF Graphics in SAS

Enhanced Metafile Format (EMF) graphics are scalable vector graphics that produce true color graphics. Applications that support EMF graphics run on Windows. The default output size of 800x600 pixels and the default resolution of 96 dpi produce output that closely resembles the screen resolution. EMF graphics are device-independent and are rendered by an EMF viewer.

Universal Printing supports three EMF metafile formats, EMF, EMFPlus, and EMFDual. The following table shows the EMF Universal Printers and their corresponding EMF metafile formats:

<b>EMF Universal Printer</b>	<b>Type of EMF Metafile Format</b>	<b>Description</b>
EMF	EMFPlus	<p>The EMF Universal Printer creates a graphic that uses commands, objects, and properties in EMFPlus records. EMFPlus records have a different structure and use different commands, objects, and properties than those in EMF records. The output .emf file contains no EMF records, only EMFPlus records that are embedded within comment records.</p> <p>The EMFPlus Universal Printer supports only TrueType fonts. If you specify another type of font, the font is mapped to the TrueType font.</p>
EMFDual	EMFDual	<p>The EMFDual Universal Printer creates both EMF and EMFPlus graphics in the same output file. The file that is presented is determined by the support of the EMF viewer.</p> <p>Output from the EMFDual printer is large because it contains both EMF and EMFPlus formats.</p>
SASEMF	EMF	<p>The SASEMF Universal Printer creates EMF records that contain drawing commands, object definitions, and properties. Alpha channel color support is available only for images, not vector graphics. The SASEMF Universal Printer is equivalent to the EMF Universal Printer in previous releases of SAS.</p>

EMFDual and SASEMF Universal Printers support TrueType and Type1 fonts. The EMF Universal Printer supports only TrueType fonts. Use the FONTREG procedure to register Fonts. If you specify another type of font when the Universal Printer is EMF, the font is mapped to the TrueType font.

Compression and font embedding are not supported.

For a description of the EMF printers, submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer emf-universal-printer;
run;
```

EMF printers do not support multiple page documents. If a procedure creates multiple pages or if more than one procedure is used in the code for ODS PRINTER output, only the first page is viewable.

## See Also

[“Color Support for Universal Printers” on page 273](#)

---

## Creating an EMF Graphic

You can create a stand-alone EMF graphic using the ODS PRINTER statement. Specify the EMF Universal Printer either as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement. The following sample code specifies the EMF Universal Printer as the value of the PRINTER= option in the ODS PRINTER statement:

```
ods html close;
ods printer printer=emf;

...more SAS code...

ods printer close;
ods html;
```

SAS creates the file sasprt.emf in the current directory.

---

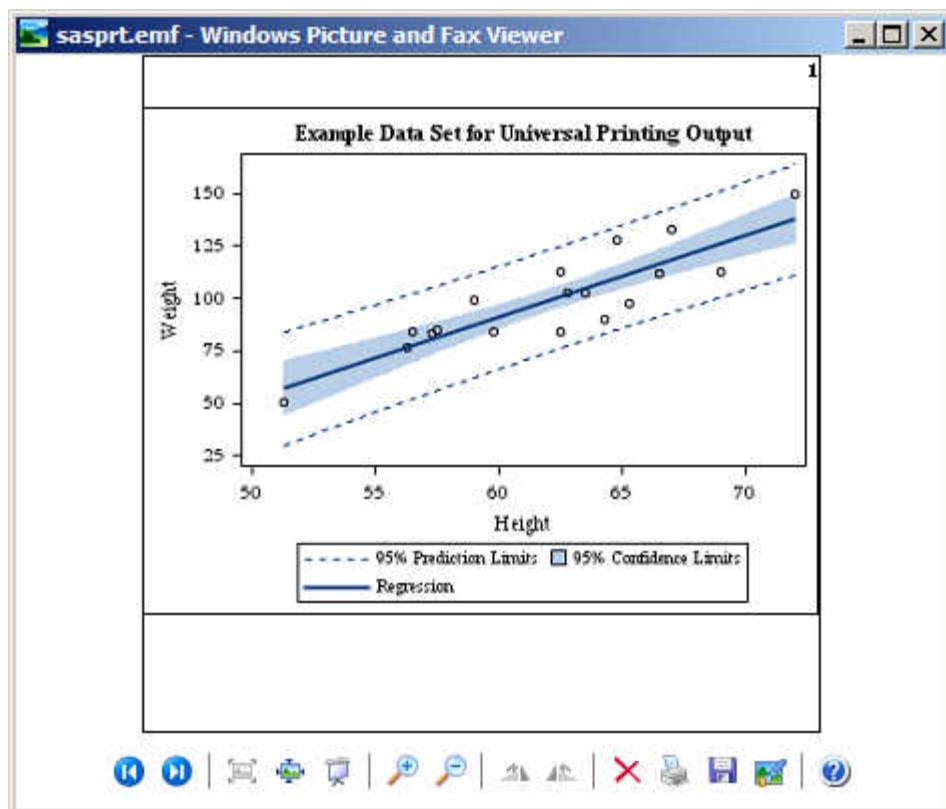
## Example of Creating an EMF Graphic Using the ODS PRINTER Statement

Using the example data set Sashelp.Class and the SGLOT procedure, the following ODS PRINTER statement prints the EMF file sasprt.emf in the current directory:

```
options printerpath=emf papersize=("4in" "4in") nodate;
ods html close;
ods printer;
proc sgplot data=sashelp.class;
    reg x=height y=weight / CLM CLI;
run;
ods printer close;
ods html;
```

The following output is the EMF metafile displayed in the Windows Picture and Fax Viewer:

Figure 15.30 Sashelp.Class in Enhanced Metafile Format



## Creating GIF Images Using Universal Printing

### GIF Images in SAS

The Graphic Interchange Format (GIF) is an image format that has been used extensively on the web. The GIF printer supports RGBA colors, animation, transparency, and renders fonts using the FreeType engine. The default output size of 800x600 pixels. For a description of the GIF printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer gif;
run;
```

The GIF printer does not support multiple page documents. If a procedure creates multiple pages or if more than one procedure is used in the code for ODS PRINTER output, only the first page is viewable.

## See Also

- “Color Support for Universal Printers” on page 273
- “Creating Animated GIF Images and SVG Documents” on page 375

## Creating a GIF Image

You can create GIF images using the ODS PRINTER statement. You specify the GIF Universal Printer either as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement. The following sample code specifies the GIF Universal Printer as the value of the PRINTER= option in the ODS PRINTER statement:

```
ods html close;
ods printer printer=gif;

...more SAS code...

ods printer close;
ods html;

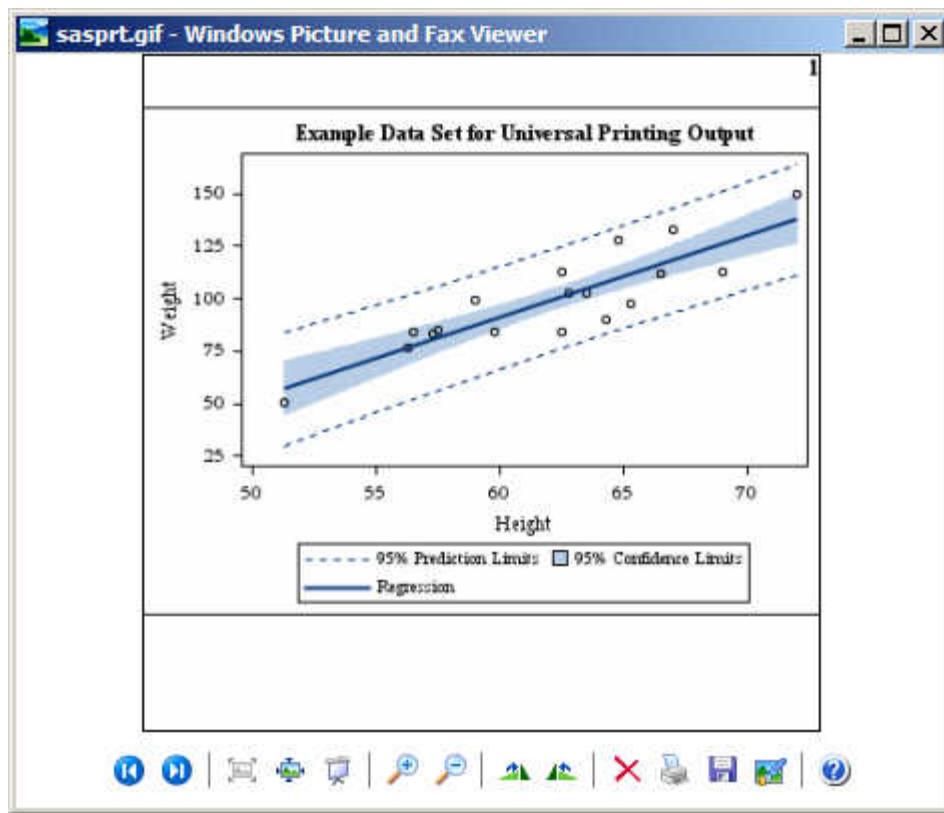
SAS creates a file sasprt.gif in the current directory
```

## Example of Creating a GIF Image Using the ODS PRINTER Statement

Using the example data set Sashelp.Class and the SGLOT procedure, the following ODS PRINTER statement prints the GIF image sasprt.gif in the current directory:

```
options printerpath=gif papersize=("4in" "4in") nodate;
ods html close;
ods printer;
proc sgplot data=sashelp.class;
    reg x=height y=weight / CLM CLI;
run;
ods printer close;
ods html;
```

Here is the GIF image in the Windows Picture and Fax Viewer:



---

## Creating PCL (Printer Command Language) Files Using Universal Printing

---

### PCL Files in SAS

PCL was developed by Hewlett-Packard (HP) as a language that applications use to control a wide range of printer features across a number of printing devices. PCL files that are created by Universal Printing can be sent to HP LaserJet printers and HP Color LaserJet printers. Universal Printing PCL printers include the PCL4, PCL5, PCL5c, and PCL5e printers:

- PCL4 produces monochrome output that is to be printed on legacy Hewlett-Packard printers that support the PCL 4 language.
- PCL5 produces monochrome output that is to be printed on Hewlett-Packard printers that support the PCL 5 language.
- PCL5c produces color output that is to be printed on Hewlett-Packard printers that support the PCL 5c language.
- PCL5e produces monochrome output at 600 dpi by default and is to be printed on Hewlett-Packard printers that support the PCL 5e language.

For a description of the PCL printers, you can either view the printers in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer pcl-printer-name;
run;
```

## See Also

[“Color Support for Universal Printers” on page 273](#)

## Creating a PCL File

You can create a PCL file using the ODS PCL or ODS PRINTER statements. ODS PCL uses the PCL5 Universal Printer by default. You can specify a different PCL printer by setting a value for PRINTER= in the ODS PCL statement. You specify the *pcl-printer* Universal Printer either as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement. If you set the PRINTERPATH=*pcl-printer* system option, you do not need to specify *pcl-printer* in the ODS PRINTER statement.

Here is some sample code to create a PCL file. The first sample specified does not specify a Universal Printer in the ODS PCL statement and SAS uses the default PCL5 printer. The second sample specified the PCL5 Universal Printer as the value of the PRINTER= option in the ODS PCL statement.

```
■ ods html close;
  ods pcl;

  ...more SAS code...

  ods pcl close;
  ods html;

■ ods html close;
  ods pcl printer=pcl5;

  ...more SAS code...

  ods pcl close;
  ods html;

:
```

Using the same sample code, you can create a PCL file by substituting ODS PCL with ODS PRINTER:

```
■ ods html close;
  ods printer printer=pcl5c;

  ...more SAS code...

  ods printer close;
  ods html;

■ options printerpath=pcl5c;
```

```
ods html close;  
ods printer;  
  
...more SAS code...  
  
ods printerl close;  
ods html;
```

SAS creates the file sasprt.pcl in the current directory. PCL files can be viewed after they are created by sending the output to a Hewlett-Packard LaserJet printer or a Hewlett-Packard Color LaserJet printer. PCL files can also be viewed on a monitor with some software applications.

---

## Creating PDF Files Using Universal Printing

---

### PDF Files in SAS

PDF files can be read by the Adobe Acrobat Reader and other applications. In SAS, you create PDF files using the Output Delivery System (ODS). ODS uses the PDF Universal Printing printer to create a PDF. ODS provides styles and templates that you can apply to a document, or you can create your own styles and templates to customize a document. For more information, see “[ODS PDF Statement](#)” in *SAS Output Delivery System: User’s Guide*.

For a description of the PDF printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;  
  printer pdf;  
run;
```

**Note:** If you have SAS/GRAF installed, your PDF output can contain links and pop-up text boxes. For more information, see “[Enhancing Web Presentations with Chart Descriptions, Data Tips, and Drill-Down Functionality](#)” in *SAS/GRAF: Reference*.

---

### Creating a PDF File

You can create a PDF file using the ODS PDF or ODS PRINTER statements. You specify the PDF Universal Printer either as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement. The ODS PDF statement creates output using the PDF Universal Printer. Therefore, you do not need to explicitly specify the PDF Universal Printer when you use the ODS PDF statement.

Here is some sample code to create a PDF file. In the first sample, the PDF Universal Printer does not need to be specified because the ODS PDF statement uses the PDF Universal Printer to create a PDF. In the second sample, the PDF

Universal Printer is specified as the value of the PRINTERPATH= system option and the ODS PRINTER statement creates the PDF:

```
■ ods html close;  
  ods pdf;  
  
  ...more SAS code...  
  
  ods pdf close;  
  ods html;  
  
■ options printerpath=pdf;  
  ods html close;  
  ods printer;  
  
  ...more SAS code...  
  
  ods printer close;  
  ods html;
```

SAS creates a file sasprt.pdf in the current directory and opens the PDF in the Results Viewer window.

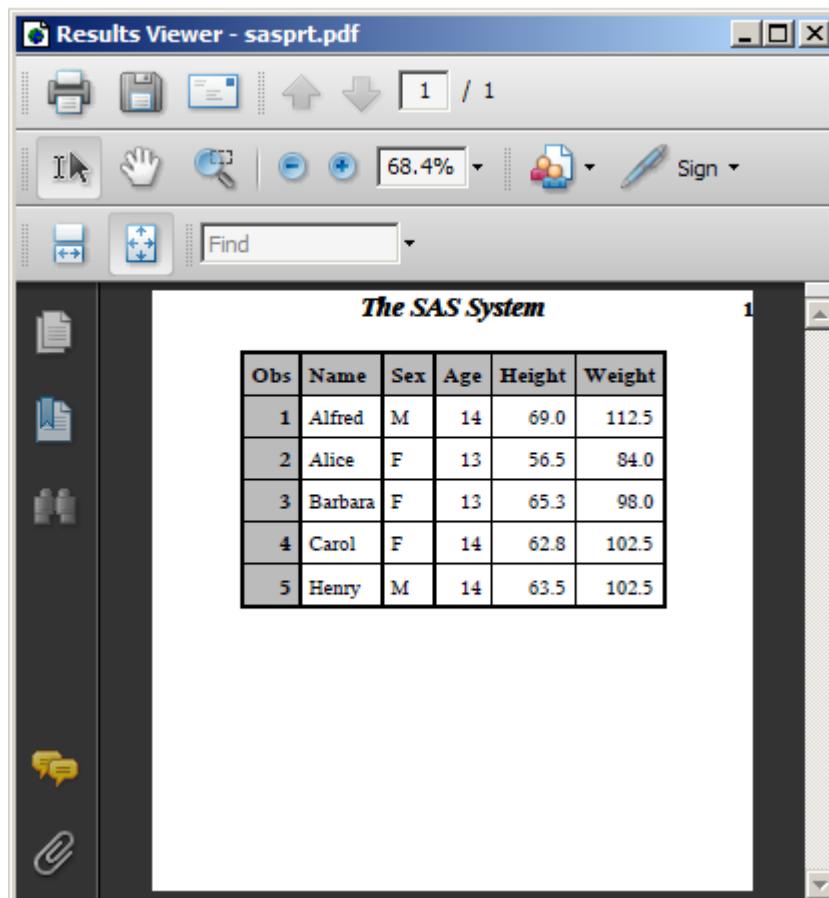
---

## Example of Creating a PDF Using the ODS PDF Statement

This example creates a PDF file that contains the first five observations of the data set Sashelp.Class:

```
options obs=5 nodate pageno=1;  
ods html close;  
ods pdf;  
  
proc print data=sashelp.class;  
run;  
  
ods pdf close;  
ods html;
```

Here is the PDF output:

**Figure 15.31** *Sashelp.Class* in a PDF File

## System Options That Affect PDF Output

Before you create PDF output, you can use SAS system options to set document security restrictions. The document security restrictions specify what can be done to the document, as well as the security method, the printing resolution, and the encryption level.

The following table lists the system options that can be used to set the PDF document security restrictions:

System Option	Description
PDFACCESS	Specifies whether the PDF document can be edited.
PDFASSEMBLY	Specifies whether PDF documents can be assembled.
PDFCOMMENT	Specifies whether PDF document comments can be modified.
PDFCONTENT	Specifies whether the contents of a PDF document can be changed.

System Option	Description
PDFCOPY	Specifies whether text and graphics from a PDF document can be copied.
PDFFILLIN	Specifies whether PDF forms can be filled in.
PDFPAGELAYOUT	Specifies the page layout for PDF documents.
PDFPAGEVIEW	Specifies the page viewing mode for PDF documents.
PDFPASSWORD	Specifies the password to use to open a PDF document and the password used by a PDF document owner.
PDFPRINT	Specifies the resolution to print PDF documents.
PDFSECURITY	Specifies the level of encryption for PDF documents.

---

## Creating PNG (Portable Network Graphics) Files Using Universal Printing

---

### Portable Network Graphics in SAS

Portable Network Graphics (PNG) is an image format that was designed to replace GIF and TIFF image formats that are viewed on the World Wide Web. PNG images that are created with the SAS Universal Printer or a SAS/GRAF device driver use the PNG Reference Library, also known as Libpng. PNG is the default format for graphics output for the ODS HTML destination and for SAS/GRAF.

For a description of the PNG printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer png;
run;
```

### See Also

[“Color Support for Universal Printers” on page 273](#)

---

### The PNG Universal Printers

SAS provides three PNG Universal Printers.

**Table 15.18** PNG Universal Printers Provided by SAS

Printer Name	Description
PNG	produces PNG images at 96 dpi
PNGt	produces PNG images at 96 dpi with a transparent background
PNG300	produces PNG images at 300 dpi

PNG printers do not support multiple page documents. If a procedure creates multiple pages or if more than one procedure is used in the code for ODS PRINTER output, only the first page is viewable.

## Creating a PNG Image

You can create a PNG image using the ODS PRINTER statements. You specify the PNG Universal Printer as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement.

Here is sample code to create a PNG image:

```
ods html close;
ods printer printer=png;

...more SAS code...

ods printer close;
ods html;
```

SAS creates the file sasprt.png in the current directory.

In SAS/GRAFPH, the PNG device is a shortcut to the PNG Universal Printer. For information about creating PNG images using SAS/GRAFPH devices, see [SAS/GRAFPH: Reference](#).

## Example of Creating a PNG File Using the ODS PRINTER Statement

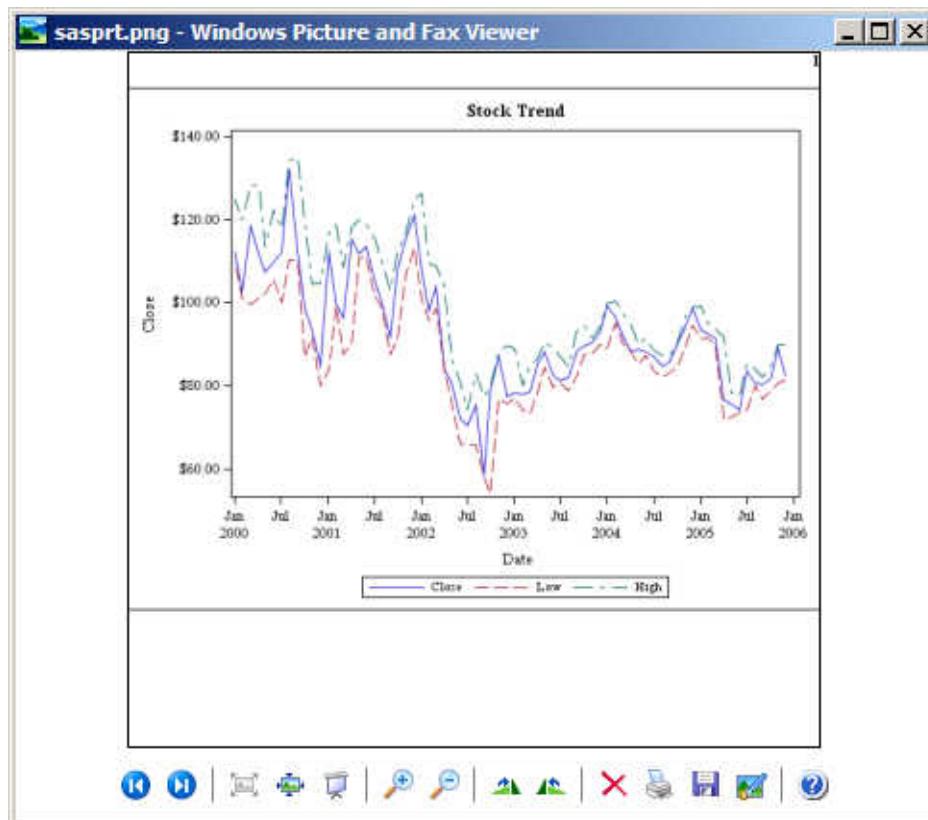
To create a PNG image in SAS using one of the PNG Universal Printers, specify the PNG printer in the PRINTERPATH= system option and the ODS PRINTER statement as shown in the following example:

```
options printerpath=png nodate;
ods html close;
ods printer;
proc sgplot data=sashelp.stocks
  (where=(date >= "01jan2000"d and stock = "IBM"));
  title "Stock Trend";
  series x=date y=close;
  series x=date y=low;
  series x=date y=high;
```

```
run;
ods printer close;
ods html;
```

The following output is the PNG graphic displayed in Windows Picture and Fax Viewer:

**Figure 15.32 A PNG Image Using ODS Printer**



## Web Browsers and Viewers That Support PNG Files

The following browsers and viewers, using the specified version or later, support most PNG image capabilities:

- Microsoft Internet Explorer 7.01b
- Mozilla Firefox 1.5.0.4
- Netscape Navigator 6
- IrfanView for Windows
- Microsoft Photo Editor
- Windows Picture and Fax Viewer

For more information about browsers and viewers that support PNG images, see the PNG web pages at [www.libpng.org](http://www.libpng.org).

---

# Creating PostScript Files Using Universal Printing

---

## PostScript Files in SAS

Universal Printing supports several levels of the PostScript printer. The default PostScript printer and the default Universal Printer is the PostScript Level 1 color printer. You create PostScript files using the Output Delivery System (ODS).

For a description of the PostScript Universal Printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer postscript;
run;
```

PostScript output supports transparent GIF files. You can use Ghostview to view PostScript files. If you have Acrobat Distiller installed, you can distill the PostScript file to create a PDF file that you can view in Adobe Reader.

### See Also

[“Color Support for Universal Printers” on page 273](#)

---

## Creating a PostScript File

You can create a PostScript file using the ODS PS or ODS PRINTER statements. You specify the PS Universal Printer either as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement. The ODS PS statement creates output using the PS Universal Printer. Therefore, you do not need to explicitly specify the PS Universal Printer when you use the ODS PS statement.

Here is some sample code to create a PS file. In the first sample, the PS Universal Printer does not need to be specified because the ODS PS statement uses the PS Universal Printer to create a PS file. In the second sample, the PS Universal Printer is specified as the value of the PRINTERPATH= system option and the ODS PRINTER statement creates the PS file:

```
■ ods html close;
  ods ps;

  ...more SAS code...

  ods ps close;
  ods html;
```

```
■ options printerpath=ps;
  ods html close;
  ods printer;

  ...more SAS code...

  ods printer close;
  ods html;
```

SAS creates a file sasprt.ps in the current directory.

---

## Example of Creating a PostScript File Using the ODS PS Statement

This example creates a PS file that contains the first five observations of the data set Sashelp.Class:

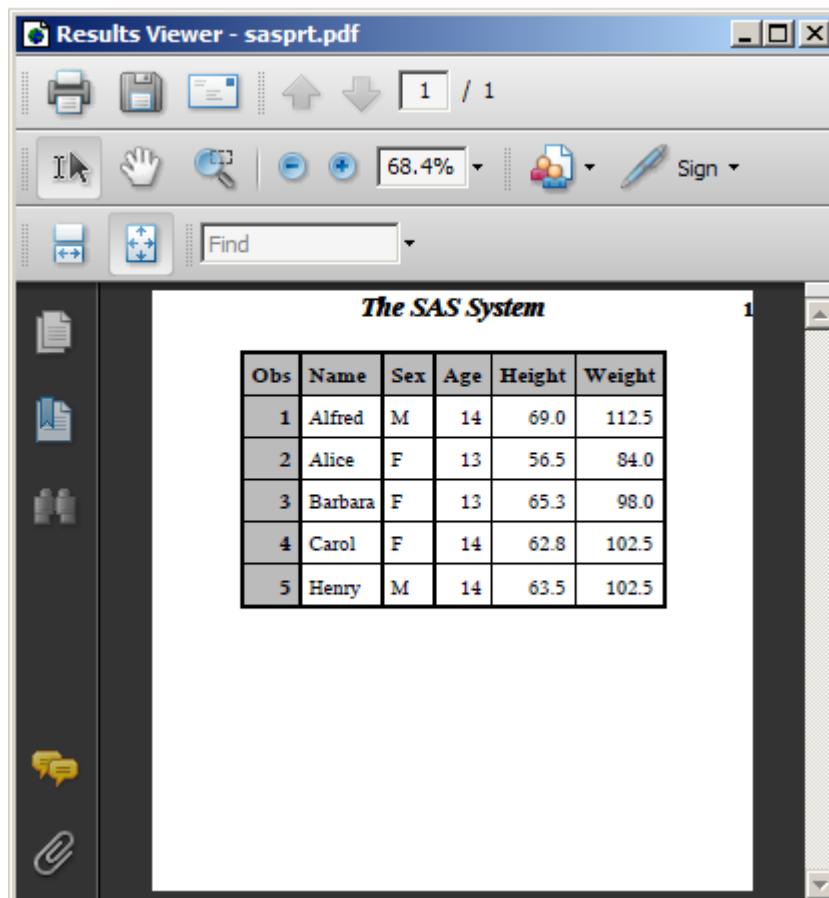
```
options obs=5 nodate pageno=1;
ods html close;
ods ps;

proc print data=sashelp.class;
run;

ods ps close;
ods html;
```

Here is the distilled PostScript file in PDF output:

Figure 15.33 Sashelp.Class in a PDF File



Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5

## Creating SVG (Scalable Vector Graphics) Files Using Universal Printing

### Overview of Scalable Vector Graphics in SAS

#### Scalable Vector Graphics in SAS

Scalable Vector Graphics (SVG) is an XML language for describing two-dimensional vector graphics. SAS creates SVG documents based on the W3C recommendation for SVG documents. SAS SVG documents are created using the UNICODE standard encoding UTF-8.

SAS can create SVG documents by using Universal Printers and SAS/GRAF device drivers. Because SAS/GRAF SVG device drivers use the SVG Universal Printers, this section contains some information about creating SVG documents using SAS/GRAF.

Most often in SAS, the SVG Universal Printers and device drivers are used to create graphs. Graphs can be created by using ODS Graphics or SAS/GRAFPH. You can also use the SVG Universal Printers to show tables or reports that you create as SVG documents.

Several ODS destinations (EPUB, HTML, HTML5, LISTING, and PRINTER destinations) can be used to create SVG documents. SVG is the default Universal Printer and device driver for the ODS HTML5 destination.

SVG documents can be stand-alone files or integrated within an HTML5 or EPUB file. A stand-alone SVG document can be referenced as a link target, referenced as an embedded file in an HTML document, or referenced as a CSS2 or XSL property. For information about embedding SVG documents in web pages, see the topic on using SVG documents in web pages in the SVG 1.1 specification on the W3 SVG website <http://www.w3.org/TR/SVG>.

Multi-page SVG documents can be animated in Base SAS and SAS/GRAFPH. When you create animated SVG documents in Base SAS using Universal Printing without specifying any ODS Graphics procedures, the animated SVG documents appear as a slide show or an animated PowerPoint presentation. For more information, see “[Creating Animated GIF Images and SVG Documents](#)” on page 375.

If you have SAS/GRAFPH installed, your SVG documents can contain links and pop-up text boxes.

The information provided here is limited to creating SVG documents using Universal Printers in Base SAS and ODS Graphics. For more information about creating SVG files in SAS/GRAFPH, see “[Enhancing Web Presentations with Chart Descriptions, Data Tips, and Drill-Down Functionality](#)” in *SAS/GRAFPH: Reference*.

For detailed information about the SVG standard, see the W3 documentation at <http://www.w3.org/TR/SVG>.

## SVG Terminology

### SVG canvas

the space upon which the SVG document is rendered.

### viewBox

specifies the coordinate system and the area of the SVG document that is visible in the viewport.

### viewport

a finite rectangular space within the SVG canvas where an SVG document is rendered. In SAS, the viewport is determined by the value of the PAPERSIZE= system option for a scalable viewport and by the SVGWIDTH= and SVGHEIGHT= system options for a static viewport.

### viewport coordinate system or viewport space

the starting X and Y coordinates and the width and height values of the viewport.

### user coordinate system or user space

the starting X and Y coordinates and the width and height values of the area of the document to display in the viewport.

### user units

is equal to one unit of measurement that is defined in your environment's coordinate system. In many cases, the coordinate system uses pixels. Check with your system administrator to determine the unit of measure that is used in your environment.

## Why Create SVG Documents?

SVG documents are displayed clearly at any size in any viewer or browser that supports SVG. SVG documents are ideal for producing documents to display on a computer monitor, PDA, or cell phone; or documents to be printed. Because it is a vector graphic, a single SVG document can be transformed to any screen resolution without compromising the clarity of the document. In comparison, a multiple raster graphic image might require using different screen resolutions in order to display the image at various screen resolutions and sizes.

An SVG document might also be smaller in file size than the same image created by a raster graphic Universal Printer, such as GIF or PNG.

## Web Server Content Type for SVG Documents

If the mime content type setting for your web server does not have the correct setting for SVG documents, your web browser might render SVG documents as text files, or SVG documents might be unreadable.

To ensure that SVG documents are rendered correctly, configure your web server to use this mime content type:

`image/svg+xml`

## The SVG Universal Printers and the Output That They Create

The following table lists the SAS SVG Universal Printers.

**Table 15.19** *SVG Universal Printers*

Printer Name	Description
SVG *	produces SVG 1.1 documents.
SVGt *	produces SVG 1.1 documents that are transparent (no background).
SVGnotip	produces SVG 1.1 documents without tooltips.
SVGZ *	produces compressed SVG 1.1 documents.
SVGView *	produces SVG1.1 documents with navigational controls when the SVG file contains multiple pages.

\* When you use this printer in SAS/GRAPH, you can create pop-up data tips. For more information, see “[Data Tips for Web Presentations](#)” in *SAS/GRAPH: Reference*.

SVG prototypes for creating printers are available in the SAS Registry under CORE \PRINTING\PROTOTYPES. You can define your own SVG printer using the PRTDEF procedure. For more information, see “[PRTDEF Procedure](#)” in *Base SAS Procedures Guide* and “[Managing Universal Printers Using the PRTDEF Procedure](#)” on page 302.

For a description of an SVG printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer svg-printer-name;
run;
```

## See Also

- [“Color Support for Universal Printers” on page 273](#)
- [“Creating Animated GIF Images and SVG Documents” on page 375](#)

---

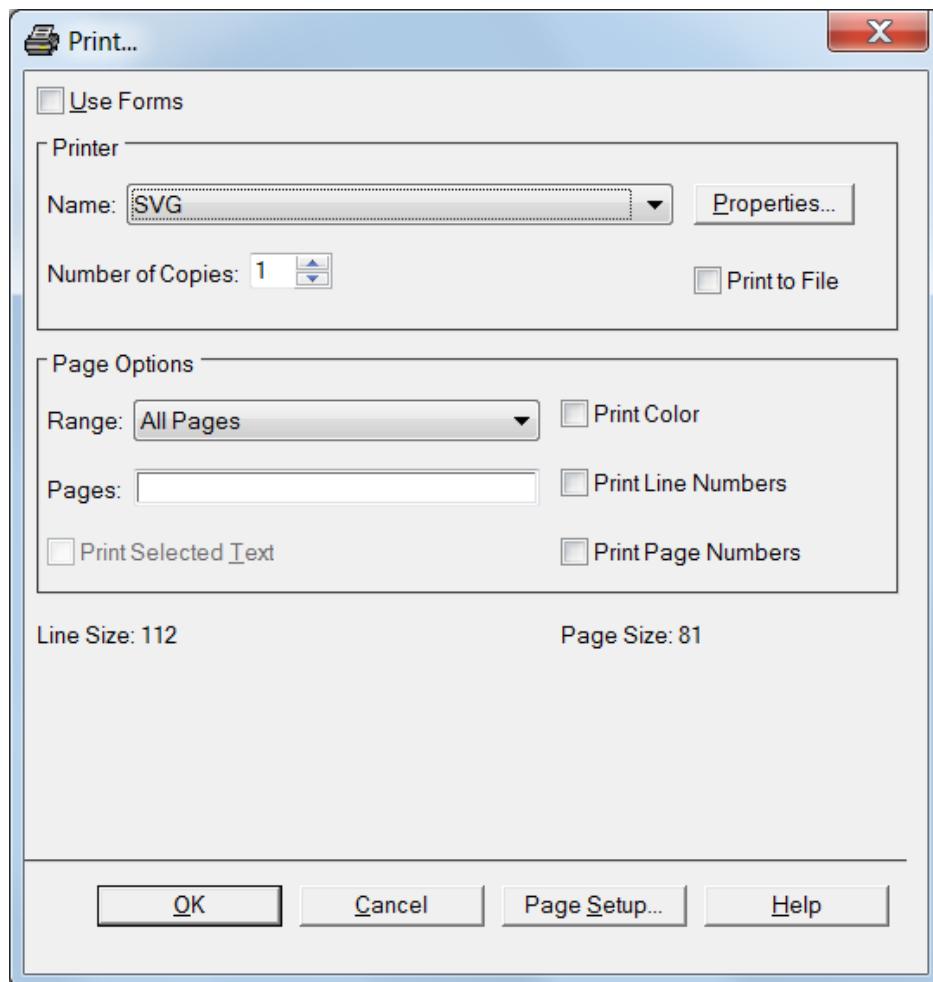
## How to Create SVG Documents

### Fundamentals of Creating SVG Documents Using Universal Printers

You can create SVG documents by using the Print dialog box or by using programming statements.

To create SVG documents by using the Print dialog box, select an SVG printer from the Name drop-down list box.

Figure 15.34 Print Dialog Box for Printing an SVG Document



To create SVG documents programmatically, specify an SVG Universal Printer as the value of the PRINTERPATH= system option. Also, specify an ODS destination, such as the ODS PRINTER statement, as shown below.

```
options printerpath(svg);
ods html close;
ods printer;

...more SAS code...
```

```
ods printer close;
ods html;
```

Alternatively, you can specify the SVG printer in the ODS PRINTER statement and eliminate the OPTIONS statement, as shown below.

```
ods html close;
ods printer printer(svg);

...more SAS code...
```

```
ods printer close;
ods html;
```

To create SVG graphs using SAS/GRAFH, you can use the ODS LISTING statement:

You can create SVG graphs for ODS Graphics using these statements:

- ods html5 options (svg\_mode="inline");  
ods graphics /imagefmt=svg;  
  
...more SAS code...
- ods html5 close;
- options printerpath=svg;  
ods html;  
  
...more SAS code...
- ods html close;
- ods listing;  
ods graphics /imagefmt=svg;  
  
...more SAS code...
- ods listing close;
- Using SAS/GRAFH:  
  
ods listing;  
goptions dev=SVG;  
  
...more SAS code...
- ods listing close;

SAS has several system options that enable you to modify various aspects of your SVG document. Here are some SVG document traits:

- a specific SVG Universal Printer
- the height and width of the SVG document
- the size of the viewBox
- whether a multi-page SVG document contains navigational controls

By using the NEWFILE option in the ODS PRINTER statement, you can create an SVG document for the output from each procedure or DATA step.

For more information, see the following language elements:

- “[PRINTERPATH= System Option](#)” in *SAS System Options: Reference*
- “[ODS PRINTER Statement](#)” in *SAS Output Delivery System: User’s Guide*
- [System Options for SVG Documents on page 353](#)

## A Summary of ODS Destinations to Create SVG Documents

The following table shows the ODS destinations that you can use to create SVG documents and the output that is created:

<b>Destination</b>	<b>Type of SVG Document</b>	<b>Output Created</b>
ODS EPUB *	graphs created by ODS Graphics and SAS/GRAFH	an EPUB file with integrated SVG documents
ODS HTML	graphs created by ODS Graphics and SAS/GRAFH	an SVG file for each graph and an HTML file
ODS HTML5 svg_mode='inline' *	graphs created by ODS Graphics and SAS/GRAFH	an HTML file with integrated SVG documents
ODS HTML5 svg_mode='embed' *	graphs created by ODS Graphics and SAS/GRAFH	an SVG file for each graph and an HTML file
ODS LISTING	graphs created by ODS Graphics and SAS/GRAFH	an SVG file for each graph
ODS PRINTER	all output created by the DATA step and SAS procedures	one SVG file for all output created between ODS PRINTER and ODS PRINTER CLOSE

\* SVG is the default printer.

**Note:** Graphs that are created by ODS Graphics do not use options that are specified by the GOPTIONS SAS/GRAFH statement. The GOPTIONS statement is valid only for SAS/GRAFH.

The default filename for an SVG file that was created with an ODS Graphics procedure is prefixed with the procedure name. For example, the default filename for PROC SGLOT output could be sgplot01.svg. The default filename for an SVG file that was created using ODS PRINTER is sasprt.svg.

## Browser Support for Viewing SVG Documents

### Browsers That Support SVG Documents

In order to view SVG documents, you need a viewer or browser that supports Scalable Vector Graphics. Some browsers, such as Mozilla Firefox, have built-in support for SVG documents.

The following table lists some browsers and viewers that support SVG documents.

*Table 15.20* SVG Browser Support

<b>Browser or Viewer</b>	<b>Company</b>
Batik SVG Toolkit	Apache Software Foundation
eSVG Viewer and IDE	eSVG Viewer for PC, PDA, Mobile
Google Chrome *	Google Inc.
GPAC Project	GPAC

Browser or Viewer	Company
Internet Explorer 9 or later*	Microsoft
Mozilla Firefox *	Mozilla Foundation
Opera	Opera Software ASA
Safari, including iPad*	Apple, Inc.
TinyLine	TinyLine

\* This browser is supported by SAS.

## Notes on Using Mozilla Firefox

- Compressed SVG documents using the SVGZ Universal Printer are not supported.
- Zooming and panning features are not currently implemented.
- If you select **View**  $\Rightarrow$  **Page Style**  $\Rightarrow$  **No Style**, all graphs appear as a black rectangle.
- Firefox does not support font embedding. To avoid font mapping problems in your SVG document, you can set the NOFONTEMBEDDING system option. If the FONTEMBEDDING option is set when an SVG document is created and the SVG document is subsequently viewed in Firefox, Firefox uses the default font setting that is defined on the **Contents** tab in the Firefox Options dialog box.

## Notes on SVG Documents in HTML5 Output

In HTML5 output, SVG document height and width attributes are set to the size of the SVG because many browsers do not scale an SVG file to 100% of the container. The Google Chrome and Safari browsers do scale the SVG file to 100% of the container.

To enable scaling of SVG files that can be scaled to the size of the container, you set the SVGHEIGHT= and SVGWIDTH= system options to 100%:

```
options svgheight="100%" svgwidth="100%";
```

SVGZ documents that you create for ODS HTML5 output can be viewed only with the Google Chrome or Opera web browsers.

---

## Images in SVG Documents

When your SAS program creates an SVG document that contains images, SAS does the following for each image in the document:

- converts the specified image to a PNG format
- encodes the PNG image using base64 encoding
- embeds the base64 encoded PNG image into the SVG document

In the SVG document, the `<image>` element has an `xlink` attribute that begins as follows:

```
xlink:href="data:image/png;base64,"
```

The base64 encoded image follows after `base64`.

By default, the SVG Universal printer encodes PNG files and embeds them in the SVG document. In the following example, you see the `<image>` element at the bottom. The encoded image begins after the `/png;base64,` element attribute. The first characters of the image are `iVBORw0KGgo`. The encoded characters extend to the right on the same line and cannot be shown in this document.

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" xml:space="pre">
<desc></desc>
<svg id="SVGMain_SVG0orig1" viewBox="-1 -1 801 621">
</svg>
<svg id="SVGMain_SVG1" viewBox="-1 -1 801 621">
<defs>
<clipPath id="SVGMain_clipPage1">
<rect x="-1" y="-21" width="801" height="621"></rect>
</clipPath>
</defs>
<g id="SVGMain_Page1" transform="translate(0,20)" clip-path="url(#SVGMain_clipPage1)">
<rect x="0" y="0" width="800" height="600" style="fill: #FFFFFF; stroke: #FFFFFF; stroke-width: 1px;">
</defs>
<image id="SVGMain_Image1" width="154" height="36" xlink:href="data:image/png;base64,iVBORw0KGgo...">
</defs>

```

The image is the SAS logo in this SVG document:

*Output 15.4 SVG document viewed in a web browser*

## Five Observations of sashelp.class



Obs	Name	Sex	Age	Height	Weight
1	Alfred	M	14	69.0	112.5
2	Alice	F	13	56.5	84.0
3	Barbara	F	13	65.3	98.0
4	Carol	F	14	62.8	102.5
5	Henry	M	14	63.5	102.5

The following program created the embedded image:

```
proc template;
  define style logo;
    parent = Styles.barrettsblue;
    style Body from Body /
      backgroundimage = 'c:\saslogo.png';
  end;
run;

ods html close;
options nodate nonumber orientation=landscape obs=5;

ods printer style=logo printer=svgview file='logo.svg';

proc print data=sashelp.class;
  title color="black" 'Five Observations of sashelp.class';
```

```
run;
ods printer close;
ods html;
```

An alternative to embedding encoded PNG files in an SVG document, the SVG Universal Printer can create separate PNG files and link to them from within the SVG document. Here is an example of an <image> element in an SVG document:

```
<image id="Image3" width="200" height="150" xlink:href="I3svgimg.png">
</image>
```

The SVG Universal Printer creates separate PNG files when the SVG printer that you are using has the **Images Embedded** registry setting set to 0.

To set this registry setting, do the following:

- 1 To open the Registry editor, select **Solutions**  $\Rightarrow$  **Accessories**  $\Rightarrow$  **Registry Editor**.
- 2 In the Registry Editor, expand **CORE**  $\Rightarrow$  **PRINTING**  $\Rightarrow$  **PRINTERS**  $\Rightarrow$  **svg-printer**  $\Rightarrow$  **ADVANCED**.
- 3 Right-click **Images Embedded**, select **Modify**, and change **Value Data** to 0.
- 4 Click **OK**.

The PNG filename has the form *counterPrinterDestinationFilename.png*. *counter* is an integer that is incremented each time a new image is created.

*PrinterDestinationFilename* is the output destination filename for the printer. For example, using the default printer destination filename, sasprt, the first three images would be named I1sasprt.png, I2sasprt.png, and I3sasprt.png.

SAS writes a note in the SAS log with the path to the images.

## Setting the Environment to Create Stand-alone SVG Documents

### Overview of Setting the Environment to Create Stand-alone SVG Documents

As shown in “[Fundamentals of Creating SVG Documents Using Universal Printers](#)” on page 346, an SVG Universal Printer must be specified either as the printer value using the PRINTERPATH= system option or the ODS Printer statement. You can set any of the SVG system options when SAS is invoked in a SAS program by using the OPTIONS statement, or by using the SAS System Options window.

SAS SVG documents can be created easily by using default values for SVG system options (except for the PRINTERPATH= system option) that establish the SVG environment. This section explains the SVG system options and how they effect stand-alone SVG documents.

## SAS System Options That Affect Stand-alone SVG Documents

You can use the following system options to set the environment for creating SVG documents:

**Table 15.21 SAS System Options That Affect SVG Documents**

Task	System Option
Specify the name of an SVG printer to create a stand-alone SVG document.	<a href="#">PRINTERPATH=</a>
Embed a comment in the SVG document	<a href="#">COLOPHON</a>
<b>Options to set the SVG document size</b>	
Set the paper size to use for Universal Printing.	<a href="#">PAPERSIZE=</a>
Set the height of the SVG document. If the SVG document has embedded SVG documents, the height value affects only the outermost SVG document.	<a href="#">SVGHEIGHT=</a>
Set the width of the SVG document. If the SVG document has embedded SVG documents, the width value affects only the outermost SVG document.	<a href="#">SVGWIDTH=</a>
Set the x-axis coordinate for the lower left corner of an embedded SVG document.	<a href="#">SVGX=</a>
Set the y-axis coordinate for lower left corner of an embedded SVG document.	<a href="#">SVGY=</a>
Specify the X and Y coordinates, and the width and height that are used to set the viewBox for the outermost SVG document; specify the coordinates of the area of the document that is displayed in the viewport.	<a href="#">SVGVIEWBOX=</a>
Specify whether to force uniform scaling of an SVG document.	<a href="#">SVGPRESERVEASPECTRATIO=</a>
<b>Options that modify the SVG document appearance</b>	
Set the title that appears in the title bar of the SVG document.	<a href="#">SVGTITLE=</a>
Specify whether to display navigational controls in a multi-page SVG document.	<a href="#">SVGCONTROLBUTTONS</a>
Specify whether to display the magnify tool in SVG documents.	<a href="#">SVGMAGNIFYBUTTON</a>
<b>Options for animating SVG documents</b>	

Task	System Option
Start or stop creating an animation file.	<a href="#">ANIMATION</a>
Set the amount of time that each frame of an animated document is held in view.	<a href="#">ANIMDURATION</a>
Specifies whether to loop through the animation continuously or to play it one time.	<a href="#">ANIMLOOP</a>
Specifies whether to overlay frames in the animation or to play them sequentially.	<a href="#">ANIMOVERLAY</a>
Specifies whether to immediately start an animation when an SVG document appears in the web browser.	<a href="#">SVGAUTOPLAY</a>
Sets the number of seconds for a frame to fade into view.	<a href="#">SVGFADEIN</a>
Specifies whether a frame in the animation overlaps the previous frame or if each frame is played sequentially when a frame is specified to fade in and out.	<a href="#">SVGFADEMODE</a>
Sets the number of seconds for a frame to fade out of view.	<a href="#">SVGFADEOUT</a>

For more information, see [SAS System Options: Reference](#).

## Setting the SVG Universal Printer

You set the SVG Universal Printer by setting the PRINTERPATH= system option to one of the SVG Universal Printers. You can set the PRINTERPATH= system option at any time. The following OPTIONS statement sets the Universal Printer to create compressed SVG documents:

```
options printerpath=svgz;
```

For more information, see the following topics:

- “[The SVG Universal Printers and the Output That They Create](#)” on page 345
- “[PRINTERPATH= System Option](#)” in [SAS System Options: Reference](#)
- “[Fundamentals of Creating SVG Documents Using Universal Printers](#)” on page 346

## Scaling an SVG Document to the Viewport

To scale an SVG document to the viewport, you can use the default value of null for the SVGHEIGHT= and SVGWIDTH= system options. A null value equates to the value of 100%, which scales the SVG document to the size of the viewport. In addition, the value of the SVGVIEWBOX= system option must be null.

For more information, see the following system options in [SAS System Options: Reference](#):

- [SVGHEIGHT= System Option](#)

- [SVGWIDTH= System Option](#)
- [SVGVIEWBOX= System Option](#)

## Setting the ViewBox

The viewBox attribute on the `<svg>` element is a set of four numbers: the starting X coordinate, the starting Y coordinate, the height of the SVG document, and the width of the SVG document. SAS sets the viewBox attribute value from the value of the `SVGVIEWBOX=` system option. If that option has no value, SAS uses the value of the `PAPERSIZE=` system option to set the height and the width arguments of the `viewBox` attribute. The starting coordinate values are set to 0.

When the `SVGVIEWBOX=`, `SVGHEIGHT=`, and `SVGWIDTH=` system options have a null value (the default value for each of these system options), the SVG document scales to the size of the viewport. If you specify a value for the `SVGVIEWBOX=` system option, the SVG document is scaled to the dimensions specified in the `SVGVIEWBOX=` option.

If you specify the `SVGHEIGHT=` option and the `SVGWIDTH=` option using percentage units, the SVG document scales to the size of the browser window whenever the browser window changes size. If these options are specified using units other than percentage, such as in, cm, or px, the SVG document is a static size and does not scale to the browser window when the window changes size.

For more information, see the following topics:

- [“ODS PRINTER Statement” in \*SAS Output Delivery System: User’s Guide\*](#)
- [“`SVGVIEWBOX=` System Option” in \*SAS System Options: Reference\*](#)
- [“`PAPERSIZE=` System Option” in \*SAS System Options: Reference\*](#)
- [“Creating a Static viewBox” on page 356](#)

## Interaction between SAS SVG System Options and the SVG Element Attributes

SAS uses the values of the `SVGHEIGHT=`, `SVGWIDTH=`, `SVGVIEWBOX=`, `SVGPRESERVEASPECTRATIO=`, `SVGX=`, and `SVGY=` system options as values for their respective attributes on the outermost `<svg>` element: height, width, viewBox, and preserveAspectRatio. For example, if you specify `SVGWIDTH=“400”` and `SVGHEIGHT=“300”`, SAS creates the `<svg>` element with the attributes `width=“400”` and `height=“300”`. The values of the `SVGX=` and `SVGY=` system options are used only on embedded `<svg>` elements for the x and y attributes.

All of these system options have a null default value. When the `SVGVIEWBOX=` system option is null, SAS determines the viewBox size based on the value of the `PAPERSIZE=` system option. Therefore, if you do not specify a value for any of these system options, the only `<svg>` attribute that SAS sets is the `viewBox` attribute using the SAS SVG system options. Other `<svg>` attributes, such as `version` and `xmlNs` are set by SAS and not by using system options.

SAS creates the following `<svg>` element when all of the SAS SVG system options are set to their default values:

```
<svg xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink"
      xml:space="preserve" baseProfile="full" version="1.1"
      id="SVGMain" onload='SVGMain_Init("SVGMain")'
```

```
viewBox="-1 -1 801 601">
```

The **SVGPRESERVEASPECTRATIO=** system option is used to set the **preserveAspectRatio** attribute in the **<svg>** element and has an effect only when the **viewBox** attribute has also been specified in an SVG document.

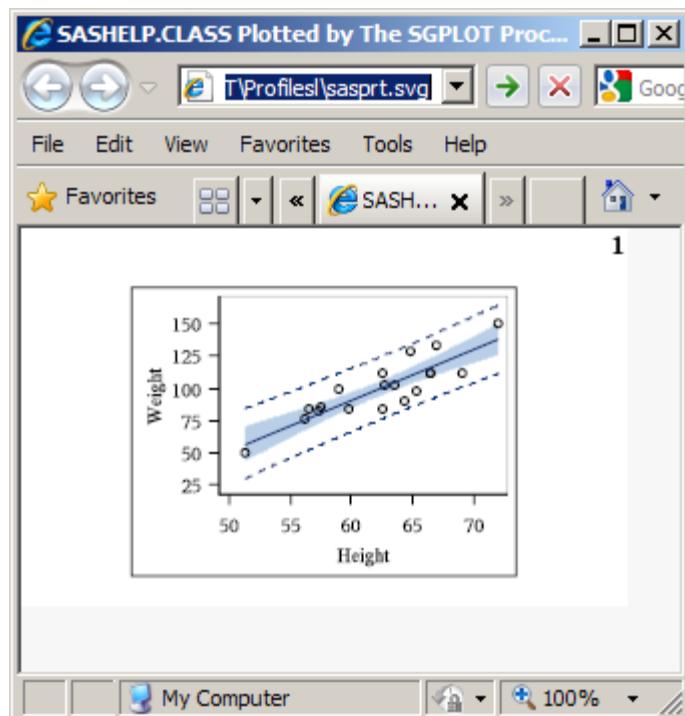
Negative values can be specified for the values of SVG options. However, if a negative value is specified for the **SVGHEIGHT=** option or the **SVGWIDTH=** option, or the height or width arguments in the **SVGVIEWBOX=** option, the SVG document is not rendered by the browser. It can be useful to specify negative values for the **x** and **y** arguments of the **SVGVIEWBOX=** option to place the origin of the SVG document. A negative argument in the **SVGVIEWBOX=** option shifts the output to the right. A negative value of the **SVGVIEWBOX=** option shifts the placement of the document downward.

## Creating a Static viewBox

A static **viewBox** is a **viewBox** that cannot be changed. When the viewport changes, such as when you resize your browser window, the **viewBox** remains the same size. To create a static **viewBox**, you specify the same width and height values for the **PAPERSIZE=**, **SVGWIDTH=**, and **SVGHEIGHT=** system options. The **PAPERSIZE=** system option sets the **viewBox**. The **SVGWIDTH=** and **SVGHEIGHT=** system options set the size of the SVG document. If the **SVGHEIGHT=** and **SVGWIDTH=** options are specified using percentage unit, the SVG document scales to the size of the browser window when the browser window changes size. [Figure 15.35 on page 356](#) shows a static **viewBox** created by using the following system options:

```
options nodate printerpath=svg papersize=("8cm" "5cm") svgwidth="8cm" svgheight="5cm"
      svgttitle="Sashelp.Class Plotted by The SGPlot Procedure";
```

**Figure 15.35** A Static Viewbox



To reset the `SVGWIDTH=`, `SVGHEIGHT=`, and `SVGPRESERVEASPECTRATIO=` system options to null, specify two single quotation marks or two double quotation marks with no space between them:

```
options printerpath=svg svgwidth="" svgheight="" svgpreserveaspectratio="";
```

## Preserving the Aspect Ratio

When you change the size of the `viewBox`, you can use the `SVGPRESERVEASPECTRATIO=` system option to specify whether you want to preserve the aspect ratio of the SVG document and how to place the SVG document in the viewport. Set this option by using one of the following assignments:

`SVGPRESERVEASPECTRATIO=align | meetOrSlice | NONE | “”`

`SVGPRESERVEASPECTRATIO=“align meetOrSlice”`

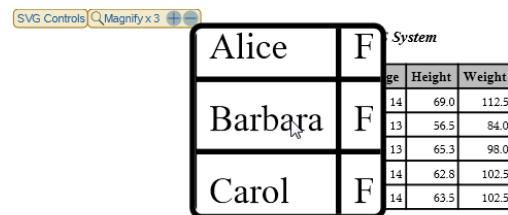
The first argument, *align*, specifies whether to force uniform scaling by specifying the alignment method to use. For example, you can use the `xMidYMid` value to align the midpoint X value of the `viewBox` to the midpoint X value of the viewport, which centers the document horizontally.

The second argument, *meetOrSlice*, specifies how to scale the SVG document to the `viewBox`. The value for this argument can be either `meet` or `slice`. If you specify `meet`, the SVG document is scaled up as much as possible while meeting other criteria. The viewport displays some unused space. If you specify `slice`, the SVG document is scaled down as much as possible while meeting other criteria. In the latter case, some of the SVG document appears to be cut off. The SVG document is still complete, but you cannot see all of it in the viewport. You can use your browser controls to move the SVG document around in the viewport.

For more information, see “[SVGPRESERVEASPECTRATIO= System Option](#)” in [SAS System Options: Reference](#).

## Including the Magnify Tool in SVG Documents

You can include a magnify tool in SVG documents by setting the `SVGMAGNIFYBUTTON` system option. When the tool is enabled, a magnifying glass is available to enlarge a portion of an SVG document. The size of the magnification area cannot be changed.



		System		
		Age	Height	Weight
Alice	F	14	69.0	112.5
Barbara	F	13	56.5	84.0
Carol	F	13	65.3	98.0
		14	62.8	102.5
		14	63.5	102.5

By default, the magnify tool is not included in the SVG document. You must explicitly set the `SVGMAGNIFYBUTTON` system option. You can use this `OPTIONS` statement:

```
options svgmagnifybutton;
```

To disable the magnify tool, use the `NOSVGMAGNIFYBUTTON` system option.

When the magnify tool is enabled, the **Magnify** button, , appears at the top of the SVG document. To make the magnifying glass appear, click **Magnify**. Using the mouse, move the magnifying glass over the SVG document to enlarge the area under the glass. To turn the magnifying glass off, click **Magnify** again. By default, the magnifying glass enlarges the area using a magnification level of three. Click on + to increase the magnification level and – to decrease the magnification level. When the SVG document is viewed on an iPad, the first tap of the **Magnify** button displays the tooltip. The second and subsequent taps change the magnification and closes the magnifying glass.

There are some restrictions for using the magnify tool:

- The magnify tool is not supported for the SVGT printer and animated SVG documents.
- When you use the SVGnotip printer, no tooltip is displayed to tell you to enable or disable the magnifying glass.
- When the magnify tool is enabled, the magnify tool is turned off on the Index page of a multi-page document.

The magnify tool is more useful when the SVG document is viewed in browsers that expect an SVG document to control the zoom level.

For more information, see “[SVGMAGNIFYBUTTON System Option](#)” in *SAS System Options: Reference*.

## Adding a Title to an SVG Document

You use the **SVGTITLE=** system option to add a title to the title bar of a window when the browser displays only the SVG document. If the SVG document is embedded in an HTML page, the **svgttitle** attribute on the **<svg>** tag has no effect. The static viewport example in the previous topic shows a title in the browser title bar.

For more information, see “[SVGTITLE= System Option](#)” in *SAS System Options: Reference*.

## Creating Stand-alone SVG Documents Using the ODS PRINTER Destination

### Creating an SVG Document

To create an SVG document, you need to at least set the **PRINTERPATH=** system option to an SVG Universal printer and specify the ODS PRINTER statement in your SAS program. Or, specify the **PRINTER=** option in the ODS PRINTER statement:

```
options printerpath=svg;
ods html close;
ods printer;
proc sgplot data=sashelp.class;
  reg x=height y=weight / CLM CLI;
run;
ods printer close;
ods html;
```

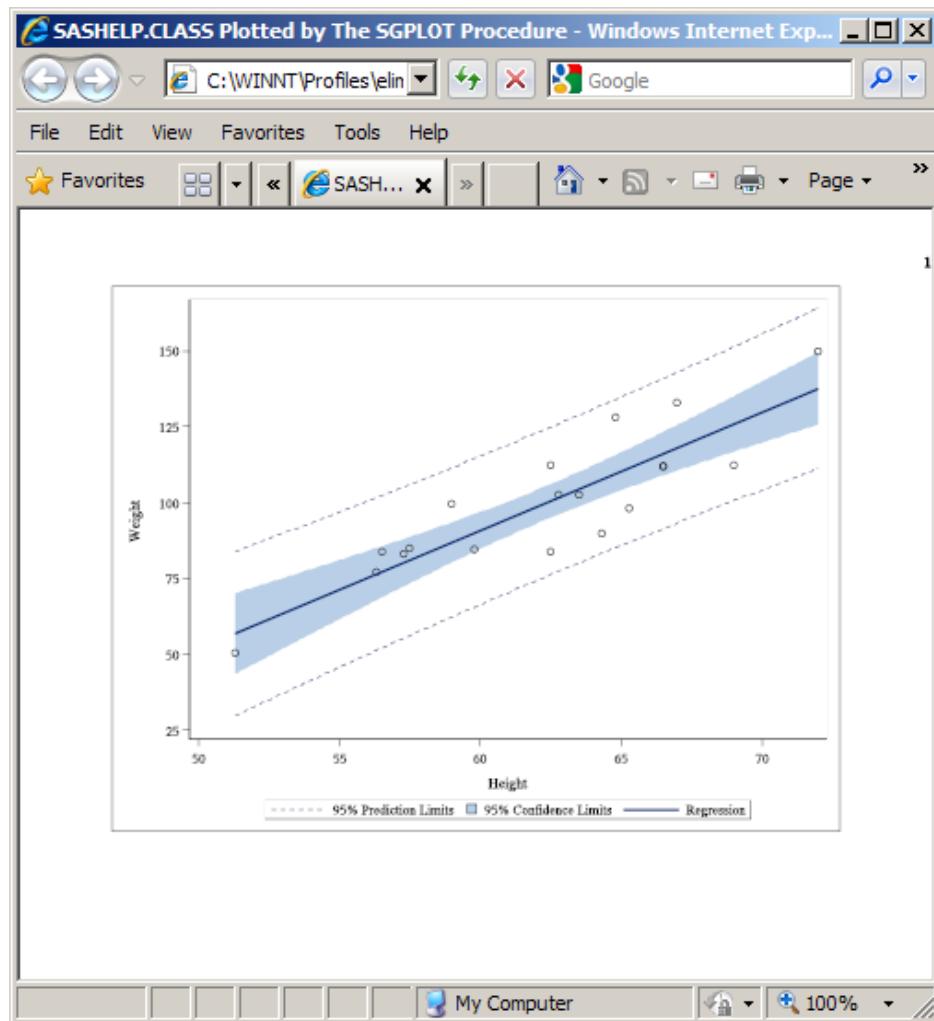
In this example, no specific SVG system option values were set to size the SVG document. Therefore, the viewBox is the default size specified by the PAPERSIZE= system option. The SVG document scales to the viewport because no value was specified for the SVGWIDTH= and SVGHEIGHT= system options. The following is the <svg> element that SAS creates:

```
<svg xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xml:space="preserve" baseProfile="full" version="1.1"
  id="SVGMain" onload='SVGMain_Init("SVGMain")'
  viewBox="-1 -1 801 601">
```

SAS creates a single SVG document named sasprt.svg and stores it in a specific location, depending on your operating environment. Under Windows, the file is stored in the current directory. Under UNIX, the file is stored in your home directory. Under z/OS, the file is stored as a z/OS UNIX System Services Hierarchical File System (HFS) file, or as a z/OS data set. If the SVG file is written to a z/OS data set, it is written to PDSE library *userid.SASPRT.SVG*. You can use the FILE= option in the ODS PRINTER statement to specify a different filename.

The following figure is an SVG file that uses the Adobe Acrobat SVG plug-in for Microsoft Internet Explorer. This file was created by using the SGPLOT procedure to plot the Sashelp.Class data set.

**Figure 15.36** Sashelp.Class Plotted by the SGPlot Procedure as an SVG File



When you use the SVG, SVGnotip, SVGt, SVGView, and SVGZ Universal Printers, SAS creates a single SVG document. Depending on the size of the SVG document, the browser might display the complete SVG document. Check the documentation for your browser to determine whether your browser has controls for viewing SVG documents. In the Adobe SVG Viewer plug-in for Internet Explorer, you can press the Alt key and the left mouse button to pan and move to different pages in a continuous, multi-page SVG document.

## Multi-Page SVG Documents in a Single File

When a DATA step or procedure creates a new page in the output, a new SVG page is created in an SVG document. SAS creates either one file with multiple pages or multiple SVG files with one file for each SVG document page. The **SVGCONTROLBUTTONS** system option and the **NEWFILE=** option in the ODS PRINTER statement control whether a multi-page SVG document is one continuous file (with controls to navigate the pages in the file) or multiple SVG files.

SAS creates a single-file, multi-page SVG document with navigational controls when the **NEWFILE=** option of the ODS PRINTER statement is a value other than PAGE, and one of the following set of system options is specified:

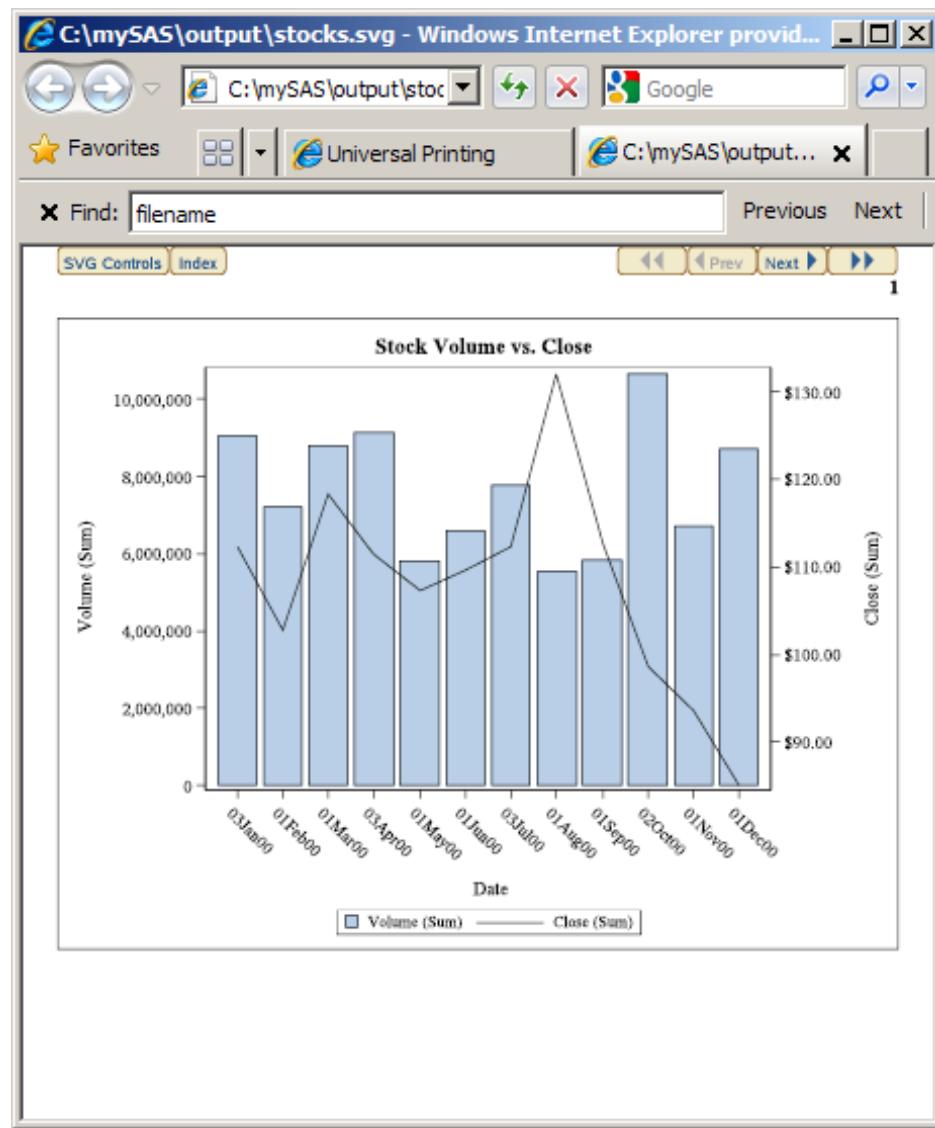
- The **PRINTERPATH=** system option is set to SVG or SVGZ, and the **SVGCONTROLBUTTONS** system option is set.
- The **PRINTERPATH=** system option is set to SVGView.

The SVGView Universal Printer enables the **SVGCONTROLBUTTONS** system option.

If the **SVGCONTROLBUTTONS** system option is not specified or the Universal Printer is not SVGView, the SVG document is created in a continuous-page layout. To navigate the document, you would use your browser controls.

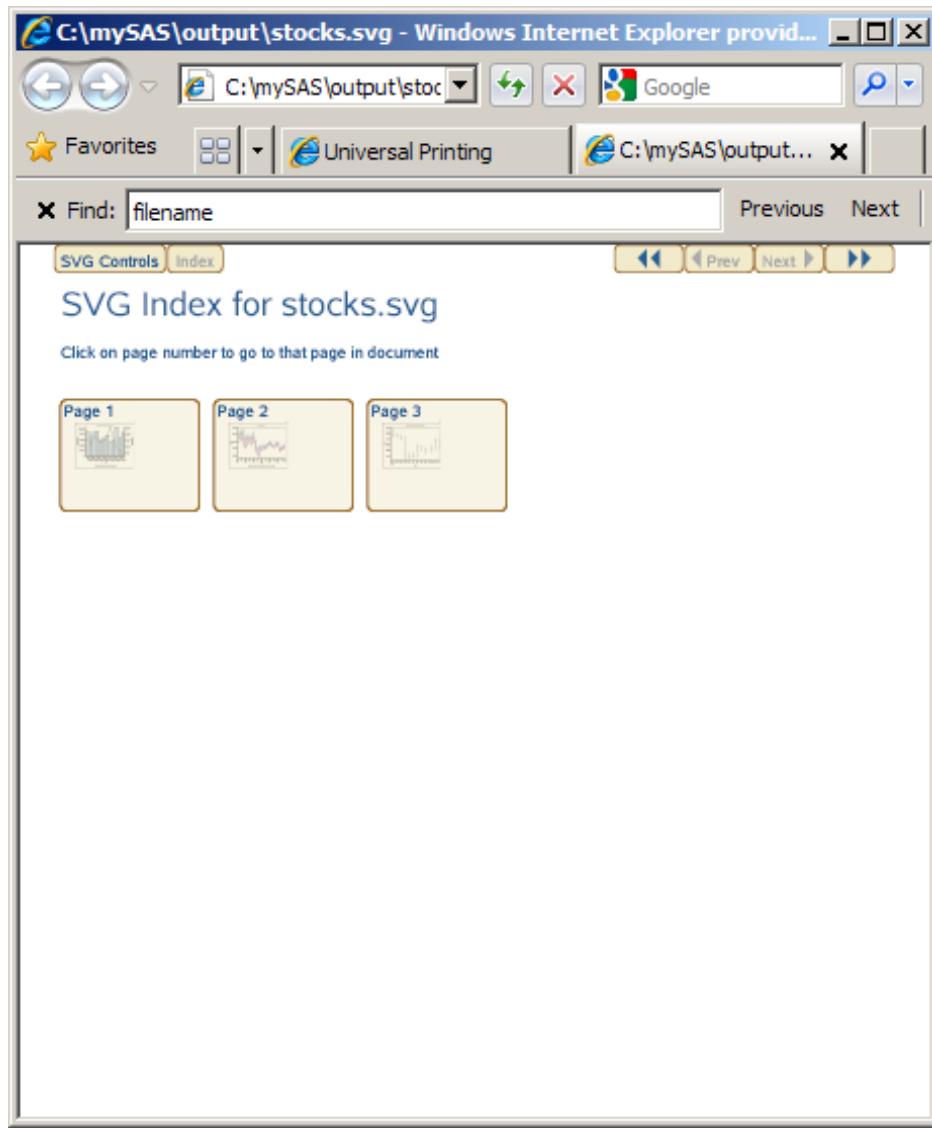
The navigation controls enable you to go to the next page, the previous page, the first page, or the last page; to display an index of all pages; or to hide or show the controls.

Figure 15.37 First Page of a Multi-page SVG File with Navigation Controls



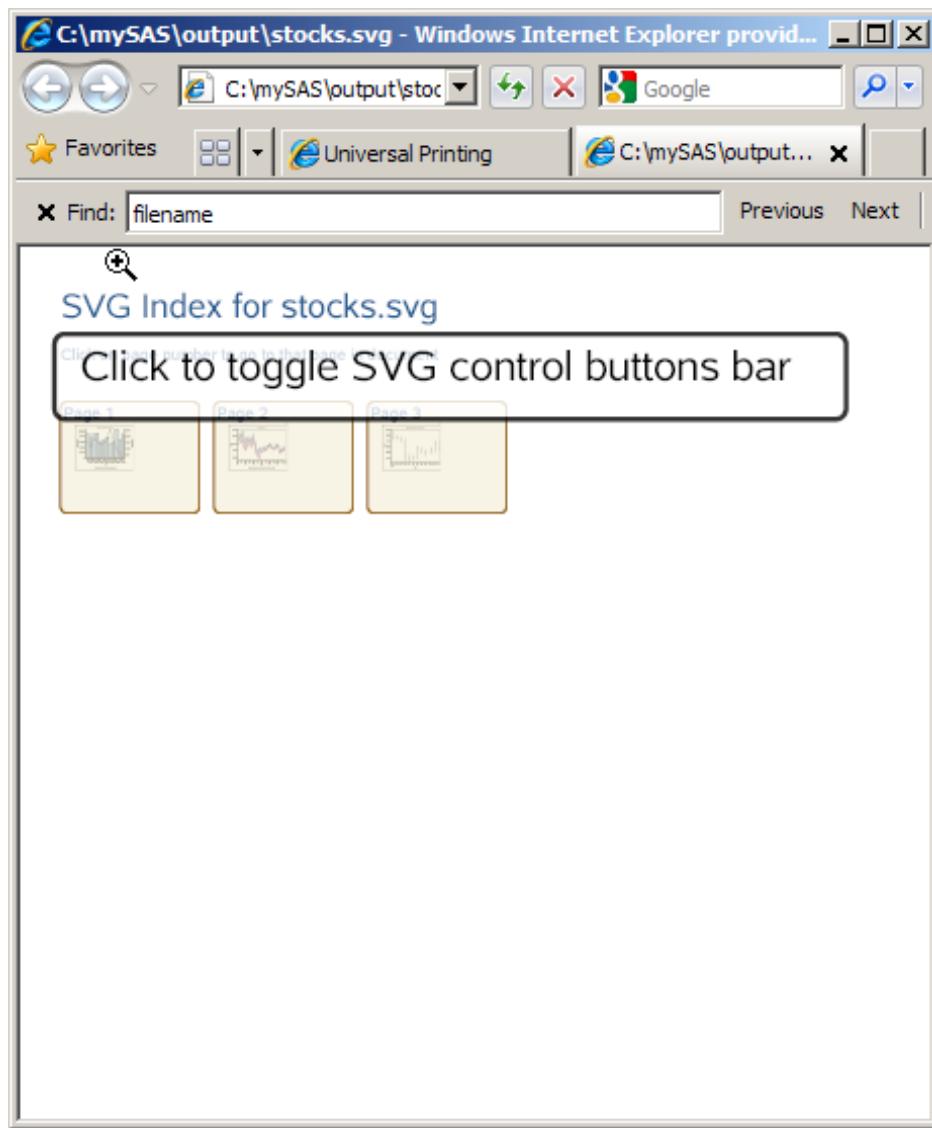
To display an index of all pages in the SVG file, select the **Index** button. To go to a specific page from the index, select the thumbnail image of the page.

Figure 15.38 Index of a Multi-page SVG File with Navigation Controls



You can hide the control buttons by selecting the **SVG Controls** button. The tooltip is displayed when the cursor is over the control. To show the navigation controls again, click in the top area of the output when you see the tooltip **Click to toggle SVG control button bar**. This is useful when you want to print a page in the document without the SVG controls.

Figure 15.39 A Multi-page SVG File That Hides the Navigational Controls



Here is the SAS code that created the stocks.svg file:

```

options nodate printerpath=(svgview stocks) papersize=("6" "6") ;
filename stocks 'c:\mySas\output\stocks.svg';
ods html close;
ods printer;
proc sgplot data=sashelp.stocks (where=(date >= "01jan2000"d
                                         and date <= "01jan2001"d
                                         and stock = "IBM"));
    title "Stock Volume vs. Close";
    vbar date / response=volume;
    vline date / response=close y2axis;
run;
title;
proc sgplot data=sashelp.stocks
    (where=(date >= "01jan2000"d and stock = "IBM"));
    title "Stock Trend";
    series x=date y=close;
    series x=date y=low;

```

```

      series x=date y=high;
run;
title;
title "Stock High, Low, and Close";
proc sgplot data=sashelp.stocks;
  where Date >= '01JAN2005'd and stock='IBM';
  highlow x=date high=high low=low
    / close=close;
run;
title;
ods printer close;
ods html;

```

For information about the NEWFILE= option, see “[ODS PRINTER Statement](#)” in [SAS Output Delivery System: User’s Guide](#).

## Animating Multi-Page SVG Files

You can animate multi-page SVG files using SAS system options. For more information, see “[Creating Animated GIF Images and SVG Documents](#)” on page [375](#).

## Creating Separate Files for Multi-Page SVG Documents

You can create a separate file for each page in an SVG document by specifying the NEWFILE=PAGE option in the ODS PRINTER statement. A new page is created when a procedure explicitly starts a new page and not when the page size is exceeded. The first file is named *filename*.svg. Subsequent filenames have a number appended, starting with the number 1: *filename*1.svg, *filename*2.svg, and so on.

Using the default filename sasprt.svg, the following code creates three files:

- sasprt.svg contains the output from the first SGPlot procedure.
- sasprt1.svg contains the output from the second SGPlot procedure.
- sasprt2.svg contains the output from the third SGPlot procedure.

```

options nodate printerpath=svgview papersize=("6" "6");
ods html close;
ods printer newfile=page;
proc sgplot data=sashelp.stocks (where=(date >= "01jan2000"d
                                         and date <= "01jan2001"d
                                         and stock = "IBM"));
  title "Stock Volume vs. Close";
  vbar date / response=volume;
  vline date / response=close y2axis;
run;
title;
proc sgplot data=sashelp.stocks
  (where=(date >= "01jan2000"d and stock = "IBM"));
  title "Stock Trend";
  series x=date y=close;
  series x=date y=low;
  series x=date y=high;
run;
title;

```

```

title "Stock High, Low, and Close";
proc sgplot data=sashelp.stocks;
  where Date >= '01JAN2005'd and stock='IBM';
  highlow x=date high=high low=low
    / close=close;
run;
title;
ods printer close;
ods html;

```

For information about the NEWFILE= option, see “[ODS PRINTER Statement](#)” in *SAS Output Delivery System: User’s Guide*.

## Creating Overlaid Transparent SVG Documents

You use the SVTt Universal Printer to create a transparent SVG document in which the pages are transparent and can be overlaid. The following is a SAS program that overlays a bar chart on a map of the United States:

```

data boxanno;
  length function color style $20 text $16;
  retain xsys ysys '2' hsys '3' when 'a';
  set maps.uscity(keep=x city state);
  where city='Raleigh' and state=stfips('NC');
  color='blue'; size=4; text='V'; position='5'; style='marker'; output;
  myx=x;
  myy=y;
  function='move';
  x=myx; y=myy; output;
  function='draw';
  x=myx-.432; y=myy+.0417; color='black'; line=1; size=.2; style='solid'; output;
  function='move';
  x=myx; y=myy; output;
  function='draw';
  x=myx-.432; y=myy+.178; output;
  function='move';
  x=myx; y=myy; output;
  function='draw';
  x=myx-.251; y=myy+.178; output;
  function='move';
  x=myx; y=myy; output;
  function='draw';
  x=myx-.251; y=myy+.0417; output;

run;

%let name=annomap;
filename odsout '..';

goptions reset=all;
/* Close the HTML and LISTING destinations for map creation. */
ods html close;
ods listing close;
options printerpath=svgt nodate nonumber;
ods printer file='annomap.svg' ;

```

```

goptions border;

goptions gunit=pct htext=3 htext=2 ftext="arial/bo"
      iback='c:\public\mySASPrograms\ripple.jpg';
pattern1 v=s c=cornsilk;

title1 c=red "SAS/Graph gmap and Overlayed gchart with printerpath=svgt";
proc gmap data=maps.us map=maps.us ;
id state;
choro state / levels=1 nolegend coutline=blue anno=boxanno
des="" name="&name";
run;

quit;

goptions iback= hsize=2.07 vsize=1.57 horigin=2.1 vorigin=3.12 autosize=on dev=svgt;
/* you must use the default ods style, for transparency to work */

goptions gunit=pct htext=8 ftext="Albany AMT" ;
title c=blue h=10 'Transparent SVG';
axis1 label=none value=none major=none minor=none style=0;
axis2 color=blue label=none offset=(7,7) value=(color=blue);
proc gchart data=sashelp.class;
vbar3d age / discrete patternid=midpoint
descending raxis=axis1 maxis=axis2 width=9 space=5
frame cframe=rgba0195FF50 coutline=blue woutline=1
des="" name="&name.b";
run;

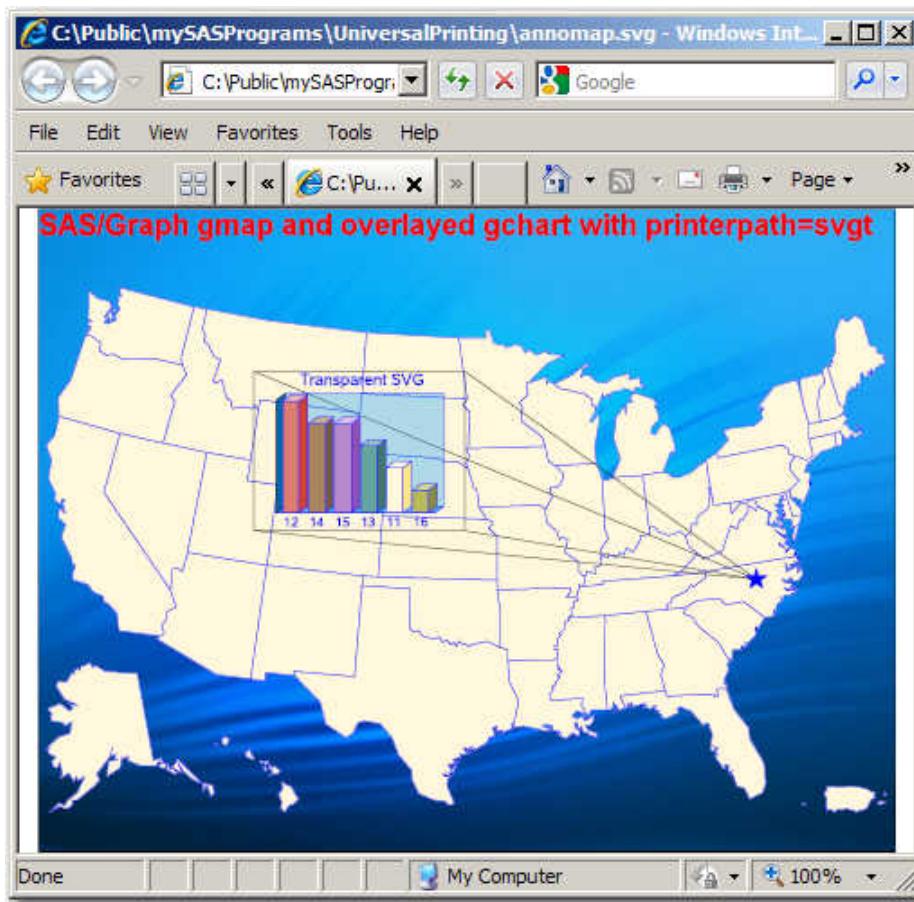
quit;

ods printer close;

```

This code creates the following SVG:

Figure 15.40 A Bar Chart Overlaying a SAS/GRAPH Map



See “Enhancing Drill-Down Behavior in SVG Presentations Using HTML Attributes” in *SAS/GRAPH: Reference* for an example of using overlaid images for drill-down links in graphs.

## SVG Documents in HTML Files

### Overview of SVG Documents in HTML Files

To view an SVG document in an HTML file, you either create a link to the SVG document, embed the SVG document in the HTML file, or create an SVG graph that is integrated in the HTML file.

You can embed an SVG document in an HTML file using these methods:

- Create an SVG graph using ODS GRAPHICS and the ODS HTML5 statement with the option `SVG_MODE="EMBED"`.
- Use SAS/GRAPH and run your SAS program using the ODS HTML `DEV=SVG` statement. SAS creates the SVG document and the HTML file, embedding the SVG document in the HTML file using the `<EMBED>` element.

- Create an SVG document using the ODS PRINTER statement and the PRINTERPATH=SVG option. Then, embed the SVG document in an HTML file using the <EMBED> element.

You can integrate an SVG graph in an HTML file by using the ODS HTML5 SVG\_MODE='INLINE' statement.

For information about creating SVG document in SAS/GRAFH, see “[Generating SVG, PNG, GIF, and TIFF Graphics](#)” in *SAS/GRAFH: Reference*.

## Linking to an SVG Document

If you link to an SVG document in an HTML document and you are using the default values for the SVG system options, the SVG document opens in the browser window and scales to the size of the viewable area in the window. For an example of an HTML file that links to an SVG document, see [Figure 15.41 on page 369](#) and [Figure 15.42 on page 370](#).

## Embedding SVG Documents in HTML Files

When you embed SVG documents in an HTML file, the height and width attributes of the <EMBED> tag become the dimensions of the viewport. If you use the default values for the SVG system options when you create your SVG document, the SVG document scales to the size of the viewport. This is because there is no default value of the SVGHEIGHT= and SVGWIDTH= system options, which effectively equates to specifying a value of 100%. A value of 100% for these system options scales the SVG document to 100% of the viewport.

If you do not specify height and width attributes on the embed tag, the viewport dimensions are determined by the browser. The embedded document might not render as you expected it to render.

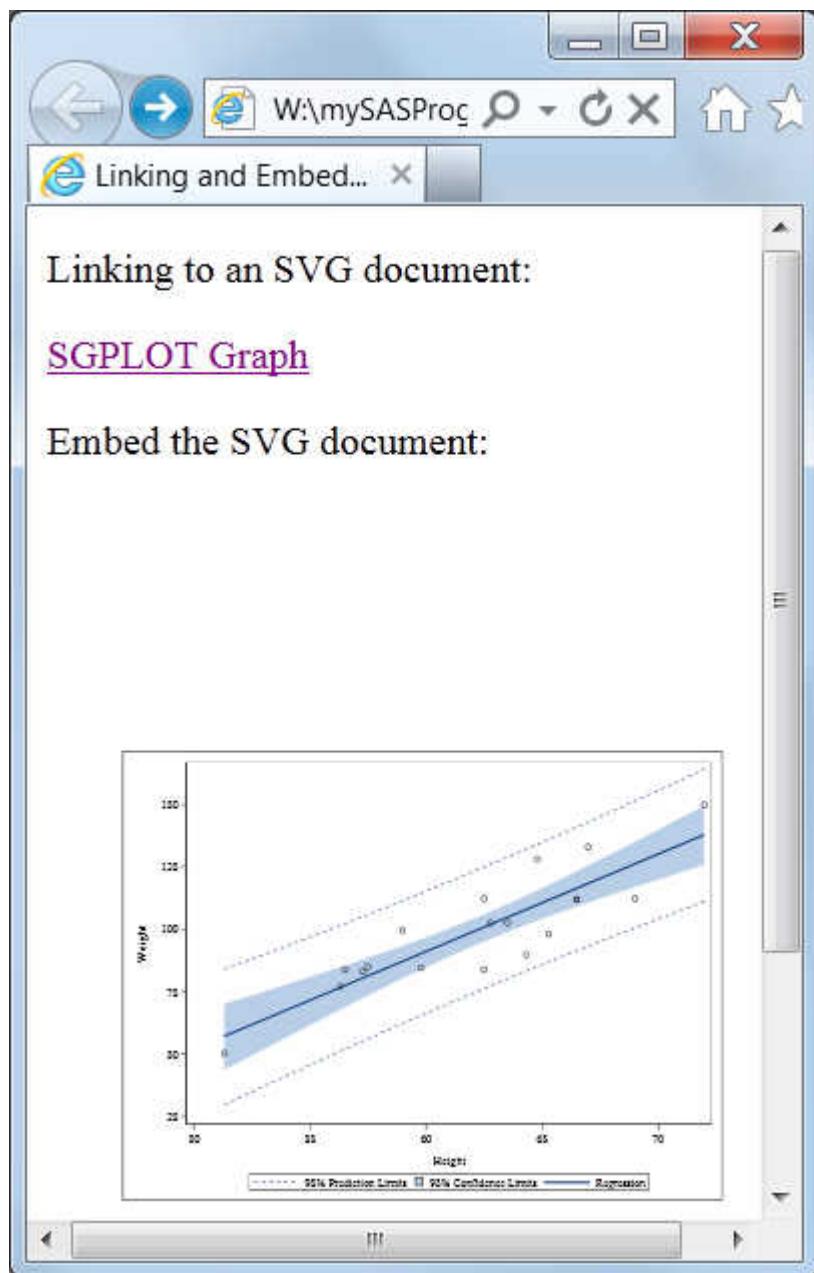
The following HTML file demonstrates linking and embedding a stand-alone SVG document in an HTML file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <title>Linking and Embedding an SVG Document in an HTML Document</title>
  <meta http-equiv="X-UA-Compatible" content="IE=9".

</head>
<body>
<p>Linking to an SVG document:</p>
<a href="sasprt.svg">SGPlot Graph</a>
<p>Embed the SVG document:</p>
<embed src="sasprt.svg" type="image/svg+xml" height="400" width="300">
</body>
</html>
```

Here is the HTML file:

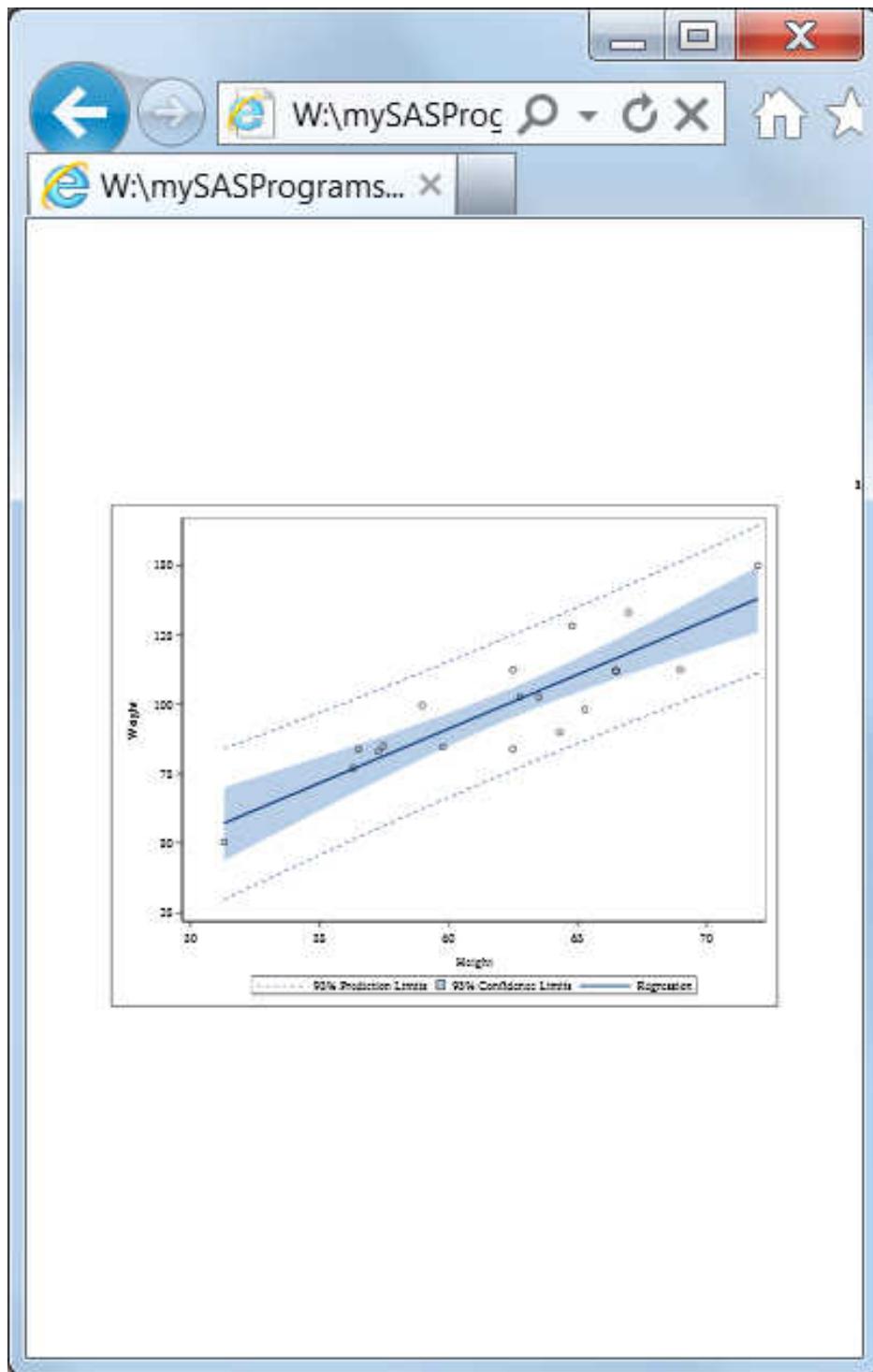
**Figure 15.41** An HTML Document Displaying a Link to a Stand-alone SVG Document and an Embedded SVG Document



The viewport has a height of 400 pixels and a width of 300 pixels. Because the default SVG system option values were used, the SVG document scales to 100% of the viewport.

If you click the SGPLOT Graph link, the browser displays the following SVG document:

Figure 15.42 A Stand-alone SVG Document Displayed as a Result of Clicking an HTML Link



The viewport is the area in the browser window that can be displayed and the SVG document scales to 100% of the viewport.

The following example uses the ODS HTML5 destination to embed an SVG graph in an HTML file:

```
ods html close;  
ods html5 options(svg_mode="embed");
```

```
ods graphics /imagefmt=svg;
proc sgplot data=sashelp.stocks
  (where=(date >= "01jan2000"d and stock = "IBM"));
  title "Stock Trend";
  series x=date y=close;
  series x=date y=low;
  series x=date y=high;
run;
ods html5 close;
ods html;
```

The default `svg_mode` for the HTML5 destination is `INLINE`. In order to embed the SVG graph, you must specify `SVG_MODE="EMBED"` as an option in the ODS HTML5 statement. Here is the `<EMBED>` element in the HTML file:

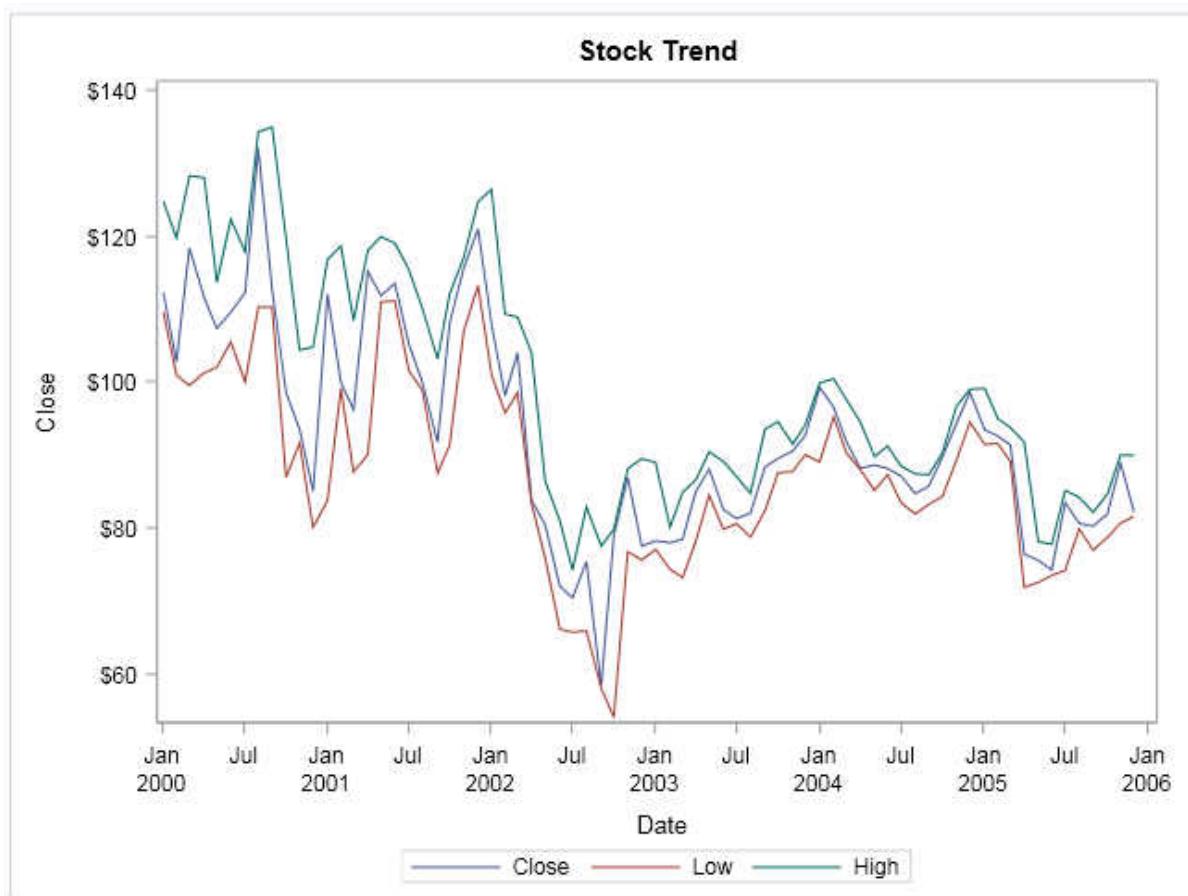
```
<embed style="height: 480px; width: 640px" src="SGPLOT.svg" type="image/svg+xml"/>
```

## Integrating an ODS Graphics SVG Graph in an HTML5 File

To integrate an ODS Graphics SVG graph in an HTML file, you specify the option `SVG_MODE='INLINE'` in the ODS HTML5 statement:

```
ods html close;
ods html5 options(svg_mode="inline");
ods graphics /imagefmt=svg;
proc sgplot data=sashelp.stocks
  (where=(date >= "01jan2000"d and stock = "IBM"));
  title "Stock Trend";
  series x=date y=close;
  series x=date y=low;
  series x=date y=high;
run;
ods html5 close;
ods html;
```

Here is the HTML file with the integrated SVG graph:



## Printing an SVG Document from a Browser

SVG document printing is controlled by the browser. The browser prints only what is displayed in the browser window.

## Creating TIFF Images Using Universal Printing

### TIFF Images in SAS

Tagged image file format (TIFF) images are raster images that are commonly used in word processing, scanning, faxing, and other applications.

SAS supports TIFF 6.0 for RGBA and CMYK colors. When SAS creates a TIFF image, the image is compressed. If the NOUPRINTCOMPRESSION system option is set, the size of a TIFF image file that SAS creates is extremely large.

For a description of the TIFF printer, you can either view the printer in the SAS registry or submit the following QDEVICE procedure and view the output in the SAS log:

```
proc qdevice;
  printer tiff;
run;
```

## See Also

[“Color Support for Universal Printers” on page 273](#)

## The TIFF Universal Printers

SAS provides these Universal Printers:

Printer Name	Description
TIFF	produces TIFF images using RGBA color and transparency.
TIFFk	produces TIFF images using CMYK color. Transparency is not supported.

TIFF printers do not support multiple-page documents. If a procedure creates multiple pages or if more than one procedure is used in the code for ODS PRINTER output, only the first page is viewable.

## Creating a TIFF Image

You can create a TIFF image using the ODS PRINTER statements. You specify the TIFF Universal Printer as the value of the PRINTERPATH= system option or as the value of the PRINTER= option in the ODS PRINTER statement.

Here is sample code to create a TIFF image:

```
ods html close;
ods printer printer=tiff;

...more SAS code...

ods printer close;
ods html;
```

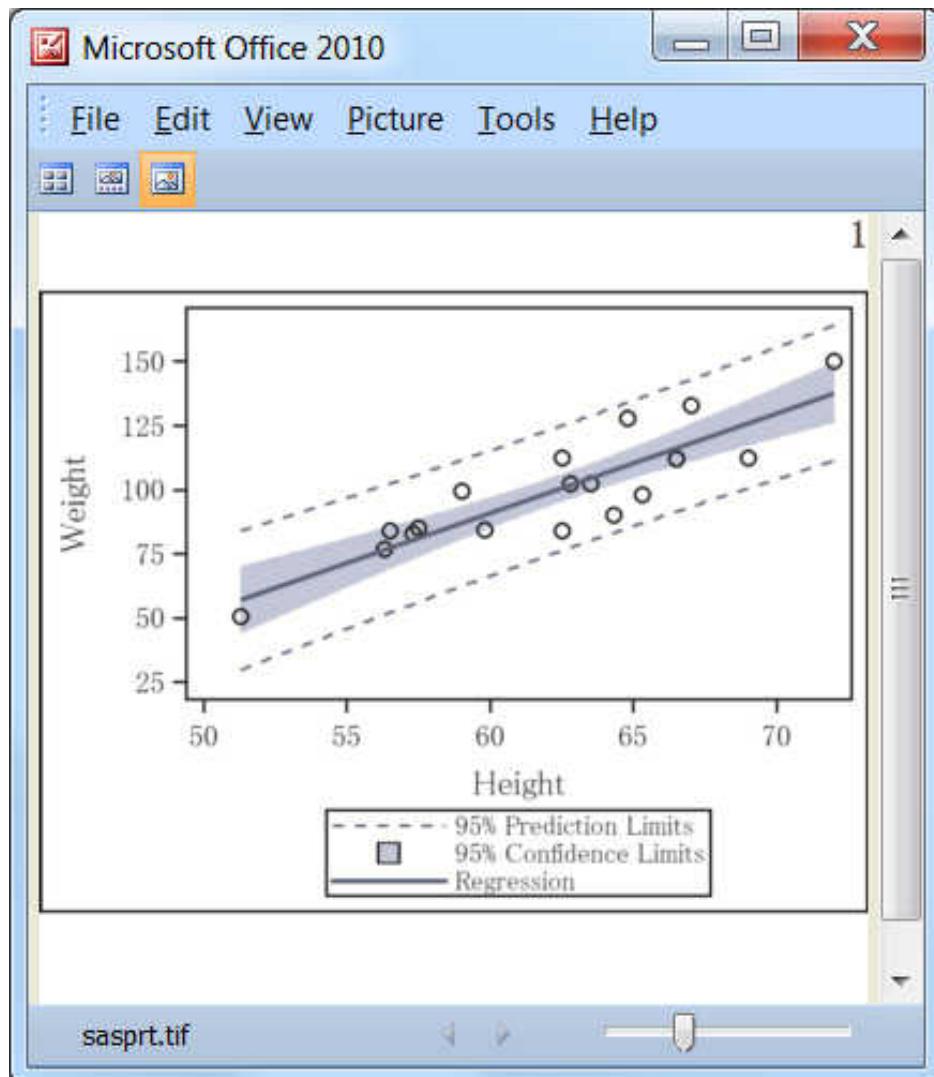
SAS creates the file sasprt.tif in the current directory.

In SAS/GRAFPH, the TIFF device is a shortcut to the TIFF Universal Printer. For information about creating TIFF images using SAS/GRAFPH devices, see [SAS/GRAFPH: Reference](#).

## Example of Creating a TIFF Image Using the ODS PRINTER Statement

Using the example data set Sashelp.Class and the SGPOINT procedure, the following ODS PRINTER statement prints the TIFF file sasprt.tif in the current directory:

```
options printerpath=tiff papersize=("4in" "4in") nodate;
ods html close;
ods printer;
proc sgplot data=sashelp.class;
  reg x=height y=weight / CLM CLI;
run;
ods printer close;
ods html;
```



---

# Creating Animated GIF Images and SVG Documents

---

## About Animated GIF Images and SVG Documents

When you create a multi-page GIF image or SVG document using the ODS PRINTER destination, you can animate the GIF image or SVG document that is created by setting SAS system options. Each page in the GIF image or SVG document creates one frame in the output file. The system options enable you to configure these animation attributes:

- start or stop creating an animated file
- the amount of time that a frame is in view
- whether frames are overlaid or are played sequentially
- the number of times an animation loop is repeated
- for SVG documents only, whether to immediately start the animation when the document is loaded in the web page
- for SVG documents only, whether a frame fades in and out of view and if during the fade-in and fade-out time, the frames are overlaid or played sequentially

SAS/GRAF is required to create animated files for the ODS HTML5, ODS HTML, and the ODS LISTING destinations. For more information, see [SAS/GRAF: Reference](#).

You set the options using the OPTIONS statement before opening the ODS PRINTER destination:

```
options printerpath=gif animation=start animduration=5 animloop=yes noanimoverlay;
ods printer file='myfile.gif';
```

In this OPTIONS statement, the ANIMATION option starts creating the animation file, the ANIMDURATION option specifies that each frame is held for 5 seconds. The ANIMLOOP option specifies to continuously repeat the animation loop. The NOANIMOVERLAY option specifies that each frame is played sequentially.

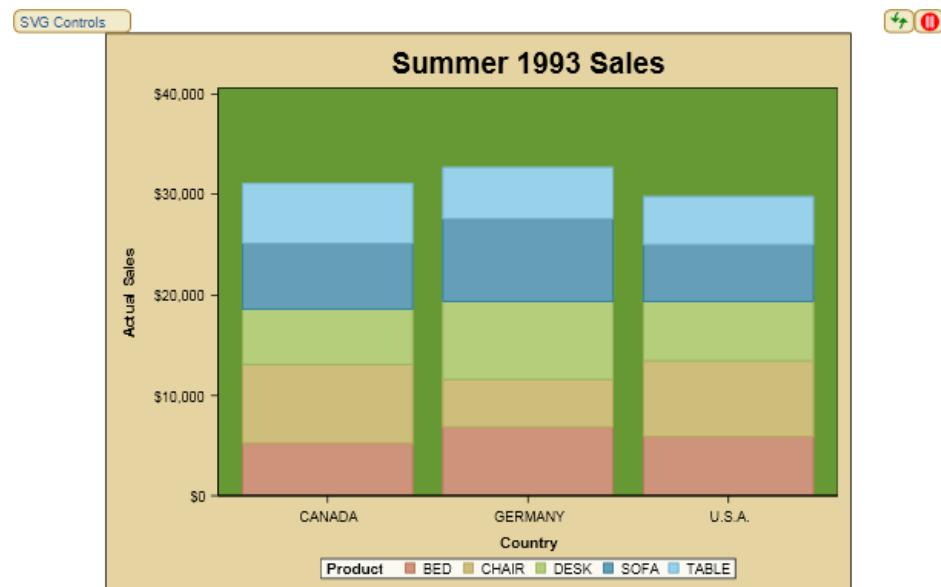
When the PRINTERPATH= option is set to SVG, you can use the SVG animation options to configure the fade-in and fade-out attributes and the autoplay attribute. The animation options that begin with SVG do not affect GIF images.

After you set the options and opened the PRINTER destination, proceed with your SAS code to create each frame in your file. The animation frame is created when you run SAS procedures. You can use animation options in between procedures to change the duration that a frame is held in view and the fade-in and fade-out times. For example, you can hold a particular frame in view for a longer period of time. You would use the OPTIONS ANIMDURATION= statement before a procedure to increase the time that the frame is held in view. Specify ANIMATION=STOP to end the creation of the animation file. Use the ODS PRINTER CLOSE statement to close the file.

**TIP** Be sure to specify ANIMATION=STOP after you create the frames for your animation file. If ANIMATION=START remains set, you might create an animation file unintentionally for subsequent procedure statements.

To embed the file in a web page or to create a link to the file from a web page, see “[SVG Documents in HTML Files](#)” on page 367.

When an animated file is displayed in a browser, the animation control buttons can be used to reset (  ) the animation, to pause (  ) the animation, and to play (  ) the animation. You can toggle **SVG Controls** to show or hide the control buttons. Here is one frame of an animated SVG document with the control buttons:



If you are creating SVG files that will be viewed on an iPad, a best practice is to use the SVGVIEW Universal Printer for optimal sizing.

You can view animated SVG files using Internet Explorer 9 or later. You can view animated GIF files in releases prior to Internet Explorer 9.

## Animation System Options

You use the animation system options to configure the attributes of an animation GIF or SVG file. All of the system options can be configured for SVG documents. The options that begin with SVG are not applicable for GIF images. Here are the animation system options.

*Table 15.22 Animation System Options and Valid Universal Printers*

Description	Option Name	Valid Universal Printers
Starts or stops the creation of an animation file.	ANIMATION	GIF and SVG

Description	Option Name	Valid Universal Printers
Specifies the amount of time that each frame in an animation is held in view.	<a href="#">ANIMDURATION=</a>	GIF and SVG
Specifies whether the animation loop is played continuously or is played once, or specifies a specific number of times that the animation loop is repeated.  ANIMLOOP=YES sets a continuous loop. ANIMLOOP=NO completes one loop. ANIMLOOP= <i>n</i> specifies a specific number of times to loop.  For SVG documents, use ANIMLOOP=YES and ANIMLOOP=NO. Setting ANIMLOOP= <i>n</i> where <i>n</i> > 0 for SVG documents completes only one loop for the document.	<a href="#">ANIMLOOP=</a>	GIF and SVG
Specifies whether animation frames are overlaid or if they are played sequentially.  If you overlay frames, your frames would require some level of transparency for the output not to appear overwritten.	<a href="#">ANIMOVERLAY</a>	GIF and SVG
Specifies whether an SVG animation starts immediately in the web browser.  If you specify NOSVGAUTOPLAY, start the animation by clicking  .	<a href="#">SVGAUTOPLAY</a>	SVG
Specifies the number of seconds for an SVG frame to fade into view.	<a href="#">SVGFADEIN=</a>	SVG
Specifies whether an SVG frame overlaps the previous frame or if each frame is played sequentially when a frame is fading in and out.	<a href="#">SVGFADEMODE=</a>	SVG
Specifies the number of seconds for an SVG frame to fade out of view.	<a href="#">SVGFADEOUT=</a>	SVG

## Example: Creating an Animated SVG Document

The data set sashelp.prdsale contains office and furniture sales data for the years 1993 and 1994. This example uses the SGLOT procedure to plot a vertical bar that displays the actual sales numbers of office and furniture products for Canada, Germany, and the United States. The plot uses a vertical bar for each country. Each vertical bar shows the data for the different products. The example groups the data by quarters for each year, creating eight charts. When the animation plays, each SVG frame is a chart for one of the quarters. The chart displays a color for each season, and the product sales values change in each vertical bar. To see the animation play for this example, go to support.sas.com. Under **Knowledge Base**,

select **Samples & SAS Notes**. Search for **SVG animation**, In the search results, look for the program **seasons.sas**.

**Create a data set for each quarter for the years 1993 and 1994.** Each DATA step uses a WHERE clause to create a data set by year and quarter. The KEEP option in the SET statement specifies the variables that are in each of the data sets.

```
data work.q1y93 (where=(year=1993 and quarter=1));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q2y93 (where=(year=1993 and quarter=2));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q3y93 (where=(year=1993 and quarter=3));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q4y93 (where=(year=1993 and quarter=4));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q1y94 (where=(year=1994 and quarter=1));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q2y94 (where=(year=1994 and quarter=2));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q3y94 (where=(year=1994 and quarter=3));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;

data work.q4y94 (where=(year=1994 and quarter=4));
set sashelp.prdsale(keep=Actual Country Product Quarter Year);
run;
```

**Create a style for each season.** The four TEMPLATE procedures create a style for each season by specifying seasonal colors for the different parts of the chart. The colors for the vertical bars are not part of the style because they are automatically generated by the SGLOT procedure.

```
proc template;
define style winter;
parent = Styles.meadow;
style body from body;

style GraphColors from GraphColors /
  "gborderlines" = cx000000
  "greferencelines" = cx000000
  "gaxis" = cx000000
  "gwalls" = cx83838C
  ;
style GraphBackground /
  Color = cxB3B2BF
```

```
;  
end;  
quit;  
  
proc template;  
define style spring;  
parent = Styles.meadow;  
    style body from body;  
  
style GraphColors from GraphColors /  
    "gborderlines" = cx000000  
    "greferencelines" = cx000000  
    "gaxis" = cx000000  
    "gwalls" = cxFF9999  
    ;  
style GraphBackground from GraphBackground "Graph background attributes" /  
    Color = cxFFFF99  
    ;  
end;  
quit;  
  
proc template;  
define style summer;  
parent = Styles.meadow;  
    style body from body;  
  
style GraphColors from GraphColors /  
    "gborderlines" = cx000000  
    "greferencelines" = cx000000  
    "gaxis" = cx000000  
    "gwalls" = cx669933  
    ;  
style GraphBackground from GraphBackground "Graph background attributes" /  
    Color = cxE5D4A1  
    ;  
end;  
quit;  
  
proc template;  
define style fall;  
parent = Styles.meadow;  
    style body from body;  
  
style GraphColors from GraphColors /  
    "gborderlines" = cx000000  
    "greferencelines" = cx000000  
    "gaxis" = cx000000  
    "gwalls" = cx996633  
    ;  
style GraphBackground from GraphBackground "Graph background attributes" /  
    Color = cxD9A465  
    ;  
end;  
quit;
```

```
ODS LISTING CLOSE;
```

**Set the options to create an animated file for an SVG document.** Set the PRINTERPATH= option to create an SVG document. The ANIMATION= option starts creating the animation. Each page in the animation is held in view for 3 seconds as specified in the ANIMDURATION option. The SVGFADEN=0 and SVGFADOUT=0 options specify that the pages do not fade in or out of view. The NOANIMOVERTLAY option specifies that the pages are played sequentially. The ODS PRINTER CLOSE statement closes any files that are open for the PRINTER destination. Because the ANIMLOOP= option is not specified, the default of YES is used and the animations loop continuously.

```
options reset=all;
options printerpath=svg animate=start animduration=3 svgfadein=0 svgfadeout=0
noanimoverlays nodate nonumber;

ODS PRINTER CLOSE;
```

**Create an SVG document for each quarter, using the SGLOT procedure.** The %LET macro variable is used to name the SVG file. The first ODS PRINTER statement opens the PRINTER destination and creates the SVG file. After the first ODS PRINTER statement, an ODS PRINTER statement is used before each procedure to specify the style that indicates the seasonal colors to use to create a chart. The TITLE statement specifies the season and the year that is reported. Each SGLOT procedure plots the sales for each country by using identical VBAR and YAXIS options in each procedure. The vbar country / response=actual group=product; statement specifies to create a vertical bar for each country. Each vertical bar contains sales data for each product. The visual aspects for each product in vertical bar are automatically determined by the SGLOT procedure. The YAXIS statement specifies the values to plot for the Y axis.

```
%let name=seasons;

ODS PRINTER file="&name..svg";
ods printer style=winter;
title1 h=18pt "Winter 1993 Sales";
proc sgplot data=work.q1y93 uniform=all;
vbar country / response=actual group=product;
yaxis values=(0 to 40000 by 10000);
run;

ods printer style=spring;
title1 h=18pt "Spring 1993 Sales";
proc sgplot data=work.q2y93 uniform=all;
vbar country / response=actual group=product;
yaxis values=(0 to 40000 by 10000);
run;

ods printer style=summer;
title1 h=18pt "Summer 1993 Sales";
proc sgplot data=work.q3y93 uniform=all;
vbar country / response=actual group=product;
yaxis values=(0 to 40000 by 10000);
run;
```

```

ods printer style=fall;
title1 h=18pt "Fall 1993 Sales";
proc sgplot data=work.q4y93 uniform=all;
vbar country / response=actual group=product ;
yaxis values=(0 to 40000 by 10000);
run;

ods printer style=winter;
title1 h=18pt "Winter 1994 Sales";
proc sgplot data=work.q1y94 uniform=all;
vbar country / response=actual group=product ;
yaxis values=(0 to 40000 by 10000);
run;

ods printer style=spring;
title1 h=18pt "Spring 1994 Sales";
proc sgplot data=work.q2y94 uniform=all;
vbar country / response=actual group=product ;
yaxis values=(0 to 40000 by 10000);
run;

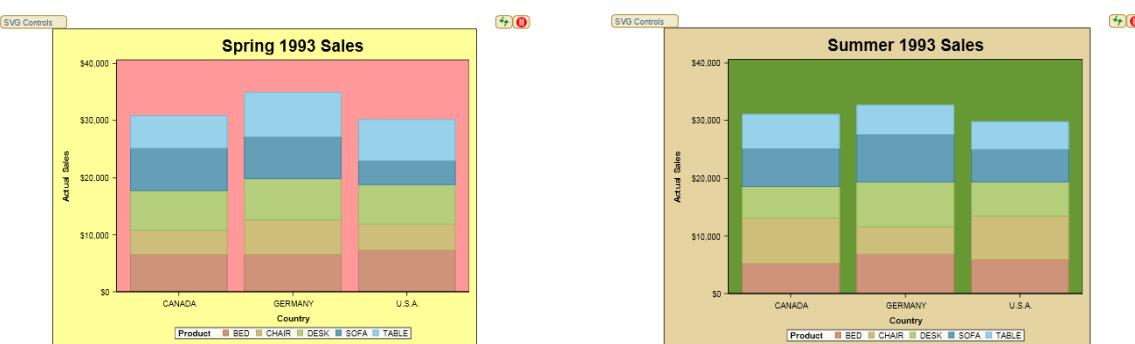
ods printer style=summer;
title1 h=18pt "Summer 1994 Sales";
proc sgplot data=work.q3y94 uniform=all;
vbar country / response=actual group=product ;
yaxis values=(0 to 40000 by 10000);
run;

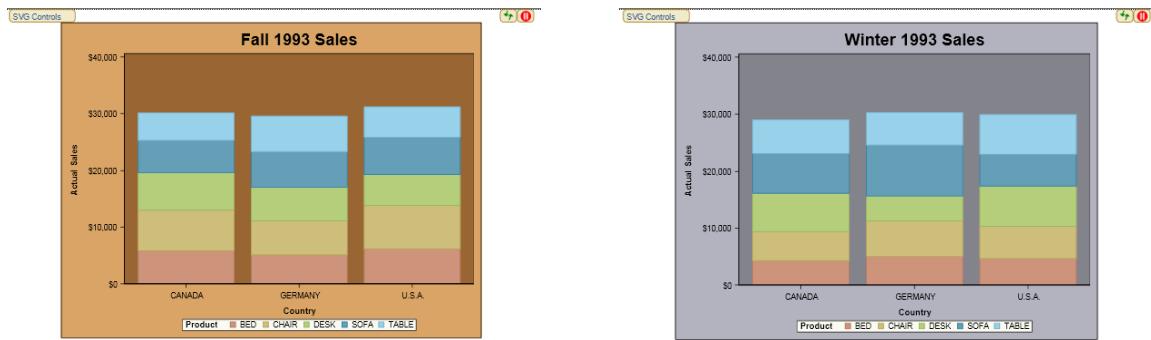
ods printer style=fall;
title1 h=18pt "Fall 1994 Sales";
proc sgplot data=work.q4y94 uniform=all;
vbar country / response=actual group=product ;
yaxis values=(0 to 40000 by 10000);
run;

options animation=stop;
ods printer close;

```

Here are the charts that were created for 1993 by season:





**PART 2**

# Windowing Environment

*Chapter 16*

*Introduction to the SAS Windowing Environment* ..... **385**

*Chapter 17*

*Managing Your Data in the SAS Windowing Environment* ..... **405**



# 16

## Introduction to the SAS Windowing Environment

---

<i>What Is the SAS Windowing Environment</i> .....	385
<i>Main Windows in the SAS Windowing Environment</i> .....	386
Overview of SAS Windows .....	386
SAS Explorer Window .....	387
Enhanced Editor Window .....	388
Log Window .....	389
Results Window .....	390
Output Window .....	391
<i>Navigating in the SAS Windowing Environment</i> .....	394
Overview of SAS Navigation .....	394
Menus in SAS .....	395
Toolbars in SAS .....	398
The Command Line .....	398
<i>Getting Help in SAS</i> .....	399
Type Help in the Command Line .....	399
Open the Help Menu from the Toolbar .....	400
Click Help in Individual SAS Windows .....	401
<i>List of SAS Windows and Window Commands</i> .....	401

---

## 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.

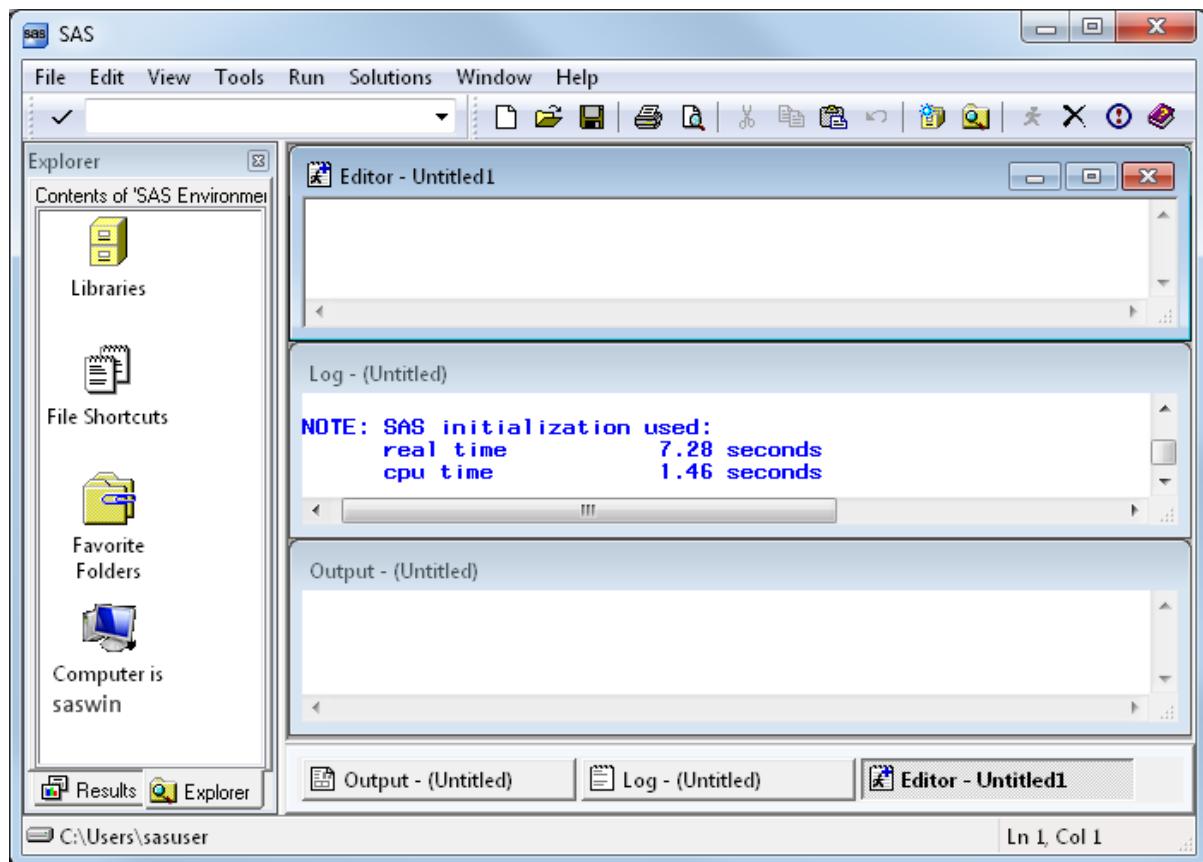
**Note:** The arrangement of your SAS windows depends on your operating environment. For example, in the Microsoft Windows operating environment, the Enhanced Editor window appears instead of the Program Editor.

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 16.1** 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**  $\Rightarrow$  **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.

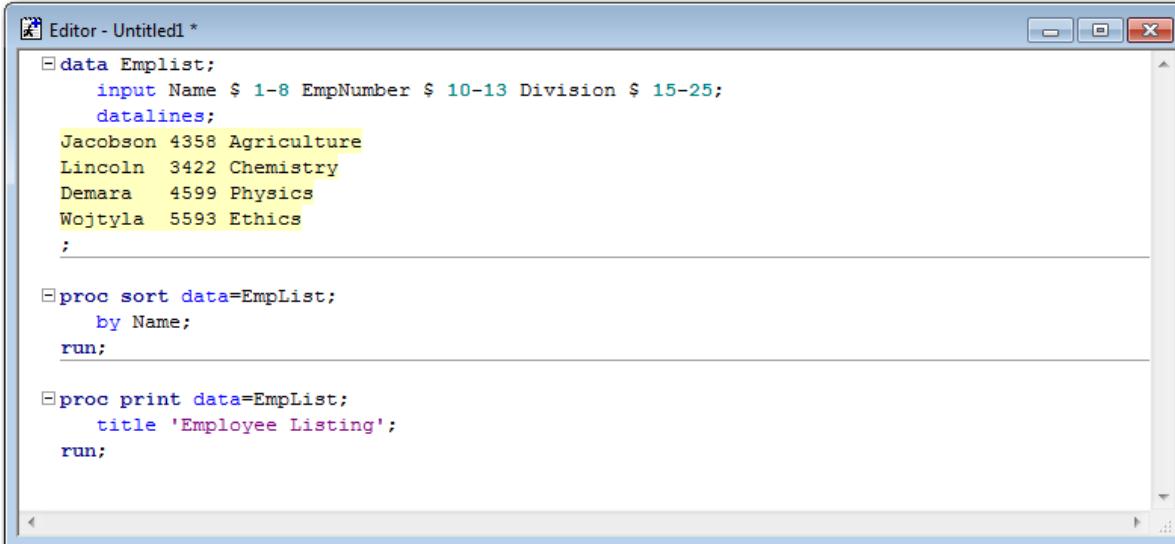
### Open the Enhanced Editor Window

To open the Enhanced Editor window select **View**  $\Rightarrow$  **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 16.2** Example of the Enhanced Editor Window


The screenshot shows the Enhanced Editor window titled "Editor - Untitled1 \*". The window contains the following SAS code:

```

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;

```

The data step displays four records with columns for Name, EmpNumber, and Division. The proc sort and proc print steps are used to sort the data by Name and then print it with a title.

**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**  $\Rightarrow$  **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.

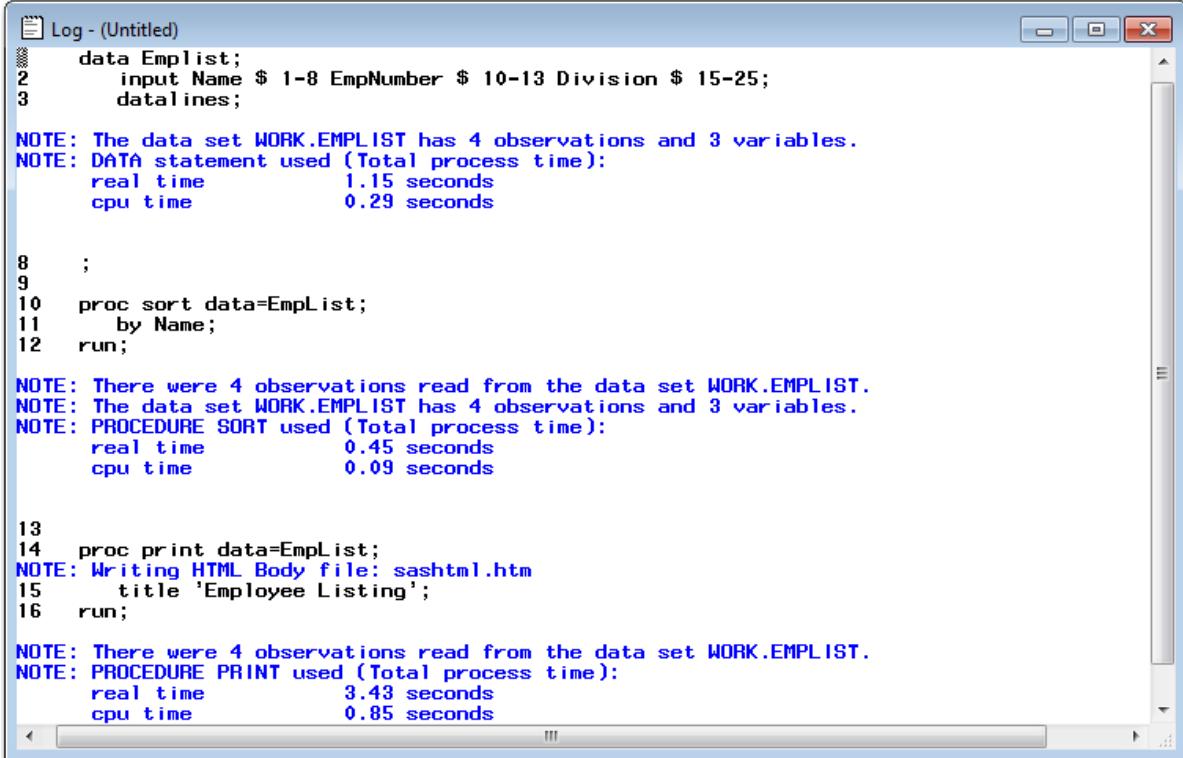
Menu:

Select **View**  $\Rightarrow$  **Log**.

### View Log Output

The following is an example of Log output.

Figure 16.3 Example of Output in the Log Window



The screenshot shows the SAS Log window titled "Log - (Untitled)". The window displays the following SAS code and its execution results:

```

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.

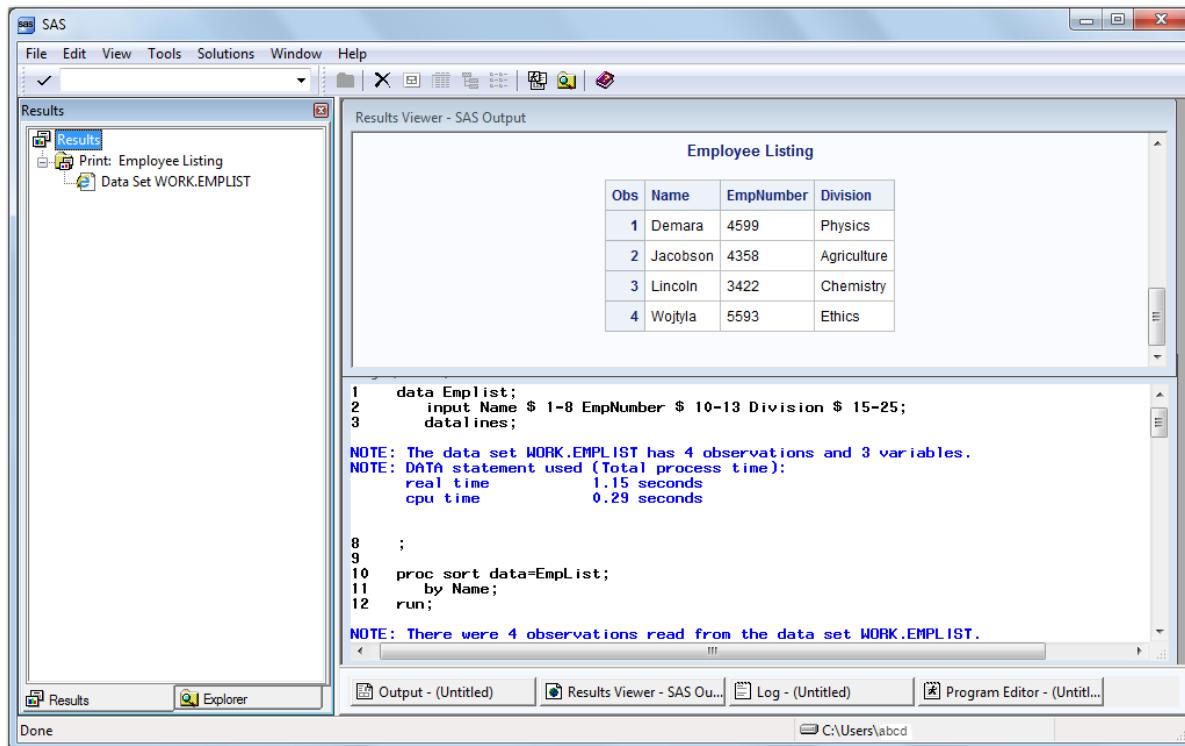
Menu:

Select **View**  $\Rightarrow$  **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 16.4** 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:

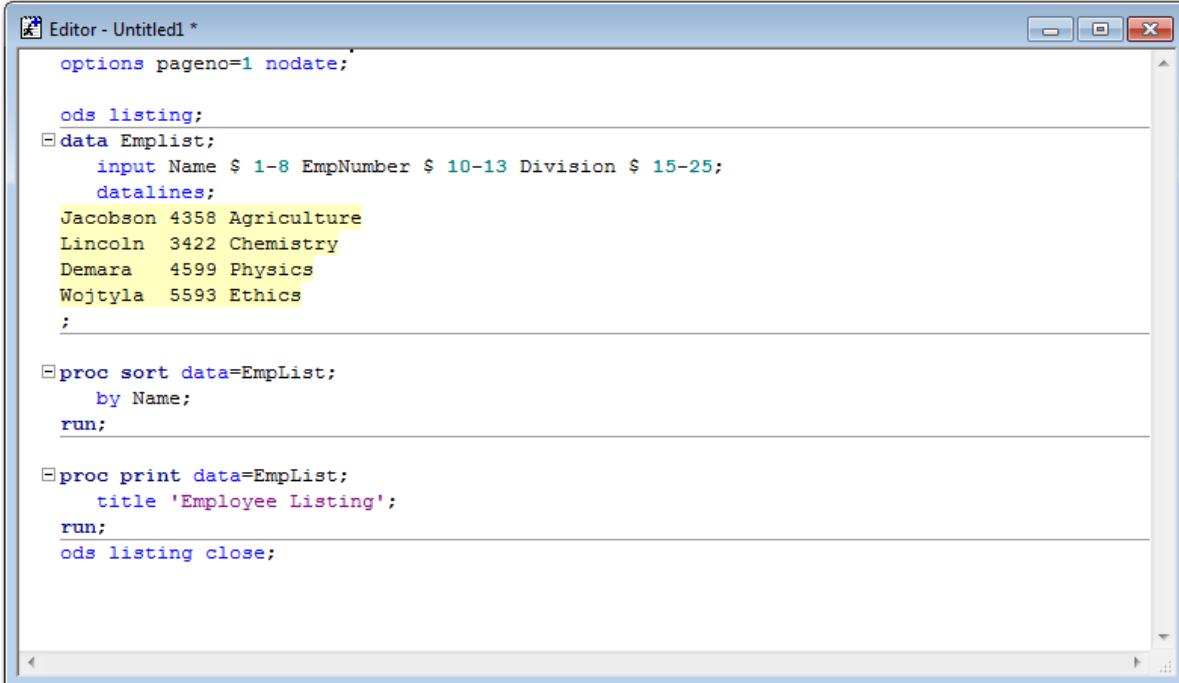
Select **View**  $\Rightarrow$  **Output**.

## 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 16.5 Example of a Program That Produces Listing Output**



The screenshot shows the SAS Editor window titled "Editor - Untitled1 \*". The code in the editor is as follows:

```
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;
```

The data step displays the following output:

Name	EmpNumber	Division
Jacobson	4358	Agriculture
Lincoln	3422	Chemistry
Demara	4599	Physics
Wojtyla	5593	Ethics

SAS creates the following LISTING output:

**Figure 16.6** Example of Listing Output in the Output Window

The screenshot shows a Windows application window titled "Output - (Untitled)". The main content is a table titled "Employee Listing". The table has four columns: "Obs", "Name", "Emp Number", and "Division". There are four rows of data:

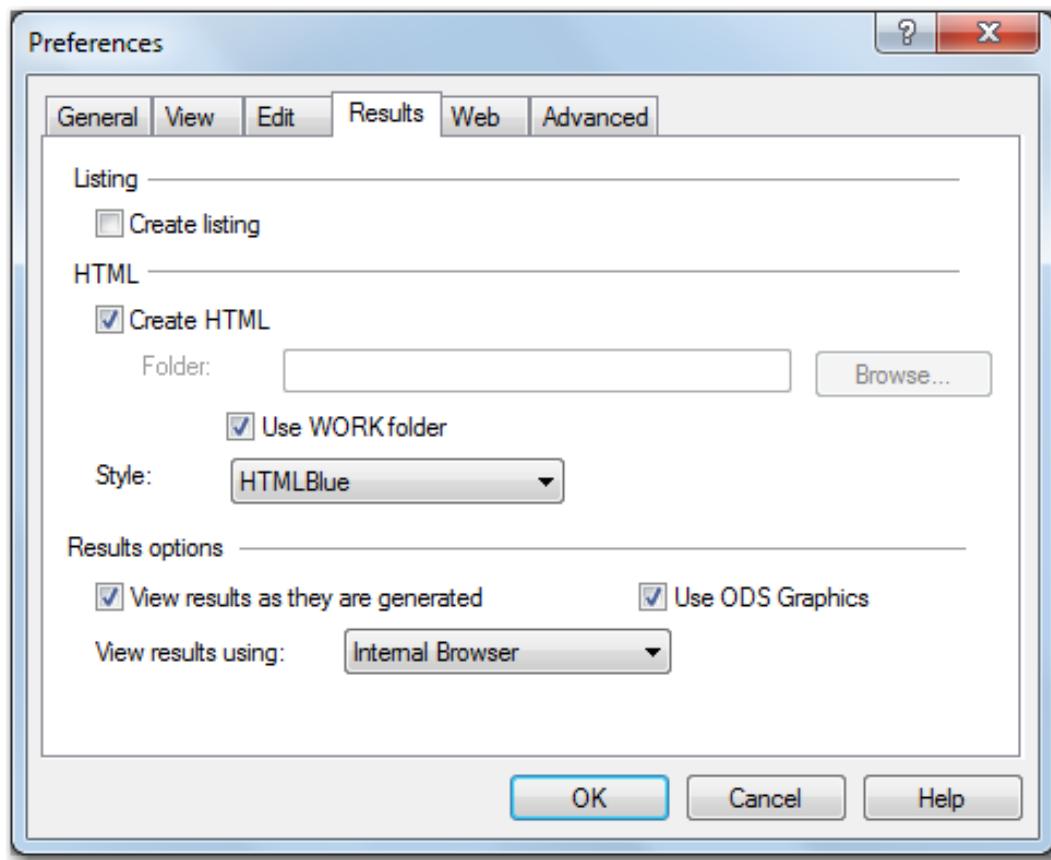
Obs	Name	Emp Number	Division
1	Demara	4599	Physics
2	Jacobson	4358	Agriculture
3	Lincoln	3422	Chemistry
4	Wojtyla	5593	Ethics

## 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**  $\Rightarrow$  **Options**  $\Rightarrow$  **Preferences**.

The following is an example of the Preferences dialog box, with the **Results** tab selected:

Figure 16.7 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

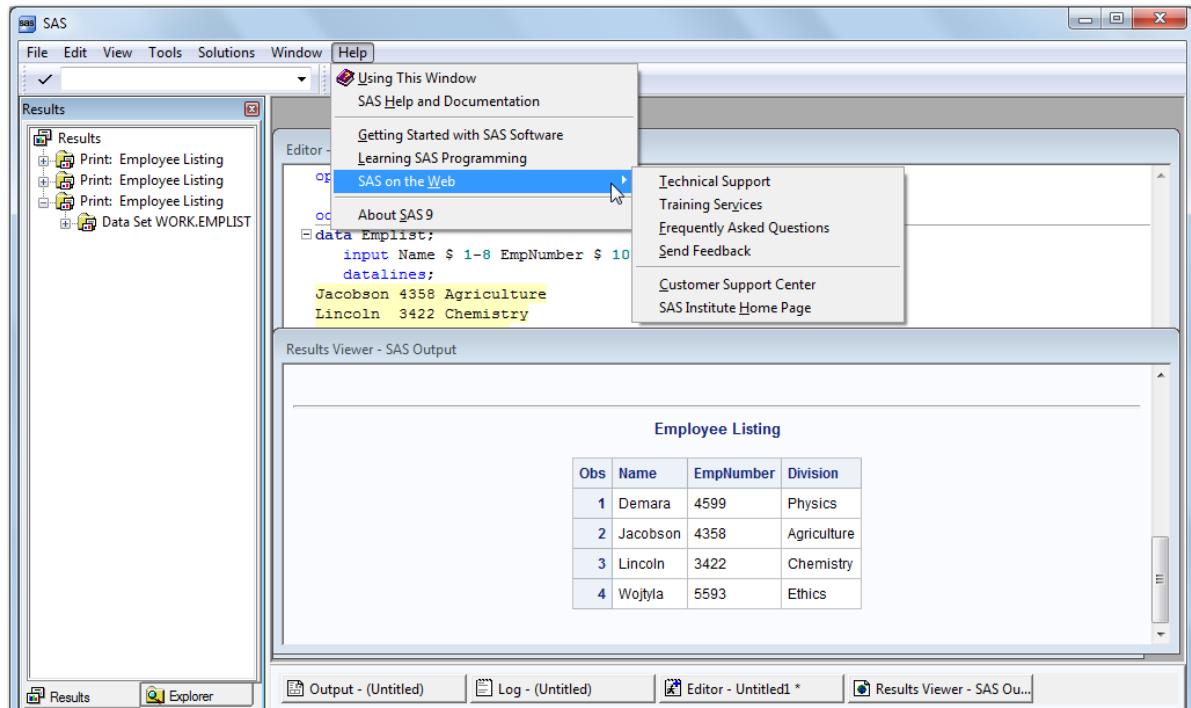
SAS windows have several features that work in a similar manner across all operating environments: menus, toolbars, and online Help. You can customize many of these features by selecting **Tools**  $\Rightarrow$  **Customize** from the menu. For specific information about these features, see the documentation for your operating environment.

## Menus in SAS

Menus contain lists of options that you can select.

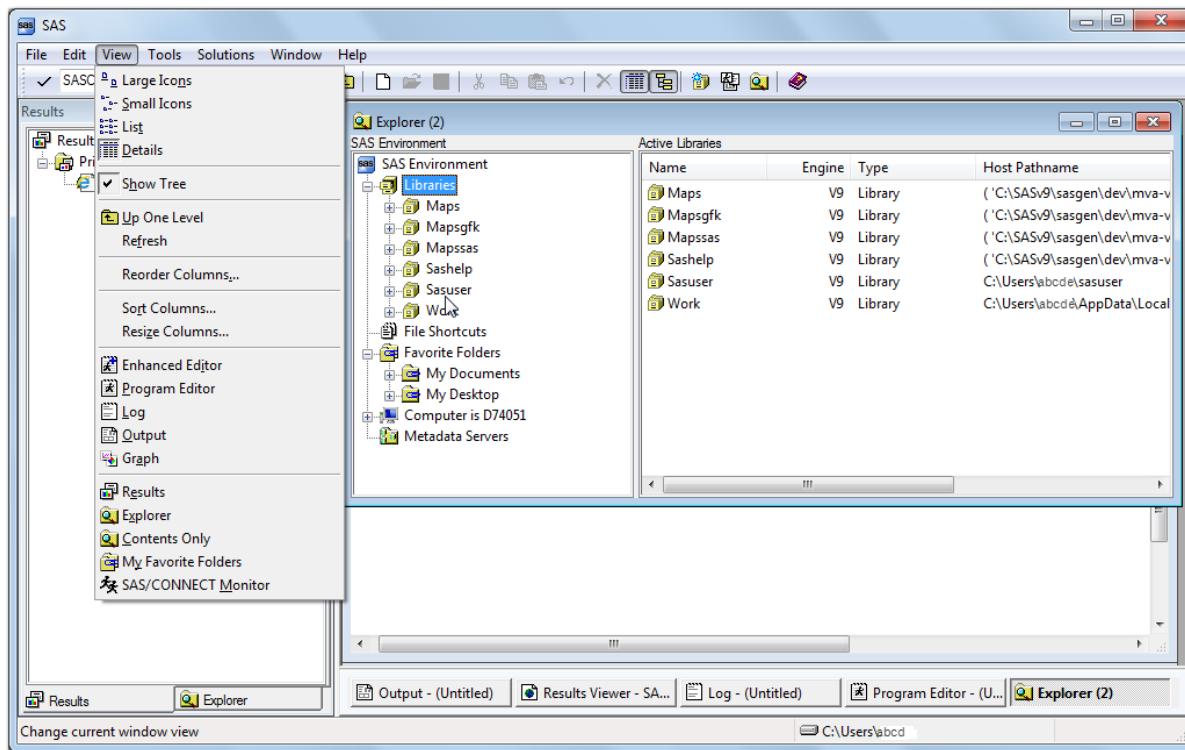
The following example shows the menu options that are available when you select **Help** from the menu bar:

**Figure 16.8** The Help Menu



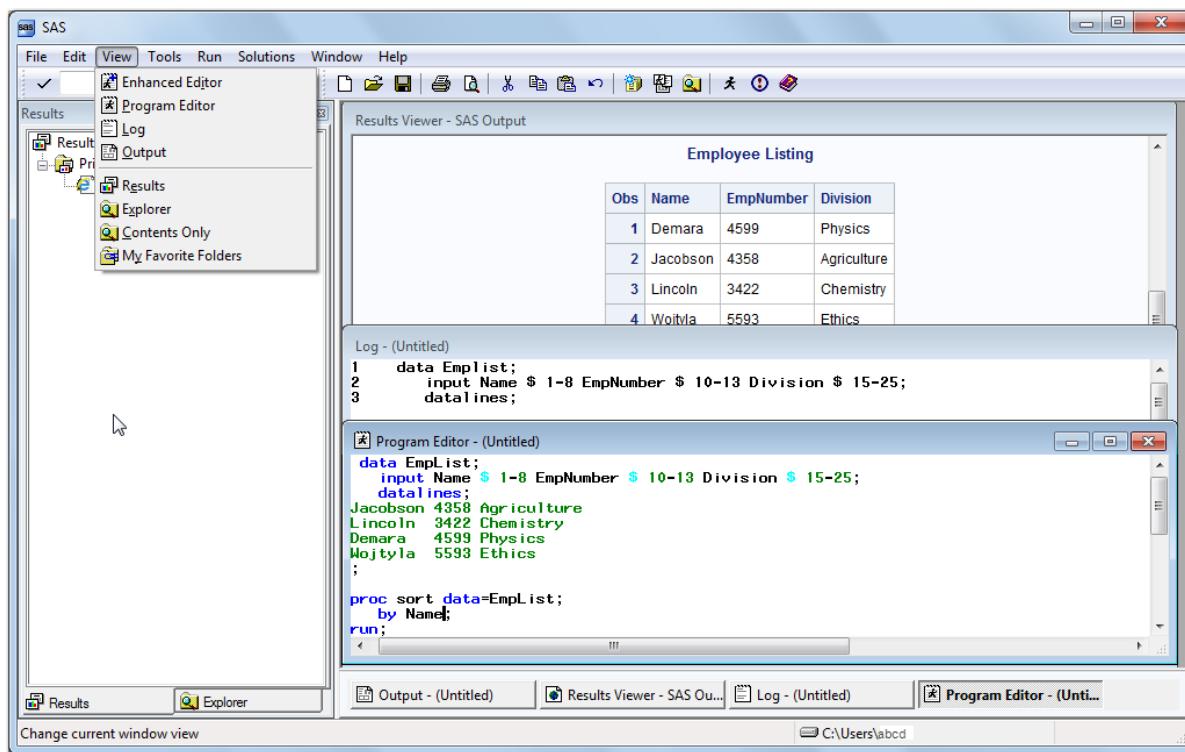
Menu choices change as you change the windows that you are using. For example, if you select **Explorer** from the **View** menu, and then select **View** again, the menu lists the **View** options that are available when the Explorer window is active.

The following display shows the **View** menu when the Explorer window is active:

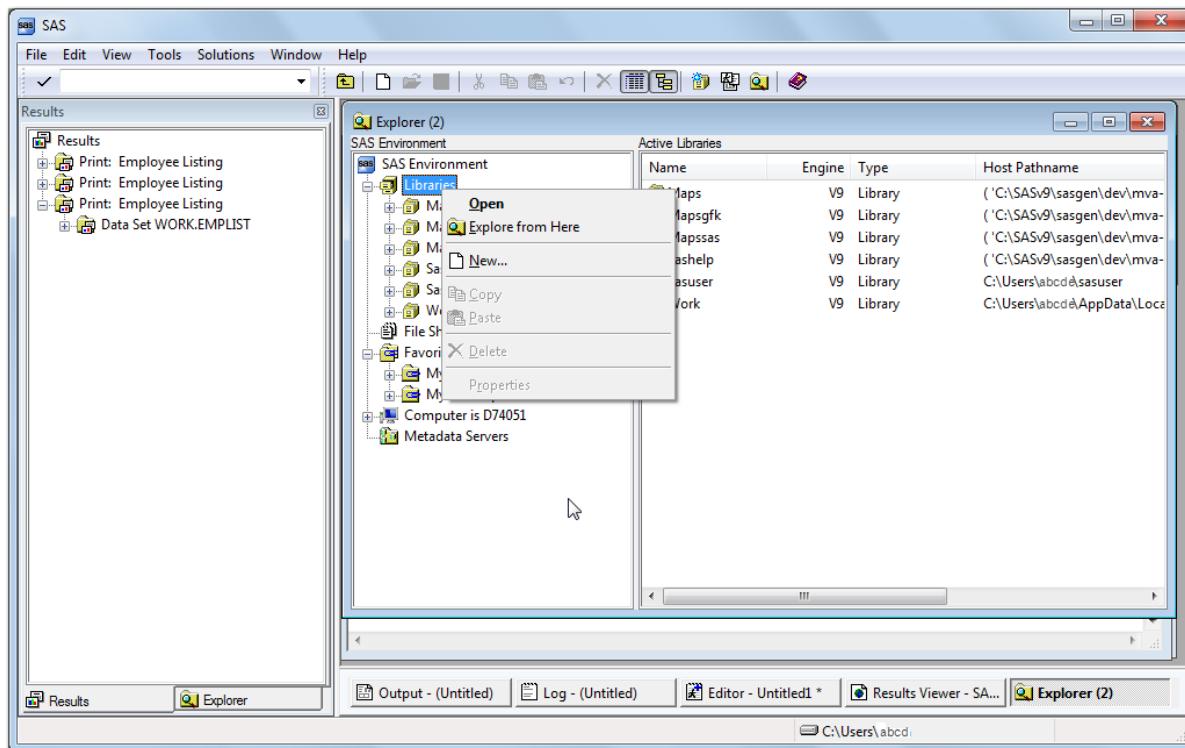
**Figure 16.9** View Options When the Explorer Window Is Active

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 16.10** 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**  $\Rightarrow$  **Explorer** and then right-click **Libraries** in the Explorer window, the following menu appears:

**Figure 16.11** 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 16.12 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

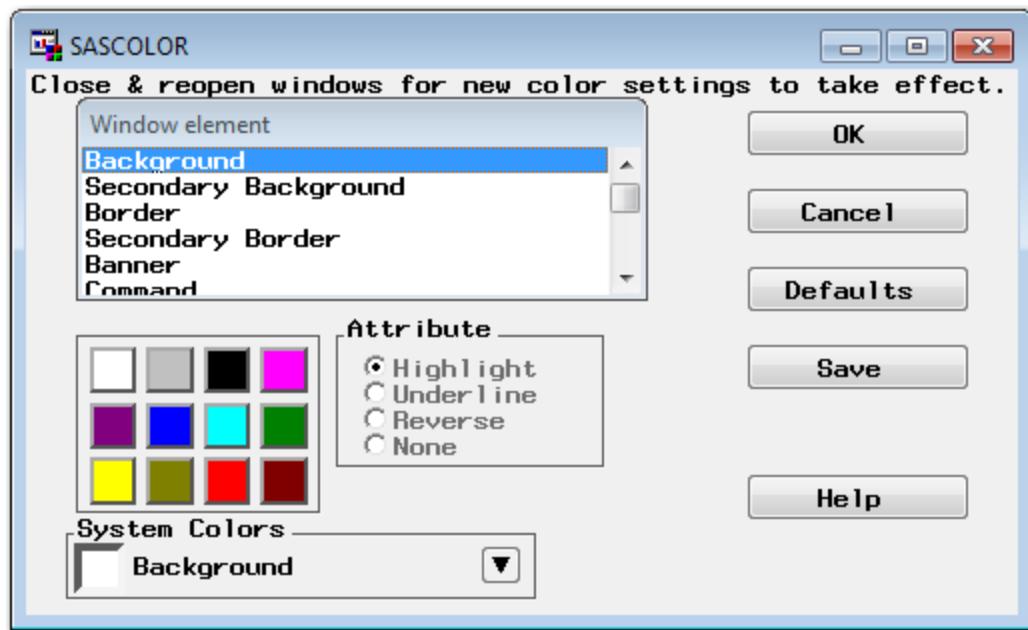
The command line is located to the left of the toolbar. In the command line, you can enter commands, such as those that open SAS windows and those that retrieve help information.

The following is an example of a command that opens the SASCOLOR window:

*Figure 16.13 Example of the Command Line*



Figure 16.14 The SASCOLOR Window

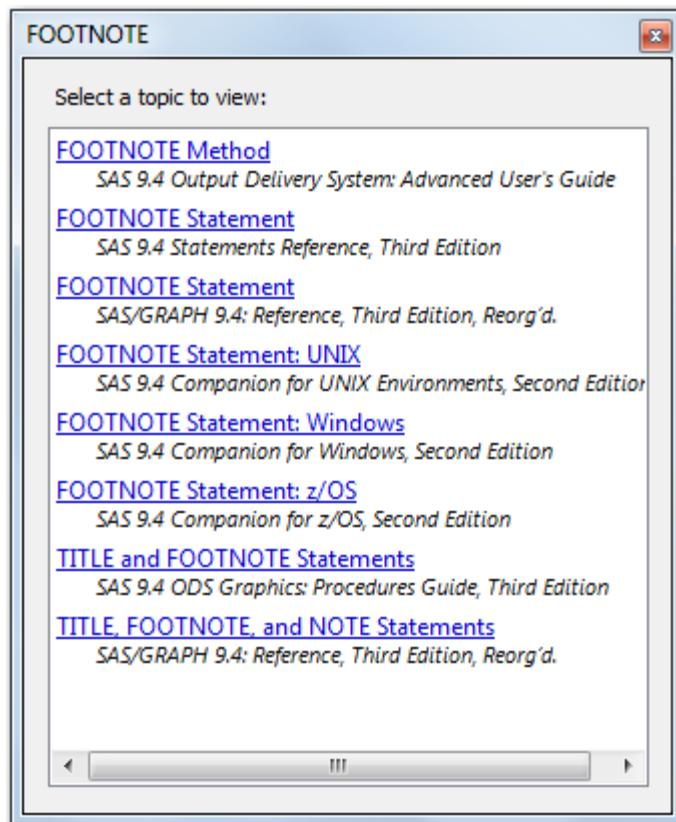


## Getting Help in SAS

### 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 16.15 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.

The following table lists all portable SAS windows, window descriptions, and the commands that open the windows.

*Table 16.1 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.	DMEXPORTS

Window Name	Description	Window Commands
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
FSFORM	Enables you to customize a form for sending output to the printer.	FSFORM <i>formname</i>
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)

Window Name	Description	Window Commands
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, reorder, and test metadata server connections.	METACON
New Library	Enables you to create a new SAS library and assign a libref.	DMLIBASSIGN
NOTE PAD	Enables you to create and store notepads of text.	NOTE PAD, NOTE, FILEPAD <i>filename</i>
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 <i>libref.SAS-data-set</i> , V6VAR <i>libref.SAS-data-set</i>

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.

# Managing Your Data in the SAS Windowing Environment

<i>Introduction to Managing Your Data in the SAS Windowing Environment</i> .....	<b>406</b>
<i>Managing Data with SAS Explorer</i> .....	<b>406</b>
Introduction to Managing Data with SAS Explorer .....	406
Viewing Libraries and Data Sets .....	407
Assign File Shortcuts .....	408
Rename a SAS Data Set .....	409
Copy or Duplicate a SAS Data Set .....	409
Sorting Data Sets in a Library .....	410
View the Properties of a SAS Data Set .....	410
<i>Working with VIEWTABLE</i> .....	<b>411</b>
Overview of VIEWTABLE .....	411
Opening a SAS Data Set in a VIEWTABLE Window .....	411
Displaying Table Headers as Names or Labels .....	413
Customizing SAS Explorer for Opening the VIEWTABLE Window .....	414
Order of Precedence for How Column Headings Are Displayed .....	415
Mapping the VIEWTABLE Command to a Function Key .....	416
Temporarily Change Column Headings .....	416
Move Columns in a Table .....	417
Sort by Values of a Column .....	418
Edit Cell Values .....	420
<i>Subsetting Data By Using the WHERE Expression</i> .....	<b>421</b>
Subset Rows of a Table .....	421
Clear the WHERE Expression .....	424
<i>Exporting a Subset of Data</i> .....	<b>425</b>
Overview of Exporting Data .....	425
Export Data .....	425
<i>Importing Data into a Table</i> .....	<b>428</b>
Overview of Importing Data .....	428
Import a Standard File .....	428
Import a Nonstandard File .....	430

---

# 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.

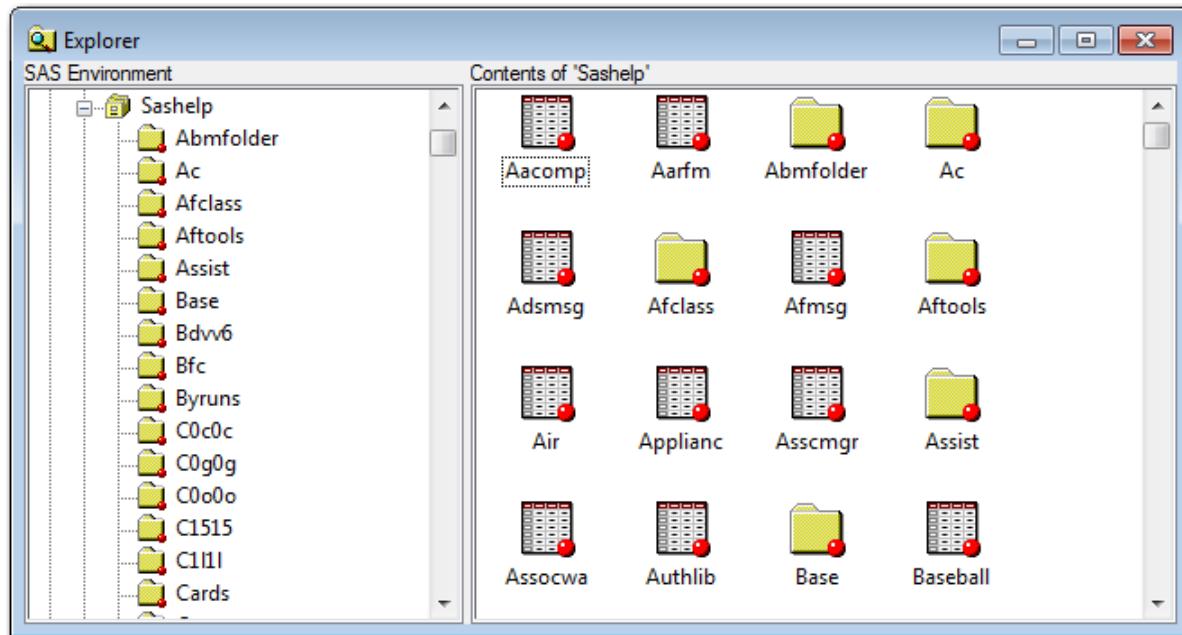
## Viewing Libraries and Data Sets

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:

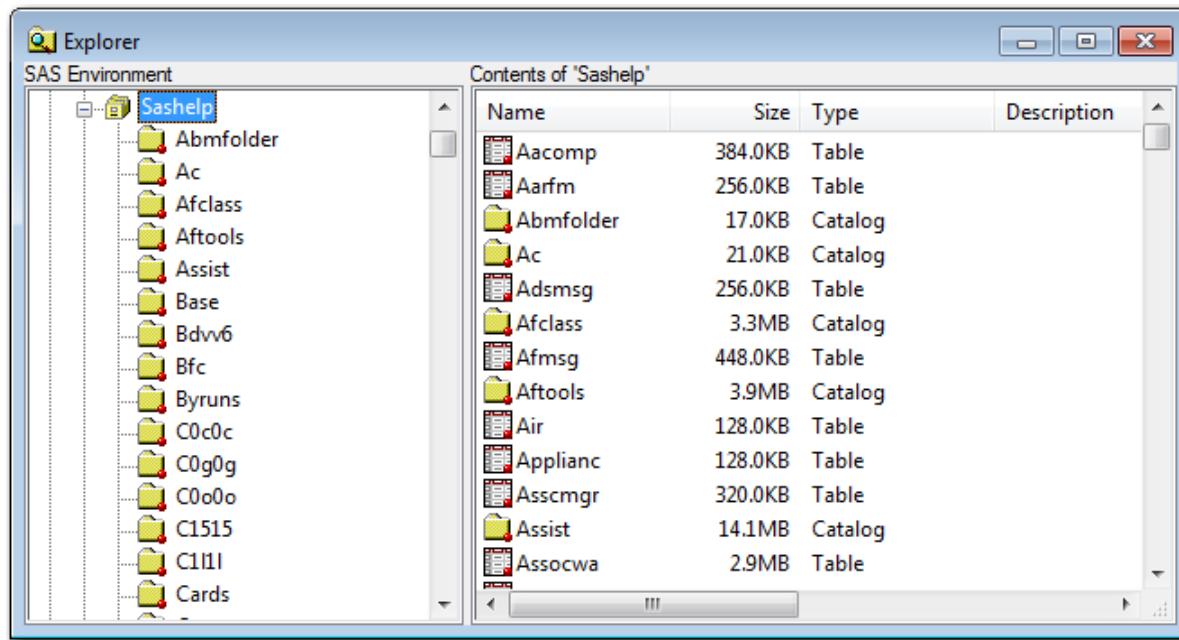
- To view large icons, select **Large Icons** from the **View** menu.
- To view small icons, select **Small Icons** from the **View** menu.
- To view data sets in a list, select **List** from the **View** menu.

The following example uses large icons to show the contents of Sashelp:

Figure 17.1 Sashelp Library Represented by Large Icons



If you select the Sashelp library and then select **View**  $\Rightarrow$  **Details** from the menu bar, the contents of the Sashelp library is displayed, along with the size and type of the data sets:

**Figure 17.2** 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.

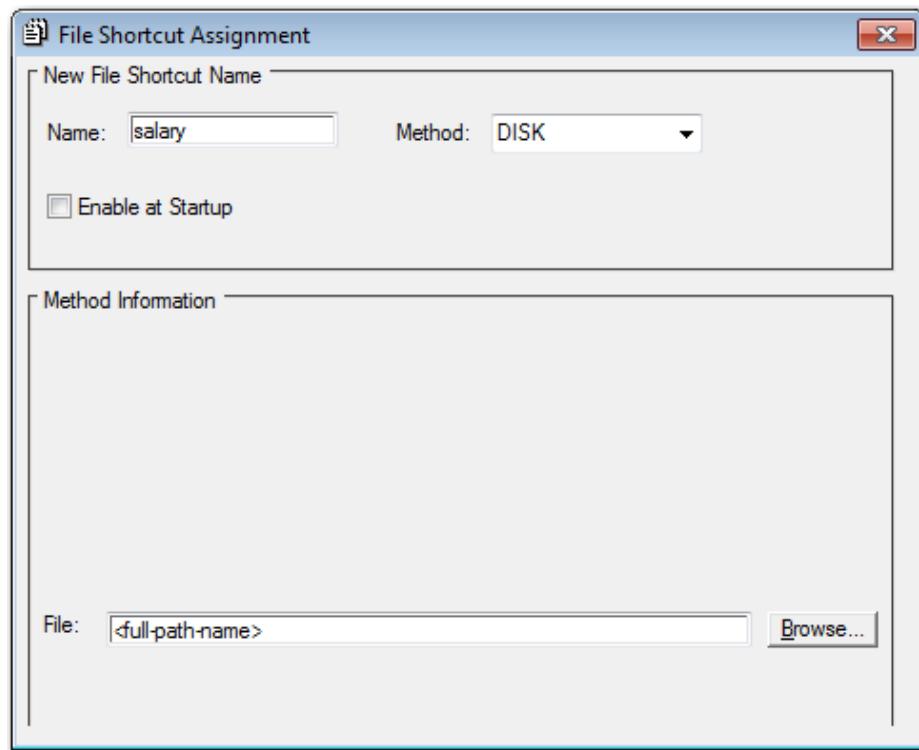
## Assign File Shortcuts

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**  $\Rightarrow$  **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**.

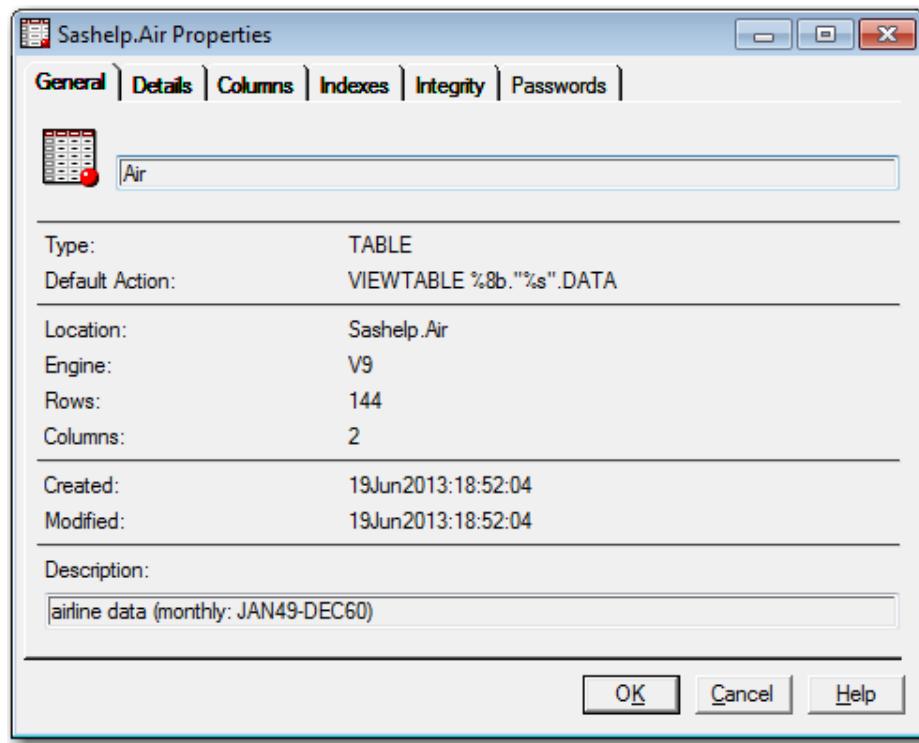
## Sorting Data Sets in a Library

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.

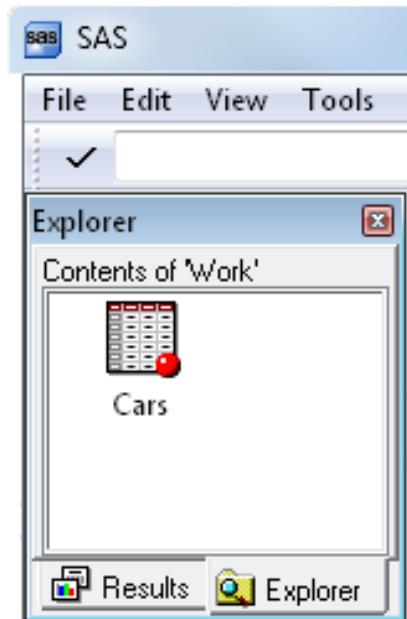
### Opening a SAS Data Set in a VIEWTABLE Window

#### Using the SAS Explorer Window

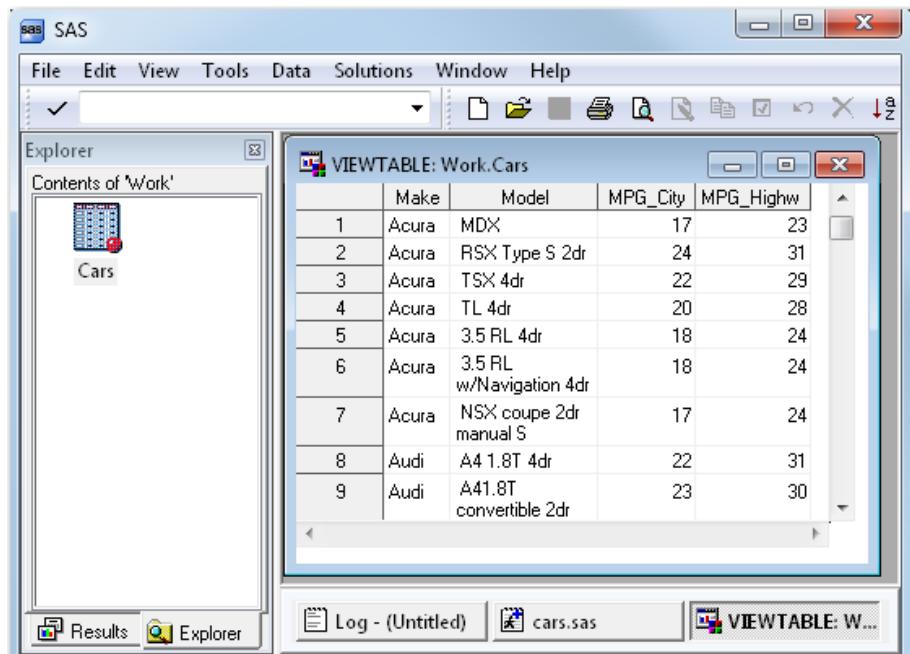
You can open an existing SAS data set in a VIEWTABLE window by double-clicking on the SAS data set icon in the SAS Explorer window, or by specifying the [VIEWTABLE Command](#) in the SAS Display Manager command line.

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

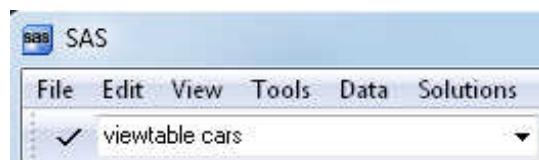
You can also open a data set in a VIEWTABLE window by using the VIEWTABLE command in the SAS Display Manager command line.

- 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:

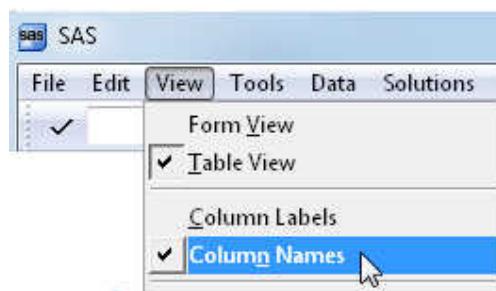
```
viewtable cars
```



## 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**  $\Rightarrow$  **Column Names** or **View**  $\Rightarrow$  **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.

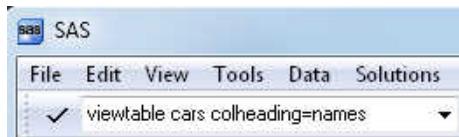
- Using the VIEWTABLE command to change the way table headers are displayed when a table is opened:

- 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:

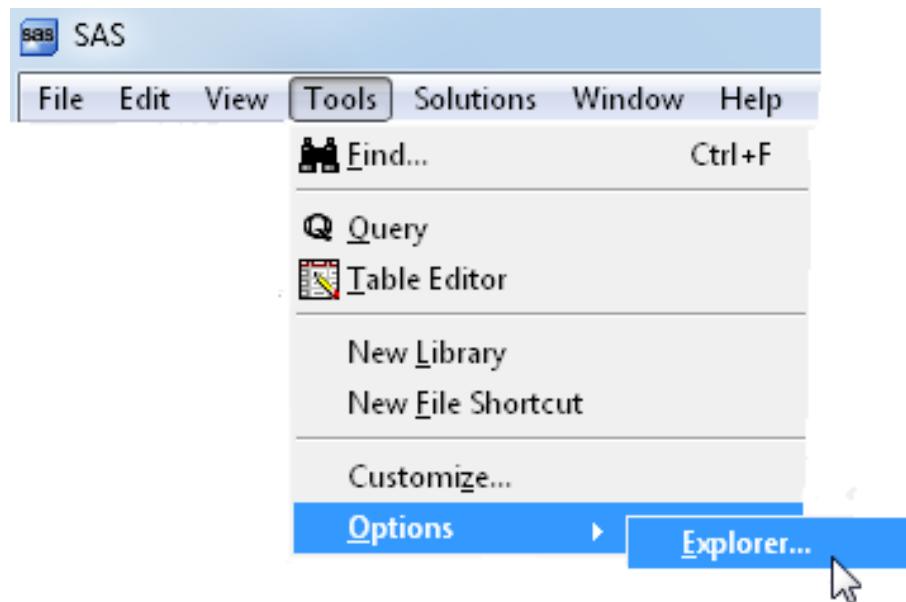
```
viewtable cars colheading=names
```



## 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.

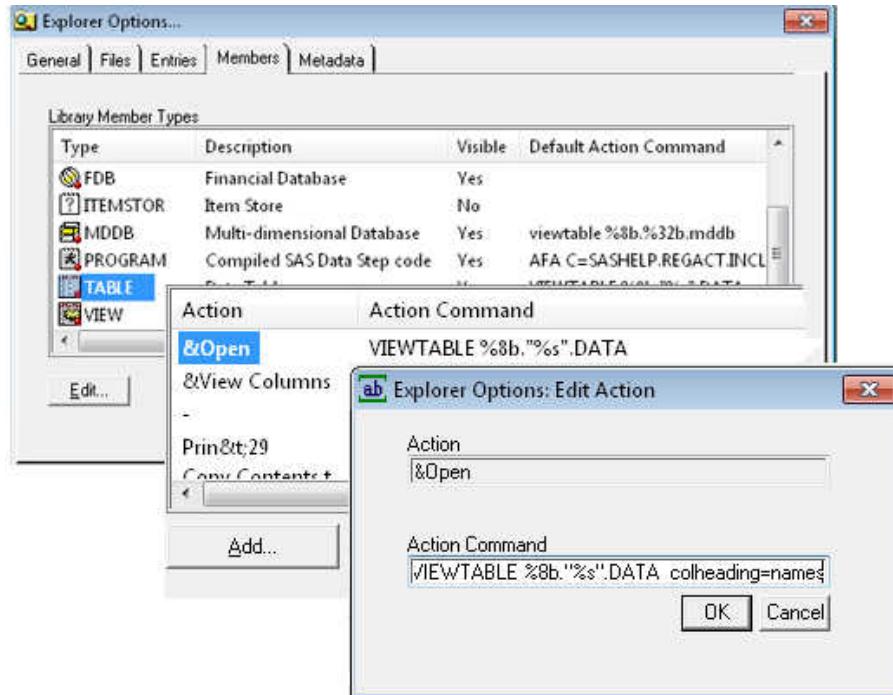
- 1 With the SAS Explorer window active, select **Tools**  $\Rightarrow$  **Options**  $\Rightarrow$  **Explorer** to open the Explorer Options window.



- 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**  $\Rightarrow$  **Column Names** or **View**  $\Rightarrow$  **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**.

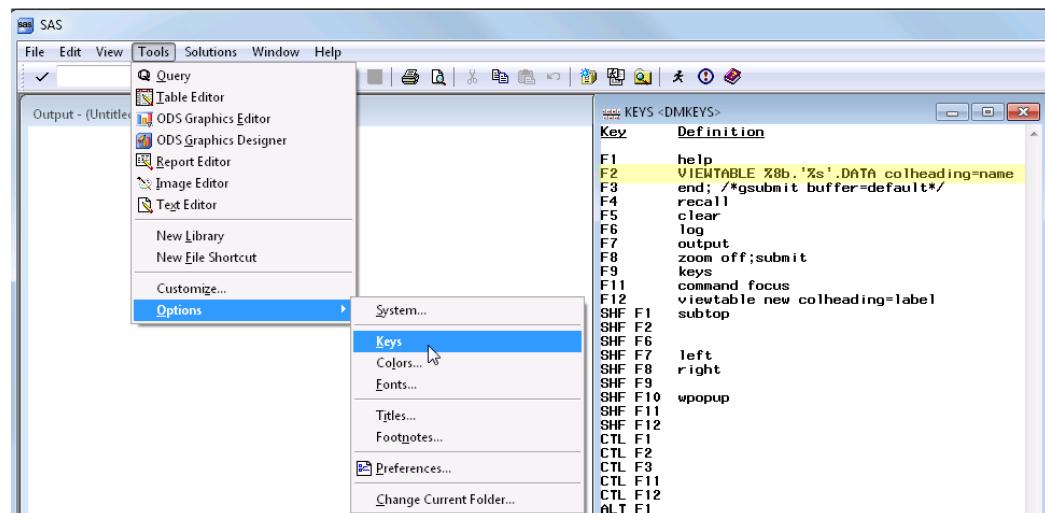
## Mapping the VIEWTABLE Command to a Function Key

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**  $\Rightarrow$  **Options**  $\Rightarrow$  **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.



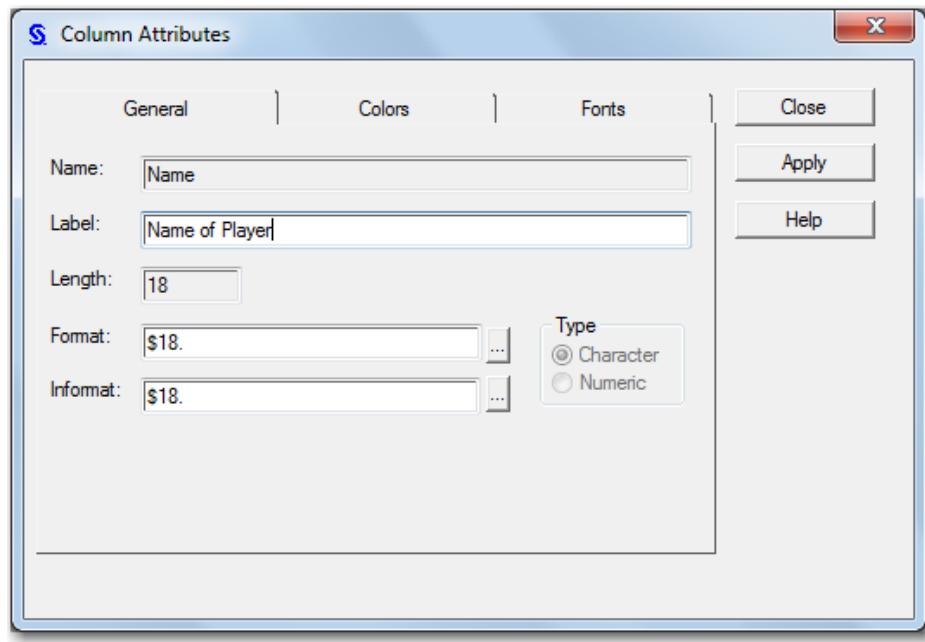
For more information about using VIEWTABLE, see [Doing More with the SAS® Display Manager: From Editor to ViewTable - Options and Tools You Should Know \(PDF\)](#).

## 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	Galaraga, 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	51	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
17	Cincinnati	Bell, Bob	560	150	20	99	75	77

- Click **Close** to close the Column Attributes window.

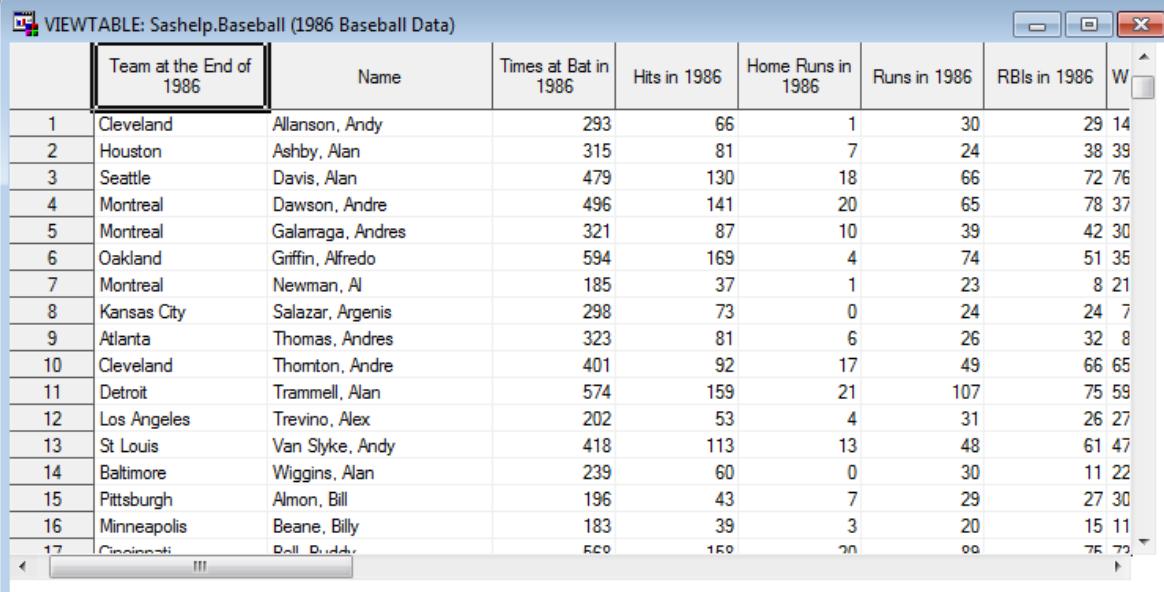
## Move Columns in a Table

Within the VIEWTABLE window, you can rearrange columns in your table. To move columns in your table, follow these steps:

- 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.



	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	Galaraga, 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	Thomton, 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
17	Cincinnati	Bull, Dickey	560	150	20	80	75	72

## 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:

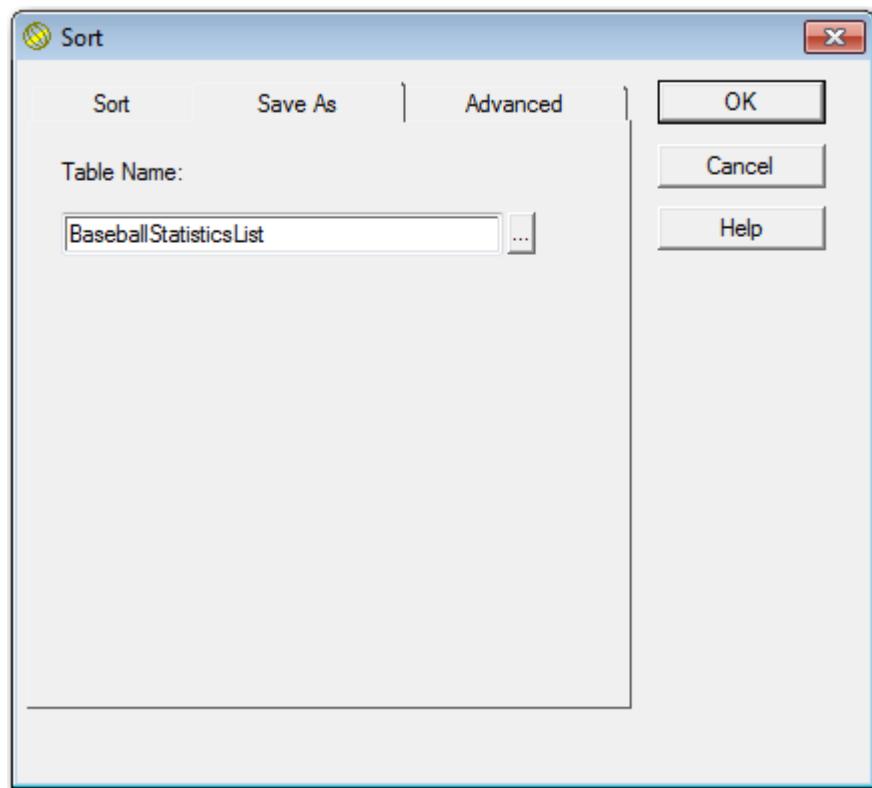
- 1 Right-click the heading of the column on which you want to sort, and select **Sort** from the menu.
- 2 Select **Ascending** or **Descending** from the menu.
- 3 When the following warning message appears, click **Yes** to create a sorted copy of the table.



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**.

The screenshot shows a Windows application window titled "VIEWTABLE: BaseballStatisticsList (1986 Baseball Data)". The window contains a table with 17 rows of data. The columns are labeled: 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, and Years Major. The data includes players from Atlanta and Baltimore, such as Thomas, Andres; Homer, Bob; Sample, Billy; Murphy, Dale; Hubbard, Glenn; Oberkfell, Ken; Moreno, Omar; Virgil, Ozzie; Ramirez, Rafael; Harper, Terry; Simmons, Ted; Wiggins, Alan; Ripken, Cal; Murray, Eddie; Lynn, Fred; Rayford, Floyd; and Rodriguez, Luis.

	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	Baltimore	Rodriguez, Luis	242	102	6	40	26	40	15

## 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**  $\Rightarrow$  **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.

The screenshot shows the same "VIEWTABLE: BaseballStatisticsList (1986 Baseball Data)" window, but now in edit mode. The cell containing the value "408" in the "Times at Bat in 1986" column for the fifth row is highlighted with a blue selection bar. The rest of the table and its structure are identical to the previous screenshot.

	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	Baltimore	Rodriguez, Luis	242	102	6	40	26	40	15

- 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**.

	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	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
17	Baltimore	Reagan, Lynn	242	102	6	40	26	40	15

- 4 Select **File**  $\Rightarrow$  **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**  $\Rightarrow$  **Close**, you are prompted to save the pending changes to the second row.

## Subsetting Data By Using the WHERE Expression

### Subset Rows of a Table

In the VIEWTABLE window, you can subset the display to show only those rows that meet one or more conditions. To subset rows of a table, follow these steps:

- 1 In the Explorer window, open a library and double-click the table that you want to subset.

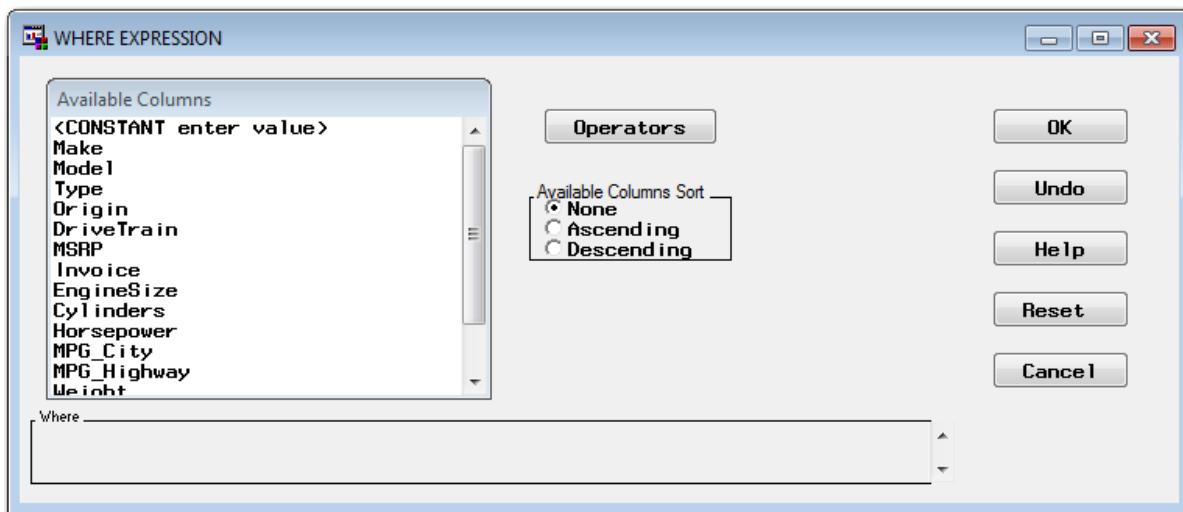
In this example, the Cars data table is selected.

**VIEWTABLE: Sashelp.Cars (2004 Car Data)**

	Make	Model	Type	Origin	Drive Train	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	++

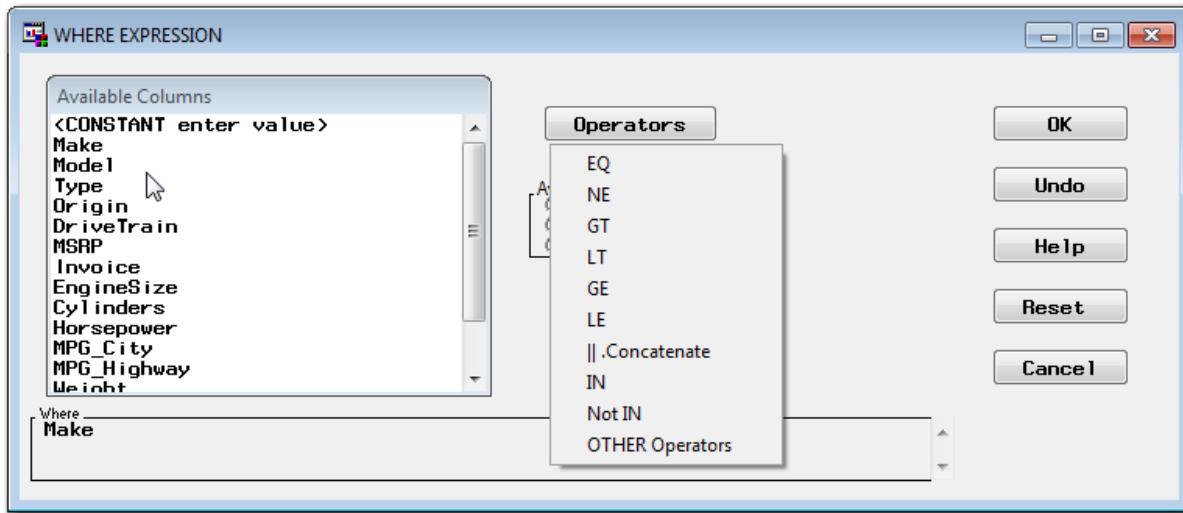
2 Right-click any table cell that is not a heading and select **Where** from the menu.

The WHERE EXPRESSION window appears.



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.

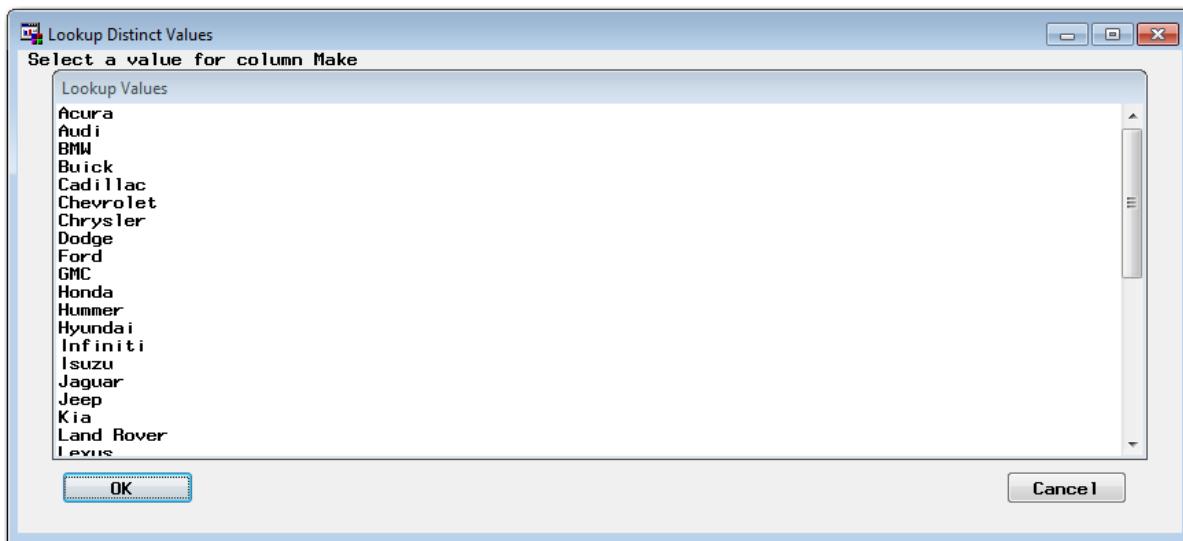


- 4 In the **Available Columns** list, select another value to complete the WHERE expression.

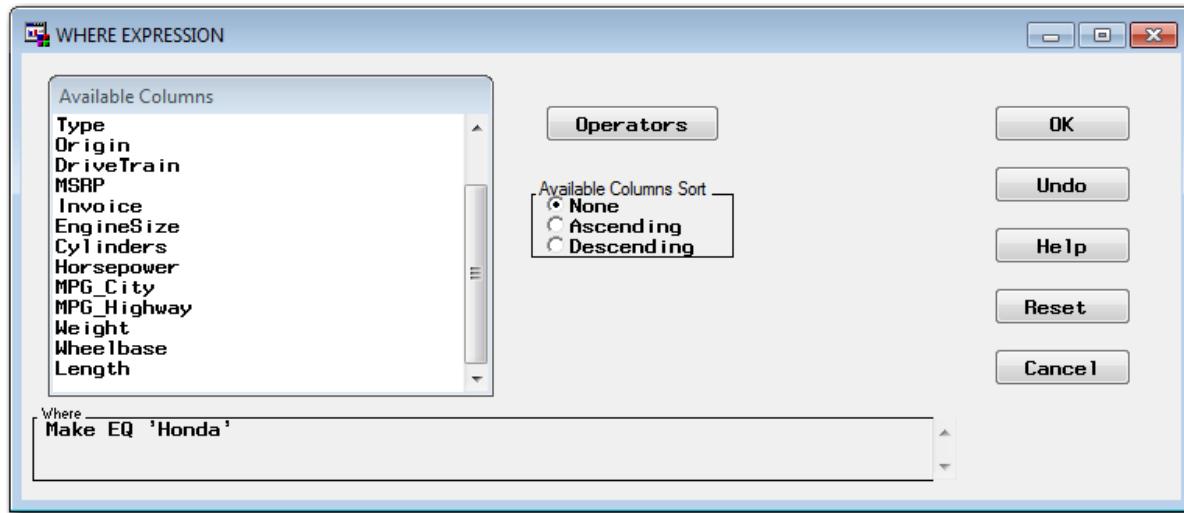
In this example, scroll to the bottom of the **Available Columns** window and select **<LOOKUP distinct values>**.

- 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.

	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

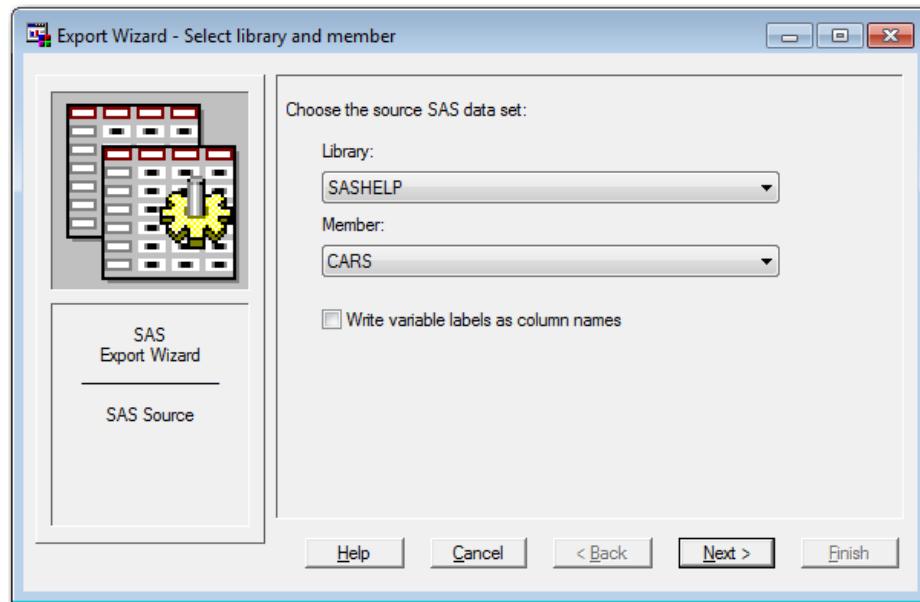
To export data, follow these steps:

- 1 With the Explorer window active, select **File**  $\Rightarrow$  **Export Data**.

The Export Wizard - Select library and member window appears.

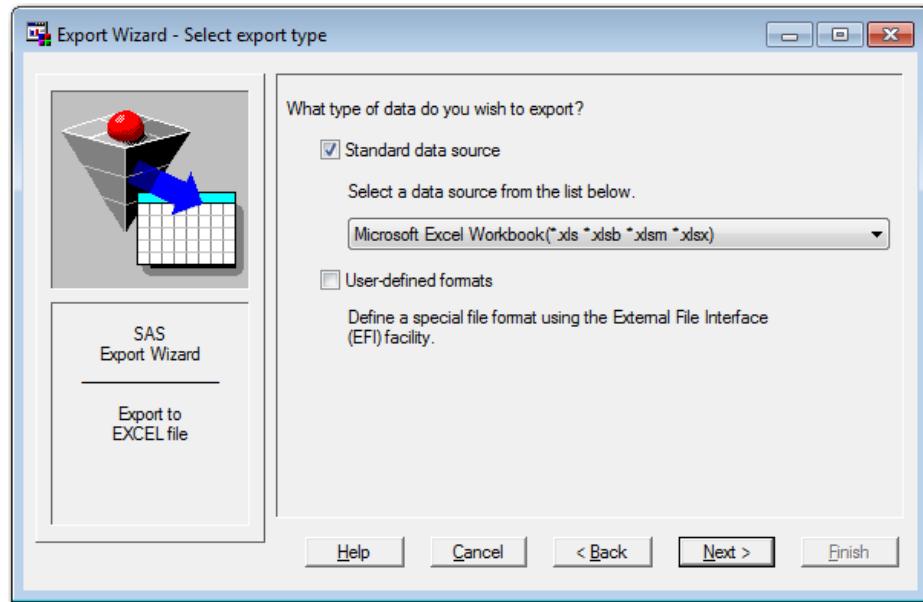
- 2 Select the SAS data set from which you want to export data.

In this example, **Sashelp** is selected as the library, and **Cars** is the member name.



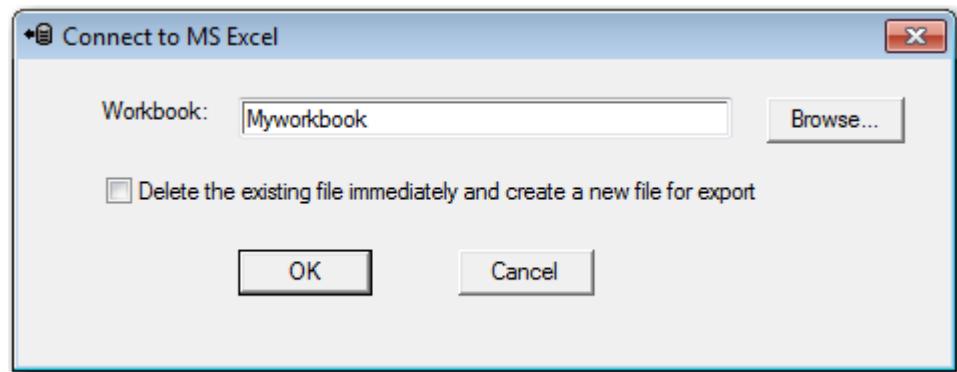
- 3 Click **Next** and the Export Wizard - Select export type window appears.
- 4 Select the type of data source to which you want to export files.

In this example, **Microsoft Excel Workbook** is selected. Note that **Standard data source** is selected by default.



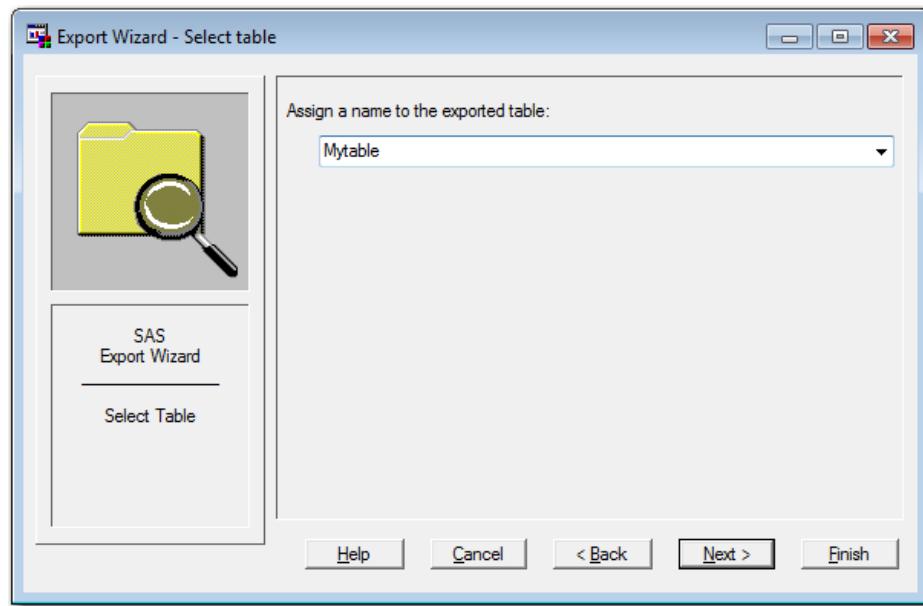
- 5 Click **Next** to display the Connect to MS Excel window.
- 6 In the **Workbook** field, enter the name of the workbook that will contain the exported file and then click **OK**.

In this example, **Myworkbook** is entered as the name of the workbook.



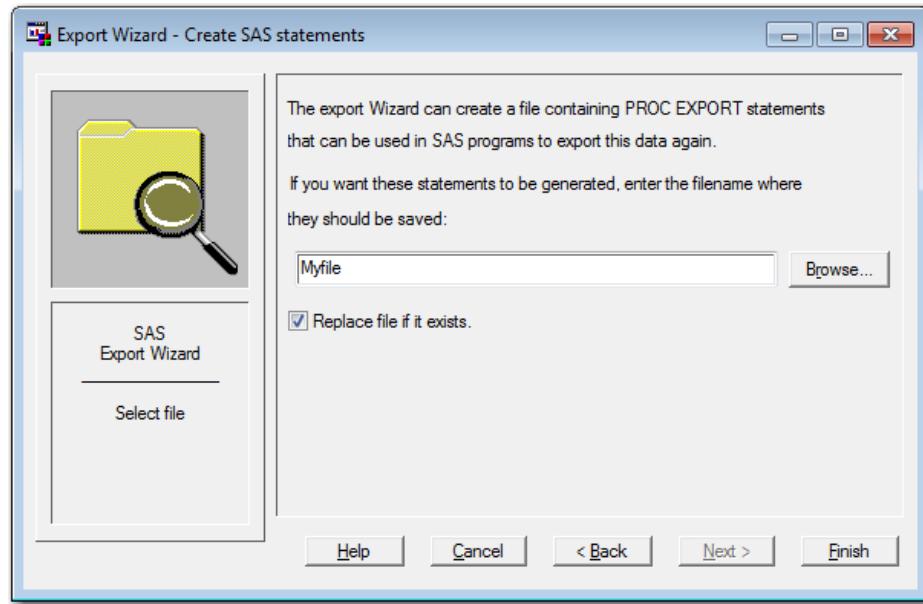
- 7 When the Export Wizard - Select table window appears, enter a name for the table that you are exporting.

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

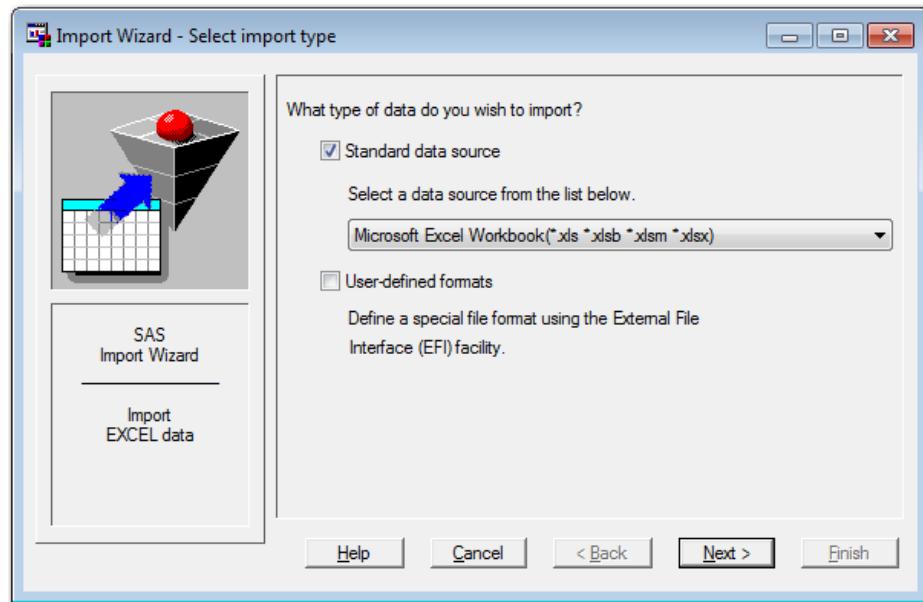
Whether your data is stored in a standard file format or in your own special file format, you can use the Import Wizard to import data into a SAS table. The types of files that you can import depend on your operating environment.

## Import a Standard File

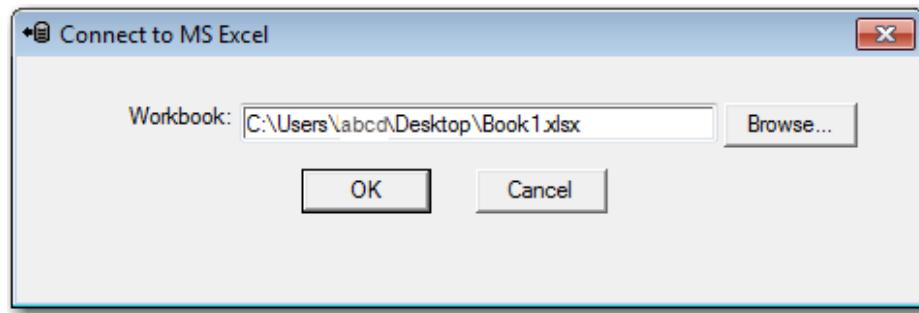
To import a standard file, follow these steps:

- 1 With the Explorer window active, select **File**  $\Rightarrow$  **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.

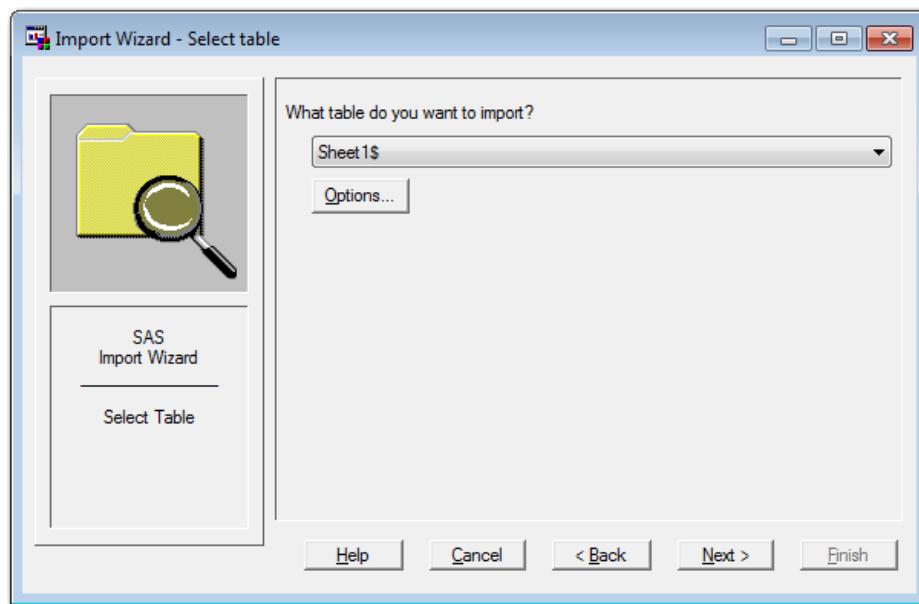
Note that **Standard data source** is selected by default. In this example, **Microsoft Excel Workbook** is selected.



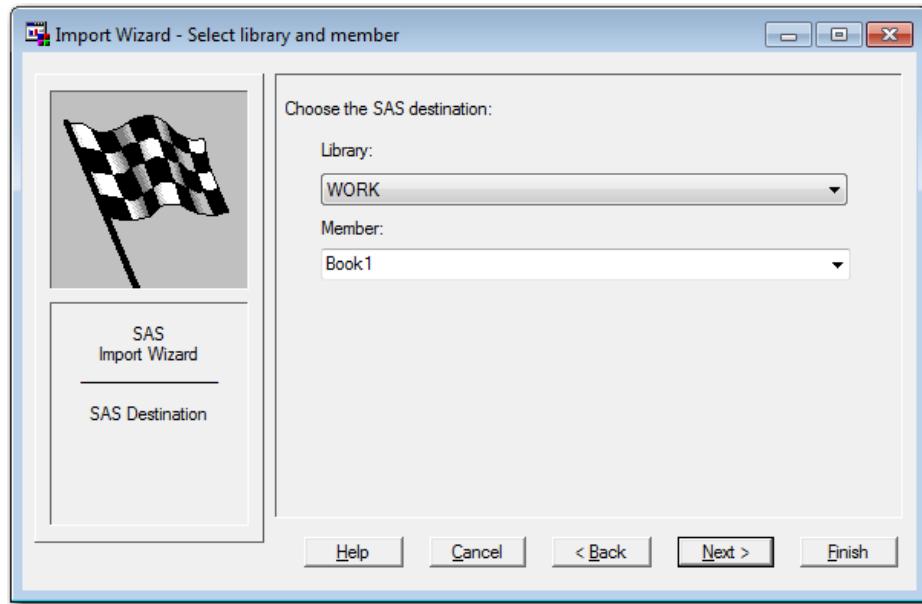
- 3 Click **Next** to continue.
- 4 In the Connect to MS Excel window, enter the pathname of the file that you want to export, and then click **OK**.



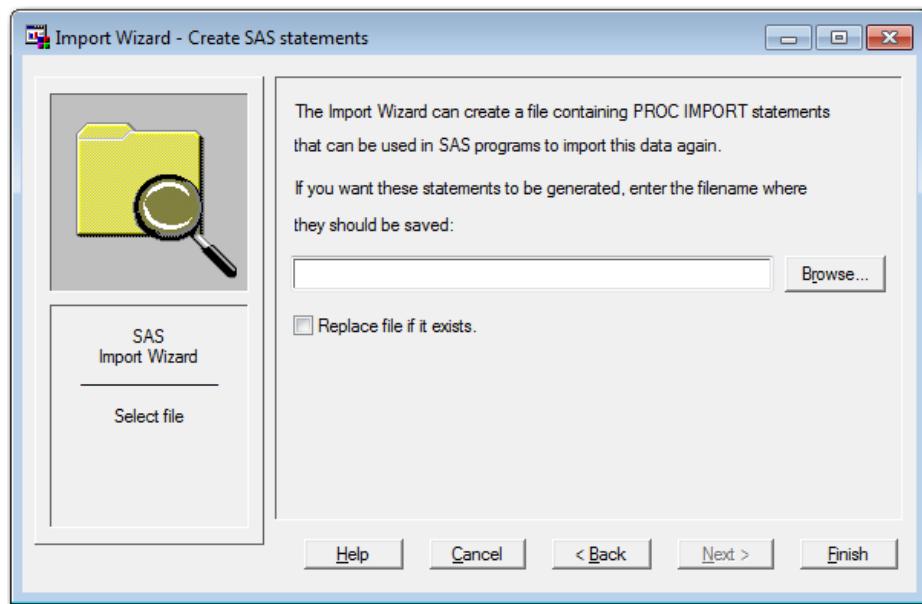
- 5 In the Import Wizard - Select table window, enter the name of the table that you want to import.



- 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.

**PART 3**

# SAS Cloud Analytic Services

<i>Chapter 18</i>	
<i>Introduction to SAS Cloud Analytic Services</i>	433
<i>Chapter 19</i>	
<i>SAS Language Support for CAS</i>	435



# 18

## Introduction to SAS Cloud Analytic Services

<i>What is SAS Cloud Analytic Services?</i> .....	433
<i>What Does This Mean for the SAS 9 Programmer?</i> .....	433

### 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.
- For more information about using the SAS language to access SAS Cloud Analytic Services, see, [Chapter 19, “SAS Language Support for CAS,” on page 435](#).



# SAS Language Support for CAS

---

<i>SAS Language Elements for CAS</i> .....	<b>435</b>
DATA Step Processing .....	435
DATA Step Language Elements for CAS .....	435
SAS Procedures for CAS .....	437
<i>CAS-specific Language Elements</i> .....	<b>437</b>

---

## 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 19.1** Example Documentation Syntax Page That Shows a “Restriction” to Indicate That the Language Element Is Not Supported in CAS

ADDR Function	ADDR Function
ADDRLONG Function	Returns the memory address of a variable on a 32-bit platform.
AIRY Function	Category: Special
ALLCOMB Function	Restrictions: Use on 32-bit platforms only. This function is not valid on the CAS server.

SAS language elements that are supported in CAS display “CAS” in the Categories field of the language elements’ syntax page:

**Figure 19.2** Example Documentation Syntax Page That Shows Support for CAS in the Categories Field

BY Statement	BY Statement
CALL Statement	Controls the operation of a SET, MERGE, MODIFY, or UPDATE statement
CARDS Statement	Valid in: DATA step or PROC step Categories: CAS File-Handling
CARDS4 Statement	Type: Derived

Each language element dictionary also contains a summary table of CAS-supported language elements:

**Figure 19.3** Example Documentation Category Page Showing CAS-supported Language Elements

SAS Data Set Options by Category	SAS Data Set Options by Category
ALTER= Data Set Option	The categories for SAS data set options correspond to the SAS data set option groups that run in the CAS server
BUFNO= Data Set Option	options that are associated with data sets
BUFSIZE= Data Set Option	options that are associated with observations
COMPRESS= Data Set Option	options that are associated with indexes
CNTLLEV= Data Set Option	options that are associated with variables
DLDMGACTION= Data Set Option	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.



**PART 4****DATA Step Concepts**

<i>Chapter 20</i>	<b><i>DATA Step Processing</i></b>	<b>441</b>
<i>Chapter 21</i>	<b><i>Reading Raw Data</i></b>	<b>471</b>
<i>Chapter 22</i>	<b><i>BY-Group Processing in the DATA Step</i></b>	<b>491</b>
<i>Chapter 23</i>	<b><i>Reading, Combining, and Modifying SAS Data Sets</i></b>	<b>509</b>
<i>Chapter 24</i>	<b><i>Using DATA Step Component Objects</i></b>	<b>565</b>
<i>Chapter 25</i>	<b><i>Array Processing</i></b>	<b>603</b>



# 20

# DATA Step Processing

<i>Why Use a DATA Step?</i>	441
<i>Overview of DATA Step Processing</i>	442
Flow of Action	442
The Compilation Phase	444
The Execution Phase	444
<i>Processing a DATA Step: A Walk-through</i>	445
Sample DATA Step	445
Creating the Input Buffer and the Program Data Vector	446
Reading a Record	446
Writing an Observation to the SAS Data Set	447
Reading the Next Record	448
When the DATA Step Finishes Executing	449
<i>About DATA Step Execution</i>	450
The Default Sequence of Execution in the DATA Step	450
Changing the Default Sequence of Execution	451
Step Boundary—How to Know When Statements Take Effect	453
What Causes a DATA Step to Stop Executing	455
<i>About Creating a SAS Data Set with a DATA Step</i>	456
Creating a SAS Data File or a SAS View	456
Sources of Input Data	456
Reading Raw Data: Examples	457
Reading Data from SAS Data Sets	459
Generating Data from Programming Statements	460
<i>Writing a Report with a DATA Step</i>	460
Example 1: Creating a Report without Creating a Data Set	460
Example 2: Creating a Customized Report	461
Example 3: Creating an HTML Report Using ODS and the DATA Step	465
<i>The DATA Step and ODS</i>	467
<i>DATA Step Processing Time</i>	468

---

## Why Use a DATA Step?

Using the DATA step is the primary method for creating a SAS data set with Base SAS software. A DATA step is a group of SAS language statements that begin with a DATA statement. The group of language statements contains other programming

statements that manipulate existing SAS data sets or create SAS data sets from raw data files.

You can use the DATA step for the following tasks:

- creating SAS data sets (SAS data files or SAS views)
- creating SAS data sets from input files that contain raw data (external files)
- creating new SAS data sets from existing ones by subsetting, merging, modifying, and updating existing SAS data sets
- analyzing, manipulating, or presenting your data
- computing the values for new variables
- report writing, or writing files to disk or tape
- retrieving information
- file management

**Note:** A DATA step creates a SAS data set. This data set can be a SAS data file or a SAS view. A SAS data file stores data values while a SAS view stores instructions for retrieving and processing data. When you can use a SAS view as a SAS data file, as is true in most cases, this documentation uses the broader term SAS data set.

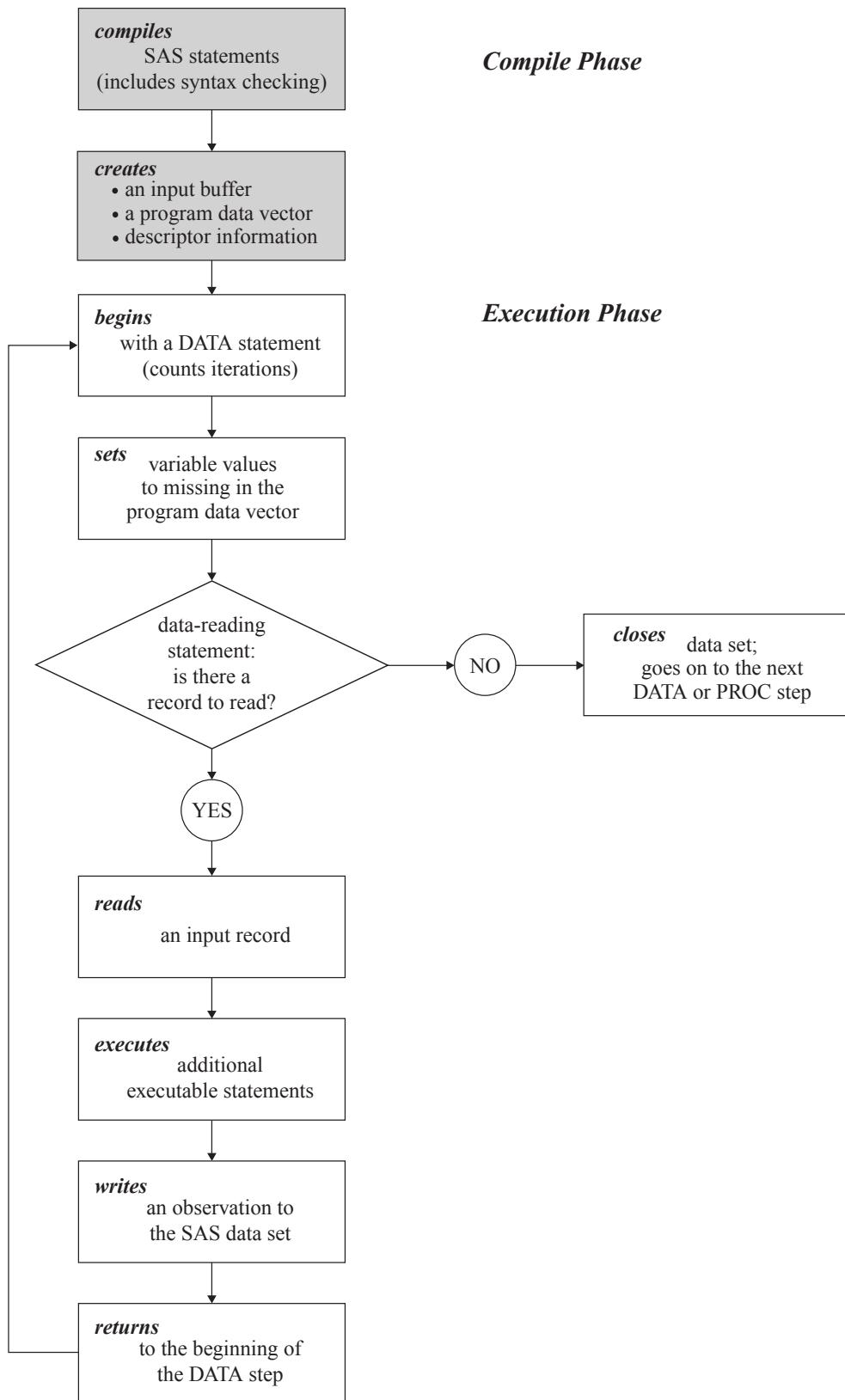
---

## Overview of DATA Step Processing

---

### 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 20.1** 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, 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. Note that 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 Data Vector

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 20.2** Input Buffer and Program Data Vector

Input Buffer											
1	2	3	4	5	6	7	8	9	0	1	2
1	2	3	4	5	6	7	8	9	0	1	2

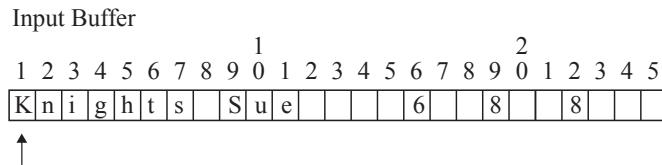
Program Data Vector								
TeamName	ParticipantName	Event1	Event2	Event3	TeamTotal	_N_	_ERROR_	
		.	.	.	0	1	0	
Drop						Drop	Drop	

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.

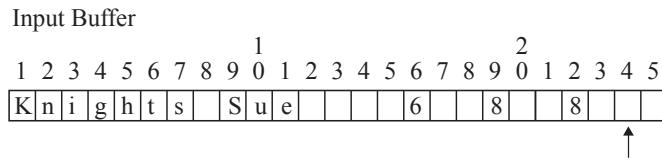
The variable TeamName is marked Drop in the PDV because of the `DROP=` data set option in the DATA statement. Dropped variables are not written to the SAS data set. The `_N_` and `_ERROR_` variables are dropped because automatic variables created by the DATA step are not written to a SAS data set. See [Chapter 4, “SAS Variables,” on page 37](#) 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 20.3** 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 20.4** 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 20.5** 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 20.6** 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.

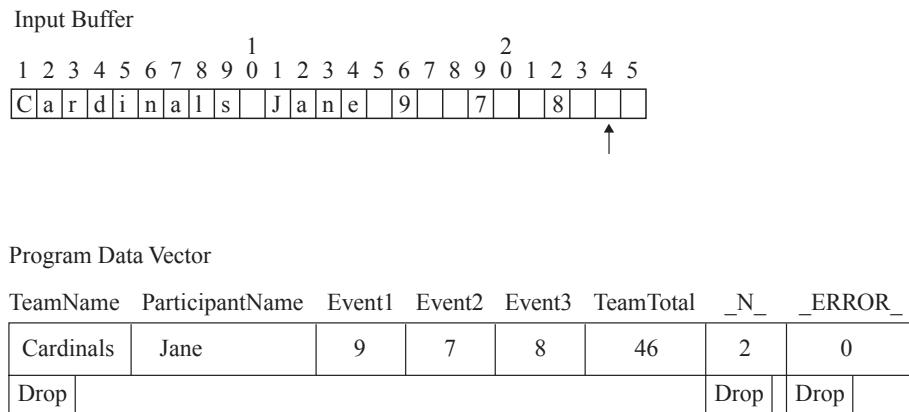
**Figure 20.7** Current Values in the Program Data Vector

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 20.8** Input Buffer, Program Data Vector, and First Two Observations

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 20.1** 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	

## About DATA Step Execution

### The Default Sequence of Execution in the DATA Step

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 20.1* Default Execution for Statements in a DATA Step

Structure of a DATA Step	Action
DATA statement	begins the step counts iterations
Data-reading statements: *	
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
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

Structure of a DATA Step	Action
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	writes an observation to a SAS data set returns to the DATA statement
At top of DATA step: Automatic reset	resets values to missing in program data vector

\* 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 Default Sequence of Execution

### Using Statements to Change the Default Sequence of Execution

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 20.2** 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](#).

## Using Functions to Change the Default Sequence of Execution

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 20.3** 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.

SAS Language Element	Function
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.
_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 Boundary—How to Know When Statements Take Effect

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 altests;
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 a DATA Step to Stop Executing

DATA steps stop executing under different circumstances, depending on the type and number of sources of input.

*Table 20.4 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*.

## About Creating a SAS Data Set with a DATA Step

### Creating a SAS Data File or a SAS View

You can create either a SAS data file, a data set that holds actual data, or a SAS view, a data set that references data that is stored elsewhere. By default, you create a SAS data file. 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 without having to edit your DATA step. For example, you can process 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 creates a SAS view called Monthly\_Sales.

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

The following DATA statement creates a data file 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, the clipboard, a data URL, an email, an FTP protocol, a Hadoop Distributed File System, TCP/IP socket, a URL, a 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

### Example 1: Reading External File Data

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 a SAS data set called 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 variable WeightLoss.
- 5 Execute the DATA step.
- 6 Print data set Weight using the PRINT procedure.
- 7 Execute the PRINT procedure.

### Example 2: Reading Instream Data Lines

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

- 7 Execute the PRINT procedure.

### Example 3: Reading Instream Data Lines with Missing Values

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;
    infile datalines missover;  1
        input IDnumber $ Week1 Week16;
        WeightLoss2=Week1-Week16;
        datalines;  2
2477 195 163
2431
2456 173 155
2412 135 116
;  3

proc print data=weight2;  4
run;  5

```

- 1 Use the MISSOVER option to assign missing values to variables that do not contain values in records that do not satisfy the current INPUT statement.
- 2 Begin data lines.
- 3 Signal end of data lines and execute the DATA step.
- 4 Print data set Weight2 using the PRINT procedure.
- 5 Execute the PRINT procedure.

### Example 4: Using Multiple Input Files in Instream Data

This example shows how to use multiple input files as instream data to your program. This example reads the records in each file and creates the All\_Errors SAS data set. The program then sorts the observations by Station, and creates a sorted data set called Sorted\_Errors. The print procedure prints the results.

```

data all_errors;
length filelocation $ 60;
input filelocation; /* reads instream data */
infile daily filevar=filelocation
filename=daily end=done;
do while (not done);
    input Station $ Shift $ Employee $ NumberOfFlaws;
    output;
end;
put 'Finished reading ' daily=;
datalines;
pathmyfile_A

```

```

pathmyfile_B
pathmyfile_C
;

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

proc print data = sorted_errors;
  title 'Flaws Report sorted by Station';
run;

```

**Output 20.2** 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

## Reading Data from SAS Data Sets

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 a SAS data set called Average\_Loss.
- 2 Read an observation from SAS data set Weight.
- 3 Calculate a value for variable Percent.
- 4 Execute the DATA step.

## 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 called Investment.
- 2 Calculate a value based on a \$2,000 capital investment and 7% interest each year from 1990 to 2009. Calculate variable values for one observation per iteration of the DO loop.
- 3 Write each observation to 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.

## Writing a Report with a DATA Step

### Example 1: Creating a Report without Creating a Data Set

You can use a DATA step to generate a report without creating a data set by using `_NULL_` in the DATA statement. This approach saves system resources because SAS does not create a data set. The report can contain both TITLE statements and FOOTNOTE statements. If you use a FOOTNOTE statement, be sure to include FOOTNOTE as an option in the FILE statement in the DATA step.

```
title1 'Budget Report'; 1
```

```

title2 'Mid-Year Totals by Department';
footnote 'compiled by Manager,
Documentation Development Department'; 2

data _null_; 3
  set budget; 4
  file print footnote; 5
  MidYearTotal=Jan+Feb+Mar+Apr+May+Jun; 6
  if _n_=1 then 7
    do;
      put @5 'Department' @30 'Mid-Year Total';
    end;
  put @7 Department @35 MidYearTotal; 8
run; 9

```

- 1** Define titles.
- 2** Define the footnote.
- 3** Begin the DATA step. `_NULL_` specifies that no data set is created.
- 4** Read one observation per iteration from data set `Budget`.
- 5** Name the output file for the PUT statements and use the PRINT fileref. By default, the PRINT fileref specifies that the file contains carriage-control characters and titles. The FOOTNOTE option specifies that each page of output contains a footnote.
- 6** Calculate a value for the variable `MidYearTotal` on each iteration.
- 7** Write variable name headings for the report on the first iteration only.
- 8** Write the current values of variables `Department` and `MidYearTotal` for each iteration.
- 9** Execute the DATA step.

The example above uses the FILE statement with the PRINT fileref to produce LISTING output. If you want to print to a file, specify a fileref or a complete filename. Use the PRINT option if you want the file to contain carriage-control characters and titles. The following example shows how to use the FILE statement in this way.

```
file 'external-file' footnote print;
```

You can also use the `data _null_;` statement to write to an external file. For more information about writing to external files, see the FILE statement in [SAS DATA Step Statements: Reference](#), and the SAS documentation for your operating environment.

## Example 2: Creating a Customized Report

You can create very detailed, fully customized reports by using a DATA step with PUT statements. The following example shows a customized report that contains three distinct sections: a header, a table, and a footer. It contains existing SAS variable values, constant text, and values that are calculated as the report is written.

## Output 20.3 Sample of a Customized Report

Around The World Retailers										
EMPLOYEE BUSINESS, TRAVEL, AND TRAINING EXPENSE REPORT										
Employee Name:	ALEJANDRO MARTINEZ	Destination:	CARY, NC	Departure Date:	11JUL2010					
Department:	SALES & MARKETING	Purpose of Trip/Activity:	MARKETING TRAINING		Return Date:	16JUL2010				
Trip ID#:	93-0002519	Activity from:	12JUL1993		to:	16JUL2010				
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
EXPENSE DETAIL		SUN	MON	TUE	WED	THU	FRI	SAT	TOTALS	PAID BY COMPANY EMPLOYEE
----- ----- ----- ----- ----- ----- ----- ----- ----- ----- -----		07/11	07/12	07/13	07/14	07/15	07/16	07/17	TOTALS	PAID BY COMPANY EMPLOYEE
Lodging, Hotel		92.96	92.96	92.96	92.96	92.96			464.80	464.80
Telephone		4.57	4.73						9.30	9.30
Personal Auto 36 miles @.28/mile		5.04					5.04		10.08	10.08
Car Rental, Taxi, Parking, Tolls			35.32	35.32	35.32	35.32	35.32		176.60	176.60
Airlines, Bus, Train (Attach Stub)		485.00					485.00		970.00	970.00
Dues										
Registration Fees		75.00							75.00	75.00
Other (explain below)							5.00		5.00	5.00
Tips (excluding meal tips)		3.00					3.00		6.00	6.00
Meals										
Breakfast							7.79		7.79	7.79
Lunch										
Dinner		36.00	28.63	36.00	36.00	30.00			166.63	166.63
Business Entertainment										
TOTAL EXPENSES		641.57	176.64	179.28	179.28	173.28	541.15		1891.20	1611.40 279.80
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+										
Travel Advance to Employee .....										\$0.00
Reimbursement due Employee (or ATWR) .....										\$279.80
Other: (i.e. miscellaneous expenses and/or names of employees sharing receipt.)										
CAR RENTAL INCLUDE \$5.00 FOR GAS										
APPROVED FOR PAYMENT BY: Authorizing Manager: _____ Emp. # _____										
Employee Signature: _____ Emp. # 1118										
Charge to Division: ATW	Region: TX	Dept: MKT	Acct: 6003	Date: 27JUL2010						

The code shown below generates the report example. You must create your own input data. It is beyond the scope of this discussion to fully explain the code that generated the report example. For a complete explanation of this example, see the *SAS Guide to Report Writing: Examples*.

```

options ls=132 ps=66 pageno=1 nodate;

data travel;

/* infile 'SAS-data-set' missover; */
infile 'c15expense.dat' missover;
input acct div $ region $ deptchg $ rptdate : date9.
      other1-other10 /
      empid empname & $char35. / dept & $char35. /
      purpose & $char35. / dest & $char35. / tripid & $char35. /
      actdate2 : date9. /
      misc1 & $char75. / misc2 & $char75. / misc3 & $char75. /
      misc4 & $char75. /
      misc5 & $char75. / misc6 & $char75. / misc7 & $char75. /
      misc8 & $char75. /
      dptdate : date9. rtrndate : date9. automile permile /
      hotel1-hotel10 /
      phone1-phone10 / peraut1-peraut10 / carrnt1-carrnt10 /
      airlin1-airlin10 / dues1-dues10 / regfee1-regfee10 /
      tips1-tips10 / meals1-meals10 / bkfst1-bkfst10 /
      lunch1-lunch10 / dinner1-dinner10 / busent1-busent10 /
      total1-total10 / empadv reimburs actdate1 : date9.;

run;

proc format;
  value category 1='Lodging, Hotel'
    2='Telephone'
    3='Personal Auto'
    4='Car Rental, Taxi, Parking, Tolls'
    5='Airlines, Bus, Train (Attach Stub)'
    6='Dues'
    7='Registration Fees'
    8='Other (explain below)'
    9='Tips (excluding meal tips)'
    10='Meals'
    11='Breakfast'
    12='Lunch'
    13='Dinner'
    14='Business Entertainment'
    15='TOTAL EXPENSES';

  value blanks 0=' '
    other=(|8.2|);
  value $cuscore ' '='_____';
  value nuscore   .='_____';

run;

data _null_;
  file print;
  title 'Expense Report';
  format rptdate actdate1 actdate2 dptdate rtrndate date9.;
  set travel;

array expenses{15,10} hotel1-hotel10 phone1-phone10
      peraut1-peraut10 carrnt1-carrnt10
      airlin1-airlin10 dues1-dues10
      regfee1-regfee10 other1-other10

```

```

tips1-tips10 meals1-meals10
bkfst1-bkfst10 lunch1-lunch10
dinner1-dinner10 busent1-busent10
total1-total10;

array misc{8} $ misc1-misc8;
array mday{7} mday1-mday7;
dptday=weekday(dptdate);
mday{dptday}=dptdate;
if dptday>1 then
  do dayofwk=1 to (dptday-1);
    mday{dayofwk}=dptdate-(dptday-dayofwk);
  end;
if dptday<7 then
  do dayofwk=(dptday+1) to 7;
    mday{dayofwk}=dptdate+(dayofwk-dptday);
  end;
if rptdate=. then rptdate=&sysdate9"d";

tripnum=substr(tripid,4,2)||'-'||substr(scan(tripid,1),6);

put // @1 'Around The World Retailers' //
@1 'EMPLOYEE BUSINESS, TRAVEL, AND TRAINING EXPENSE REPORT' //

@1 'Employee Name: ' @16 empname
@44 'Destination: ' @57 dest
@106 'Departure Date:' @122 dptdate /

@4 'Department: ' @16 dept
@44 'Purpose of Trip/Activity: ' @70 purpose
@109 'Return Date:' @122 rtrndate /

@6 'Trip ID#: ' @16 tripnum
@107 'Activity from:' @122 actdate1 /

@118 'to:' @122 actdate2 //
@1 '+-----+-----+-----+-----+-----+
'-----+-----+-----+-----+-----+-----+ /'

@1 '|          |          |          |          |          |
| TUE | WED | THU | FRI | SAT | MON |
| PAID BY | PAID BY' /'

@1 '| EXPENSE DETAIL '
|   mday1 mmddyy5. |   mday2 mmddyy5.
|   mday3 mmddyy5. |   mday4 mmddyy5.
|   mday5 mmddyy5. |   mday6 mmddyy5.
|   mday7 mmddyy5.

@100 '| TOTALS | COMPANY EMPLOYEE' ;
do i=1 to 15;

if i=1 or i=10 or i=15 then
  put @1 '|-----|-----|-----|-----|-----|-----|-----|';
if i=3 then
  put @1 '| i category. @16 automile 4.0 @21 'miles @'

```

```

            @28 permile 3.2 @31 '/mile' @37 '|'| @;
else put @1 '|'| i category. @37 '|'| @;
col=38;
do j=1 to 10;
  if j<9 then put @col expenses{i,j} blanks8. '|'| @;
  else if j=9 then put @col expenses{i,j} blanks8. @;
  else put @col expenses{i,j} blanks8. ;
  col+9;
  if j=8 then col+2;
end;
end;
Put @1 '+-----+-----+-----+-----+-----+-----+-----+-----+-----+'
      '-----+-----+-----+-----+-----+-----+-----+-----+-----+ //'

@1 'Travel Advance to Employee .....'
      '.....'
@121 empadv dollar8.2 //'

@1 'Reimbursement due Employee (or ATWR) ..'
      '.....'
@121 reimburs dollar8.2 //'

@1 'Other: (i.e. miscellaneous expenses and/or names of '
      'employees sharing receipt.)' /;
do j=1 to 8;
  put @1 misc{j} ;
end;
put / @1 'APPROVED FOR PAYMENT BY: Authorizing Manager:'
      @48 '_____
@100 'Emp. # _____' ///

@27 'Employee Signature:'
@48 '_____
@100 'Emp. # ' empid ///

@6 'Charge to Division:' @26 div $cuscore.
@39 'Region:' @48 region $cuscore.
@59 'Dept:' @66 deptchg $cuscore.
@79 'Acct:' @86 acct nuscore.
@100 'Date:' @107 rptdate /
      _page_;
run;

```

---

## Example 3: Creating an HTML Report Using ODS and the DATA Step

```

ods html body='your_file.html';

title 'Leading Grain Producers';
title2 'for 2012';

```

```
proc format;
  value $cntry 'BRZ'='Brazil'
    'CHN'='China'
    'IND'='India'
    'INS'='Indonesia'
    'USA'='United States';
run;

data _null_;
  length Country $ 3 Type $ 5;
  input Year country $ type $ Kilotons;
  format country $cntry.;
  label type='Grain';

file print
ods=(variables=(country type kilotons));

put _ods_;

datalines;
2012 BRZ Wheat 3302
2012 BRZ Rice 10035
2012 BRZ Corn 31975
2012 CHN Wheat 109000
2012 CHN Rice 190100
2012 CHN Corn 119350
2012 IND Wheat 62620
2012 IND Rice 120012
2012 IND Corn 8660
2012 INS Wheat .
2012 INS Rice 51165
2012 INS Corn 8925
2012 USA Wheat 62099
2012 USA Rice 7771
2012 USA Corn 236064
;
run;
```

**Output 20.4 HTML File Produced by ODS**

The screenshot shows a Windows application window titled "Results Viewer – SAS Output". Inside the window, there is a title "Leading Grain Producers for 2012" followed by a table with three columns: "Country", "Grain", and "Kilotons". The table lists data for Brazil, China, India, and Indonesia across three grains: Wheat, Rice, and Corn. The data is as follows:

Country	Grain	Kilotons
Brazil	Wheat	3302
	Rice	10035
	Corn	31975
China	Wheat	109000
	Rice	190100
	Corn	119350
India	Wheat	62620
	Rice	120012
	Corn	8660
Indonesia	Wheat	.
Indonesia	Rice	51165
Indonesia	Corn	8925
United States	Wheat	62099
	Rice	7771
	Corn	236064

---

## The DATA Step and ODS

The Output Delivery System (ODS) is a method of delivering output in a variety of formats and making these formats easy to access. ODS provides templates that define the structure of the output from DATA steps and from PROC steps. The DATA step enables you to use the ODS option in a FILE statement and in a PUT statement.

ODS combines raw data with one or more templates to produce several types of output called output objects. Output objects are sent to destinations such as the LISTING destination, the PRINTER destination, or the HTML destination. For more information, see ["Routing and Customizing SAS Output" on page 177](#). For complete information about ODS, see the *SAS Output Delivery System: User's Guide*.

## 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 detailed information about these two phases, see “[The Compilation Phase](#)” on page 444 and “[The Execution Phase](#)” on page 444.

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 has to 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 are methods available for reducing compilation time by decreasing the amount of code that has to be compiled. For more information about how improve performance when running CPU-intensive programs, see “[Techniques for Optimizing CPU Performance](#)” on page 227.

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 219.

Several SAS system options provide information that can help you minimize processing time and optimize performance. For example, the FULLSTIMER option in SAS 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 12, “Optimizing System Performance,”](#) on page 217.

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 20.1 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;
```

*Output 20.5 Log Output for Finding Compilation and Execution Time*

---

```
NOTE: DATA statement used (Total process time):
      real time          0.00 seconds
      cpu time          0.01 seconds

Starting compilation of DATA step: 29JUN12:16:17:54.725
Starting execution of DATA step: 29JUN12:16:17:54.755
The Compile time for this DATA Step is approximately 0:00:00.030000
NOTE: The data set WORK.A has 1 observations and 2 variables.
NOTE: DATA statement used (Total process time):
      real time          0.02 seconds
      cpu time          0.01 seconds
```

Note: Macro statements and macro variables are resolved at compilation time and have no bearing on the time it takes to execute the DATA step. For information about how SAS processes statements with Macro activity, see “[Getting Started with the Macro Facility](#)” in *SAS Macro Language: Reference*, and “[SAS Programs and Macro Processing](#)” in *SAS Macro Language: Reference*.



# 21

## Reading Raw Data

<i>Definition of Reading Raw Data</i>	471
<i>Ways to Read Raw Data</i>	472
<i>Types of Data</i>	473
Definitions	473
Numeric Data	473
Character Data	475
<i>Sources of Raw Data</i>	476
Instream Data	476
Instream Data Containing Semicolons	476
External Files	477
<i>Reading Raw Data with the INPUT Statement</i>	477
Choosing an Input Style	477
List Input	478
Modified List Input	478
Column Input	479
Formatted Input	480
Named Input	481
Additional Data-Reading Features	481
<i>How SAS Handles Invalid Data</i>	483
<i>Reading Missing Values in Raw Data</i>	484
Representing Missing Values in Input Data	484
Special Missing Values in Numeric Input Data	484
<i>Reading Binary Data</i>	485
Definitions	485
Using Binary Informats	486
<i>Reading Column-Binary Data</i>	487
Definition	487
How to Read Column-Binary Data	488
Description of Column-Binary Data Storage	488

---

## Definition of Reading Raw Data

### raw data

is unprocessed data that has not been read into a SAS data set. You can use a DATA step to read raw data into a SAS data set from two sources:

- instream data
- an external file

**Note:** 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 [Chapter 33, “About SAS/ACCESS Software,” on page 757](#).

---

## Ways to Read Raw Data

You can read raw data by using one of the following items:

- SAS statements
- SAS functions
- External File Interface (EFI)
- Import Wizard

When you read raw data with a DATA step, you can use a combination of the INPUT, DATALINES, and INFILE statements. SAS automatically reads your data when you use these statements. For more information about these statements, see [“Reading Raw Data with the INPUT Statement” on page 477](#).

You can also use SAS functions to manipulate external files and to read records of raw data. These functions provide more flexibility in handling raw data. 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](#). For more information about how statements and functions manipulate files differently, see [“Using Functions to Manipulate Files” in SAS Functions and CALL Routines: Reference](#).

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. See [SAS/ACCESS Interface to PC Files: Reference](#) for more information about EFI.

**Note:** If the data file that you are passing to EFI is password protected, you are prompted multiple times for your login ID and password.

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. See [SAS/ACCESS Interface to PC Files: Reference](#) for more information about the wizard.

**Operating Environment Information:** Using external files with your SAS jobs requires that you specify filenames with syntax that is appropriate to your operating environment. See the SAS documentation for your operating environment for more information.

---

# Types of Data

---

## Definitions

**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. [Table 21.2 on page 474](#) 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 21.1** 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

Data	Description	Solution
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 21.2** 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 21.3** 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 (+).[
..	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 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 21.4** *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 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.

Characters in the Data	What to Use	Reason
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 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.

## Sources of Raw Data

### Instream Data

The following example uses the INPUT statement to read in instream data:

```
data weight;
  input PatientID $ Week1 Week8 Week16;
  loss=Week1-Week16;
  datalines;
2477 195 177 163
2431 220 213 198
2456 173 166 155
2412 135 125 116
;
```

**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.

### Instream Data Containing Semicolons

The following 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
;;;
```

---

## External Files

The following example shows how to read in raw data from an external file using the INFILE and INPUT statements:

```
data weight;
  infile file-specification or path-name;
  input PatientID $ Week1 Week8 Week16;
  loss=Week1-Week16;
run;
```

**Note:** See the SAS documentation for your operating environment for information about how to specify a file with the INFILE statement.

---

## Reading Raw Data with the INPUT Statement

---

### Choosing an Input Style

The INPUT statement reads raw data from instream data lines or external files into a SAS data set. You can use the following different input styles, depending on the layout of data values in the records:

- list input
- column input
- formatted 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 in [SAS DATA Step Statements: Reference](#).

## 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 list input follows:

```
data scores;
  length name $ 12;
  input name $ score1 score2;

datalines;
Riley 1132 1187
Henderson 1015 1102
;
```

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 input value longer than 8 bytes, define a variable's length by using a LENGTH, INFORMAT, or ATTRIB statement before the INPUT statement, or by using modified list input, which consists of an informat and the colon modifier in the INPUT statement. See “[Modified List Input](#)” on page 478 for more information.
- 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. See “[Formatted Input](#)” on page 480 for more information.

## 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:

- The & (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.

- The : (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.
- The ~ (tilde) format modifier enables you to read and retain single quotation marks, double quotation marks, and delimiters within character values.

The following 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.

```
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 21.1 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	

## 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
;
```

**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.

Features of column input include the following:

- 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:

- Replace all tabs in the data with single spaces using another editor outside of SAS.
- Use the %INCLUDE statement from the SAS editor to submit your code.
- If you are using the SAS Enhanced Editor, select **Tools**  $\Rightarrow$  **Options**  $\Rightarrow$  **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. 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
;
```

**Note:** You can also use informats to read data that is not aligned in columns. See “[Modified List Input](#)” on page 478 for more information.

Important points about formatted input are:

- 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

You can use named input 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
;
```

**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 21.5 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.

Input	Goal	Use
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 DSD options in the INFILE statement.
	read non-delimited data	\$VARYINGw. informat in the INPUT statement and the LENGTH= and TRUNCOVER options in the INFILE statement.
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.
more than one input file or to control the program flow at EOF		EOF= or END= option in an INFILE statement. multiple INFILE and INPUT statements. FILEVAR=option in an INFILE statement. 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.
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= or DLMSTR= option, DSD option, or both in an INFILE statement.

Input	Goal	Use
the standard tab character		DLM= or DLMSTR= option in an INFILE statement; or the EXPANDTABS option in an INFILE statement.
missing values (with list input or modified list input with the colon modifier)	create observations without compromising data integrity protect data integrity by overriding the default behavior	TRUNCOVER option in an INFILE statement; DLM= or DLMSTR= options, DSD option, or both might also be needed.

For further information about data-reading features, see the INPUT and INFILE statements in *SAS DATA Step Statements: Reference*.

## 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 the SAS documentation for your operating environment for more information.

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.

# Reading Missing Values in Raw Data

## 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 [Chapter 5, “Missing Values,” on page 95](#).

## 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 upper- 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 21.2 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 displays and prints special missing values that use letters in uppercase.

---

## Reading Binary Data

---

### Definitions

**binary data**

is numeric data that is stored in binary form. Binary numbers have a base of two and are represented with the digits 0 and 1.

**packed decimal data**

are 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

are 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.

## Using Binary 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 21.6** Informats for Native or IBM 370 Mode

Description	Native Mode Informs	IBM 370 Mode Informs
ASCII Character	\$w.	\$ASCIIw.
ASCII Numeric	w.d	\$ASCIIw.
EBCDIC Character	\$w.	\$EBCDICw.
EBCDIC Numeric (Standard)	w.d	S370FFw.d
Integer Binary	IBw.d	S370FIBw.d
Positive Integer Binary	PIBw.d	S370FPIBw.d
Real Binary	RBw.d	S370FRBw.d
Unsigned Integer Binary	PIBw.d	S370FIBUw.d, S370FPIBw.d
Packed Decimal	PDw.d	S370FPDw.d
Unsigned Packed Decimal	PKw.d	S370FPDUw.d or PKw.d

Description	Native Mode Informats	IBM 370 Mode Informats
Zoned Decimal	ZDw.d	S370FZDw.d
Zoned Decimal Leading Sign	S370FZDLw.d	S370FZDLw.d
Zoned Decimal Separate Leading Sign	S370FZDSw.d	S370FZDSw.d
Zoned Decimal Separate Trailing Sign	S370FZDTw.d	S370FZDTw.d
Unsigned Zoned Decimal	ZDw.d	S370FZDUw.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.<sup>1</sup> The IBM 370 informats can also be used to read data originally written with the corresponding native mode formats on an IBM mainframe.

**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 on 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

### Definition

column-binary data storage

is 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

---

1. 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.

amount of space. Because card-image data sets remain in existence, SAS provides informats for reading column-binary data. See “[Description of Column-Binary Data Storage](#)” on page 488 for a more detailed explanation of column-binary data storage.

## How to Read 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 21.7** SAS Informats for Reading Column-Binary Data

Informat Name	Description
\$CBw.	reads standard character data from column-binary files
CBw.	reads standard numeric data from column-binary files
PUNCH.d	reads whether a row is punched
ROWw.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:

```
infile file-specification or path-name
      recfm=f
      lrecl=160;
```

**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.

## 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

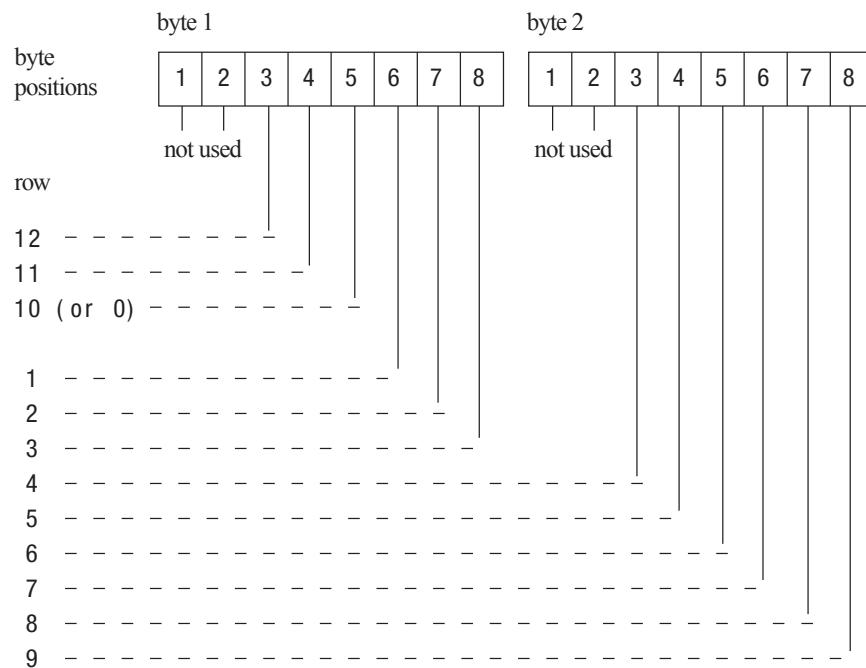
the Hollerith system, each column on a card had a maximum of two punches, one punch in the zone portion, and one in the digit portion. These punches corresponded to a pair of values, and each pair of values corresponded to a specific alphabetic character or sign and numeric digit.

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 21.1** Columns and Rows in a Punched Card

SAS stores each column of column-binary data (a “virtual” punched card) in two bytes. Since each column has only 12 positions and since 2 bytes contain 16 positions, the 4 extra positions within the bytes are located at the beginning of each byte. The following figure shows the correspondence between the rows of “virtual” punched card data and the positions within 2 bytes that SAS uses to store them. SAS stores a punched position as a binary 1 bit and an unpunched position as a binary 0 bit.

**Figure 21.2** Column-Binary Representation on a “Virtual” Punched Card

## 22

# BY-Group Processing in the DATA Step

<i>Definitions for BY-Group Processing</i> .....	491
<i>Syntax for BY-Group Processing</i> .....	492
Syntax .....	492
FIRST. and LAST. Automatic DATA Step Variables .....	492
<i>Understanding BY Groups</i> .....	493
BY Groups with a Single BY Variable .....	493
BY Groups with Multiple BY Variables .....	494
<i>Invoking BY-Group Processing</i> .....	496
<i>Determining Whether the Data Requires Preprocessing for BY-Group Processing</i> .....	496
<i>Preprocessing Input Data for BY-Group Processing</i> .....	497
Sorting Observations for BY-Group Processing .....	497
Indexing for BY-Group Processing .....	497
<i>FIRST. and LAST. DATA Step Variables</i> .....	498
How the DATA Step Identifies BY Groups .....	498
Using a Name Literal as the FIRST. and LAST. Variable .....	498
How SAS Determines FIRST.variable and LAST.variable .....	499
Example 1: Grouping Observations by State, City, and ZIP Code .....	499
Example 2: Grouping Observations by City, State, and ZIP Code .....	500
Example 3: A Change Affecting the FIRST.variable .....	501
<i>Processing BY-Groups in the DATA Step</i> .....	502
Overview .....	502
Processing BY-Groups Conditionally .....	503
Data Not in Alphabetic or Numeric Order .....	504
Data Grouped by Formatted Values .....	504
Example 1: Using GROUPFORMAT with Formats .....	505
Example 2: Using GROUPFORMAT with Formats .....	505

---

## 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](#)” on page 498.

For more information about BY-Group processing, see [Chapter 23, “Reading, Combining, and Modifying SAS Data Sets,” on page 509](#). See also *Combining and Modifying SAS Data Sets: Examples*.

## Syntax for BY-Group Processing

### 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 “[How the DATA Step Identifies BY Groups](#)” on page 498 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 [Example Code 22.1 on page 493](#) and [Example Code 22.2 on page 494](#).

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 22.1 BY Group Using a Single BY Variable (ZipCode)*

BY variable			
ZipCode	State	City	Street
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
			Corsica St
33801	FL	Lakeland	French Ave
33809	FL	Lakeland	Egret Dr
85730	AZ	Tucson	Domenic Ln
			Gleeson Pl

*Example Code 22.1 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 22.2** 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 Using 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 [Example Code 22.1 on page 493](#), 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 22.2** BY Groups with Multiple BY Variables (State and City)

BY variables			
State	City	Street	ZipCode
AZ	Tucson	Domenic Ln	85730
AZ	Tucson	Gleeson Pl	85730
FL	Lakeland	French Ave	33801
FL	Lakeland	Egret Dr	33809
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

Here is the code for creating the output shown in the figure [Figure 22.2 on page 495](#):

**Example Code 22.3** 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 22.4** 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](#)” on page 692.

# 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 503 for more information.

## Using a Name Literal as the FIRST. and LAST. Variable

### 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. 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](#)” on page 499 and “[How SAS Identifies the Beginning and End of a BY Group](#)” in *SAS DATA Step Statements: Reference*.

## 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 changes 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.

## 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.
- 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 22.1 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 22.3 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](#)” on page 499 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 22.2** 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_= 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 22.1** 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 22.3 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 22.2 SAS Output Using the BY Statement GROUPFORMAT Option**

Score Categories		
<b>Score=Low</b>		
Obs	name	Score
<b>1</b>	Jon	Low
<b>Score=Medium</b>		
Obs	name	Score
<b>2</b>	Anthony	Medium
<b>3</b>	Miguel	Medium
<b>4</b>	Joseph	Medium
<b>Score=High</b>		
Obs	name	Score
<b>5</b>	Ian	High
<b>6</b>	Jan	High



# 23

## Reading, Combining, and Modifying SAS Data Sets

---

<i>Definitions for Reading, Combining, and Modifying SAS Data Sets</i>	509
<i>Overview of Tools</i>	510
<i>Reading SAS Data Sets</i>	510
Reading a Single SAS Data Set	510
Reading from Multiple SAS Data Sets	511
Controlling the Reading and Writing of Variables and Observations	511
<i>Combining SAS Data Sets: Basic Concepts</i>	512
What You Need to Know Before Combining Information Stored in Multiple SAS Data Sets	512
The Four Ways That Data Can Be Related	512
Access Methods: Sequential versus Direct	515
Overview of Methods for Combining SAS Data Sets	516
Overview of Tools for Combining SAS Data Sets	519
How to Prepare Your Data Sets	521
<i>Combining SAS Data Sets: Methods</i>	523
Concatenating	523
Interleaving	527
One-to-One Reading	532
One-to-One Merging	534
Match-Merging	539
Updating with the UPDATE and the MODIFY Statements	543
<i>Error Checking When Using Indexes to Randomly Access or Update Data</i>	555
The Importance of Error Checking	555
Error-Checking Tools	555
Example 1: Routing Execution When an Unexpected Condition Occurs	556
Example 2: Using Error Checking on All Statements That Use KEY=	559

---

## Definitions for Reading, Combining, and Modifying SAS Data Sets

In the context of DATA step processing, the terms reading, combining and modifying have these meanings:

**Reading a SAS data set**

refers to opening a SAS data set and bringing an observation into the program data vector for processing.

**Combining SAS data sets**

refers to reading data from two or more SAS data sets and processing them by

- concatenating
- interleaving
- one-to-one reading
- one-to-one merging
- match-merging
- updating a master data set with a transaction data set

The methods for combining SAS data sets are defined in “[Combining SAS Data Sets: Methods](#)” on page 523.

**Modifying SAS data sets**

refers to using the MODIFY statement to update information in a SAS data set in place. The MODIFY statement can save disk space because it modifies data in place, without creating a copy of the data set. You can modify a SAS data set with programming statements or with information that is stored in another data set.

## Overview of Tools

The primary tools that are used for reading, combining, and modifying SAS data sets are four statements: SET, MERGE, MODIFY, and UPDATE. This section describes these tools and shows examples. For complete information about these statements, see the [SAS DATA Step Statements: Reference](#).

## Reading SAS Data Sets

### Reading a Single SAS Data Set

To read data from an existing SAS data set, use a SET statement. In this example, the DATA step creates data set Perm.Tour155\_PeakCost by reading data from data set Perm.Tour155\_Basic\_Cost and by calculating values for the three new variables Total\_Cost, Peak\_Cost, and Average\_Night\_Cost.

```
data perm.tour155_peakcost;
  set perm.tour155_basic_cost;
  Total_Cost=AirCost+LandCost;
  Peak_Cost=(AirCost*1.15);
  Average_Night_Cost=LandCost/Nights;
```

```
run;
```

---

## Reading from Multiple SAS Data Sets

You can read from multiple SAS data sets and combine and modify data in different ways. Here are some examples:

- combine two or more input data sets to create one output data set
- merge data from two or more input data sets that share a common variable
- update a master file based on transaction records

For details about reading from multiple SAS data sets, see “[Combining SAS Data Sets: Methods](#)” on page [523](#).

---

## Controlling the Reading and Writing of Variables and Observations

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, however, control which variables and observations you want to read and write by using SAS statements, data set options, and functions. The statements and data set options that you can use are listed in the following table.

**Table 23.1** Statements and Options That Control Reading and Writing

Task	Statements	Data Set Options	System Options
Control variables	DROP	DROP=	
	KEEP	KEEP=	
	RENAME	RENAME=	
Control observations	WHERE	WHERE=	FIRSTOBS=
	subsetting IF	FIRSTOBS=	OBS=
	DELETE	OBS=	
	REMOVE		
	OUTPUT		

Use statements or data set options (such as KEEP= and DROP=) to control the variables and observations that you want to write to the output data set. The WHERE statement is an exception: it controls which observations are read into the program data vector based on the value of a variable. You can use data set options (including WHERE=) on input or output data sets, depending on their function and what you want to control. You can also use SAS system options to control your data.

---

## Combining SAS Data Sets: Basic Concepts

---

### What You Need to Know Before Combining Information Stored in Multiple SAS Data Sets

Many applications require input data to be in a specific format before the data can be processed to produce meaningful results. The data typically comes from multiple sources and might be in different formats. Therefore, you often, if not always, have to take intermediate steps to logically relate and process data before you can analyze it or create reports from it.

Application requirements vary, but there are common factors for all applications that access, combine, and process data. Once you have determined what you want the output to look like, you must perform the following tasks:

- Determine how the input data is related.
- Ensure that the data is properly sorted or indexed, if necessary.
- Select the appropriate access method to process the input data.
- Select the appropriate SAS tools to complete the task.

---

### The Four Ways That Data Can Be Related

#### Data Relationship Categories

Relationships among multiple sources of input data exist when each of the sources contains common data, either at the physical or logical level. For example, employee data and department data could be related through an employee ID variable that shares common values. Another data set could contain numeric sequence numbers whose partial values logically relate it to a separate data set by observation number.

You must be able to identify the existing relationships in your data. This knowledge is crucial for understanding how to process input data in order to produce desired results. All related data fall into one of these four categories, characterized by how observations relate among the data sets:

- one-to-one
- one-to-many
- many-to-one
- many-to-many

To obtain the results that you want, you should understand how each of these methods combines observations, how each method treats duplicate values of

common variables, and how each method treats missing values or nonmatched values of common variables. Some of the methods also require that you preprocess your data sets by sorting them or by creating indexes. See the description of each method in “[Combining SAS Data Sets: Methods](#)” on page 523.

## One-to-One Relationship

In a one-to-one relationship, typically a single observation in one data set is related to a single observation from another based on the values of one or more selected variables. A one-to-one relationship implies that each value of the selected variable occurs no more than once 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 following example, observations in data sets Salary and Taxes are related by common values for EmployeeNumber.

**Figure 23.1** One-to-One Relationship

SALARY		TAXES	
EmployeeNumber	Salary	EmployeeNumber	TaxBracket
1234	55000	1111	0.18
3333	72000	1234	0.28
4876	32000	3333	0.32
5489	17000	4222	0.18
		4876	0.24

The diagram illustrates a one-to-one relationship between two data sets: SALARY and TAXES. The SALARY data set has four observations with EmployeeNumbers 1234, 3333, 4876, and 5489, and their corresponding salaries. The TAXES data set has five observations with EmployeeNumbers 1111, 1234, 3333, 4222, and 4876, and their corresponding tax brackets. Arrows connect the EmployeeNumber 1234 in the SALARY set to the EmployeeNumber 1234 in the TAXES set, and the EmployeeNumber 3333 in the SALARY set to the EmployeeNumber 3333 in the TAXES set. There are also arrows from the EmployeeNumber 4876 in the SALARY set to the EmployeeNumber 4876 in the TAXES set, and from the EmployeeNumber 5489 in the SALARY set to the EmployeeNumber 4222 in the TAXES set. This indicates that while each employee in the SALARY set is associated with exactly one tax bracket in the TAXES set, some tax brackets appear for more than one employee in the SALARY set.

## 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 observation 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 following example, observations in data sets One and Two are related by common values for variable A. Values of A are unique in data set One but not in data set Two.

**Figure 23.2** One-to-Many Relationship

ONE			TWO		
A	B	C	A	E	F
1	5	6	1	2	0
3	3	4	1	3	99
			1	4	88
			1	5	77
			2	1	66
			2	2	55
			3	4	44

In the following example, observations in data sets One, Two, and Three are related by common values for variable ID. Values of ID are unique in data sets One and Three but not in Two. For values 2 and 3 of ID, a one-to-many relationship exists between observations in data sets One and Two. A many-to-one relationship exists between observations in data sets Two and Three.

**Figure 23.3** One-to-Many and Many-to-One Relationships

ONE		TWO		THREE	
ID	Name	ID	Sales	ID	Quota
1	Joe Smith	1	28000	1	15000
2	Sally Smith	2	30000	2	7000
3	Cindy Long	2	40000	3	15000
4	Sue Brown	3	15000	4	5000
5	Mike Jones	3	20000	5	8000
		3	25000	4	35000
			5	40000	

## Many-to-Many Relationships

The many-to-many category implies that multiple observations from each input data set can be related based on values of one or more common variables.

In the following example, observations in data sets BreakDown and Maintenance are related by common values for variable Vehicle. Values of Vehicle are not unique in either data set. A many-to-many relationship exists between observations in these data sets for values AAA and CCC of Vehicle.

**Figure 23.4** Many-to-Many Relationship

BREAKDOWN		MAINTENANCE	
Vehicle	BreakDownDate	Vehicle	MaintenanceDate
AAA	02MAR99	AAA	03JAN99
AAA	20MAY99	AAA	05APR99
AAA	19JUN99	AAA	10AUG99
AAA	29NOV99	CCC	28JAN99
BBB	04JUL99	CCC	16MAY99
CCC	31MAY99	CCC	07OCT99
CCC	24DEC99	DDD	24FEB99
		DDD	22JUN99
		DDD	19SEP99

## Access Methods: Sequential versus Direct

### Overview

Once you have established data relationships, the next step is to determine the best mode of data access to relate the data. You can access observations sequentially in the order in which they appear in the physical file. Or you can access them directly. That is, you can go straight to an observation in a SAS data set without having to process each observation that precedes it.

### Sequential Access

The simplest and perhaps most common way to process data with a DATA step is to read observations in a data set sequentially. You can read observations sequentially using the SET, MERGE, UPDATE, or MODIFY statements. You can also use the SAS File I/O functions. OPEN, FETCH, and FETCHOBS are examples.

### Direct Access

Direct access allows a program to access specific observations based on one of two methods:

- by an observation number
- by the value of one or more variables through a simple or composite index

To access observations directly by their observation number, use the POINT= option with the SET or MODIFY statement. The POINT= option names a variable whose current value determines which observation a SET or MODIFY statement reads.

To access observations directly based on the values of one or more specified variables, you must first create an index for the variables and then read the data set using the KEY= option. The KEY= option can be specified with either the SET statement or the MODIFY statement. An index is a separate structure that contains

the data values of the key variable or variables, paired with a location identifier for the observations containing the value.

**Note:** You can also use the SAS File I/O functions such as CUROBS, NOTE, POINT, and FETCHOBS to access observations by observation number.

## Overview of Methods for Combining SAS Data Sets

### Methods for Combining SAS Data Sets

You can use these methods to combine SAS data sets:

- concatenating
- interleaving
- one-to-one reading
- one-to-one merging
- match merging
- updating

### Concatenating

The following figure shows the results of concatenating two SAS data sets. Concatenating the data sets appends the observations from one data set to another data set. The DATA step reads Data1 sequentially until all observations have been processed, and then reads Data2. Data set Combined contains the results of the concatenation. Note that the data sets are processed in the order in which they are listed in the SET statement.

*Figure 23.5 Concatenating Two Data Sets*

#### DATA1      DATA2      COMBINED

Year	Year	Year
1991	1991	1991
1992	1992	1992
1993	1993	1993
1994	1994	1994
1995	1995	1995

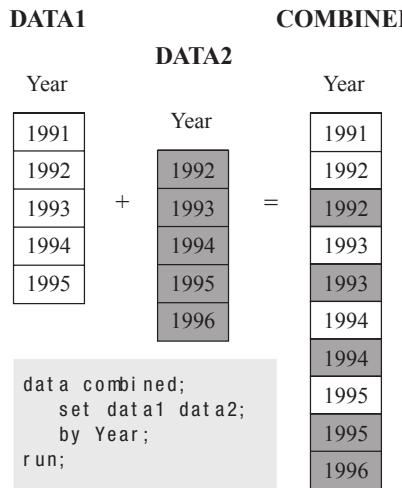
+                            =

<pre>data combined;   set data1 data2; run;</pre>
---

## Interleaving

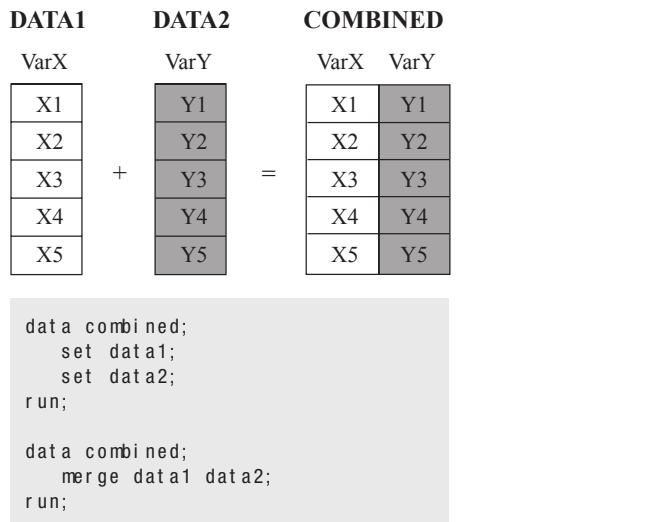
The following figure shows the results of interleaving two SAS data sets. Interleaving intersperses observations from two or more data sets, based on one or more common variables. Data set Combined shows the result.

**Figure 23.6** Interleaving Two Data Sets



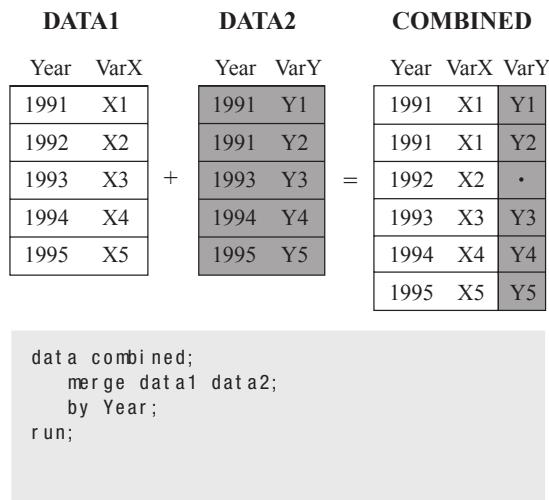
## One-to-One Reading and One-to-One Merging

The following figure shows the results of one-to-one reading and one-to-one merging. One-to-one reading combines observations from two or more SAS data sets by creating observations that contain all of the variables from each contributing data set. Observations are combined based on their relative position in each data set, that is, the first observation in one data set with the first in the other, and so on. The DATA step stops after it has read the last observation from the smallest data set. One-to-one merging is similar to a one-to-one reading, with two exceptions: you use the MERGE statement instead of multiple SET statements, and the DATA step reads all observations from all data sets. Data set Combined shows the result.

**Figure 23.7** One-to-One Reading and One-to-One Merging

## Match-Merging

The following figure shows the results of match-merging. Match-merging combines observations from two or more SAS data sets into a single observation in a new data set based on the values of one or more common variables. Data set Combined shows the results.

**Figure 23.8** Match-Merging Two Data Sets

## Updating

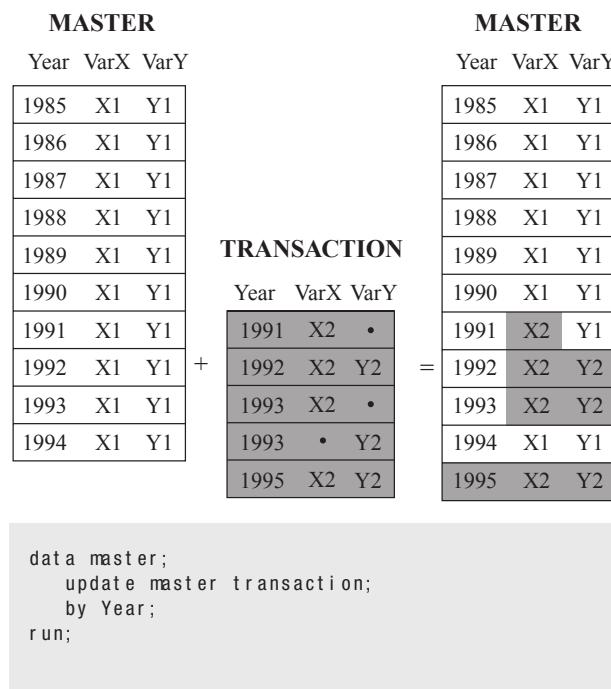
The following figure shows the results of updating a master data set. Updating uses information from observations in a transaction data set to delete, add, or alter information in observations in a master data set. You can update a master data set by using the UPDATE statement or the MODIFY statement. If you use the UPDATE statement, your input data sets must be sorted by the values of the variables listed in the BY statement. (In this example, Master and Transaction are both sorted by

Year.) If you use the MODIFY statement, your input data does not need to be sorted.

UPDATE replaces an existing file with a new file, enabling you to add, delete, or rename columns. MODIFY performs an update in place by rewriting only those records that have changed, or by appending new records to the end of the file.

Note that by default, UPDATE and MODIFY do not replace nonmissing values in a master data set with missing values from a transaction data set.

*Figure 23.9 Updating a Master Data Set*



## Overview of Tools for Combining SAS Data Sets

### Using Statements and Procedures

You can choose from a variety of SAS tools for accessing, combining, and processing your data. The following table describes the DATA step statements and procedures that you can use for combining SAS data sets.

**Table 23.2** Statements or Procedures for Combining SAS Data Sets

Statement or Procedure	Action Performed	Access Method		Can Use with BY statement	Comments
		Sequential	Direct		
BY	Controls the operation of a SET, MERGE, UPDATE, or MODIFY statement in the DATA step and sets up special grouping variables.	NA	NA	NA	BY-group processing is a means of processing observations that have the same values of one or more variables.
MERGE	Reads observations from two or more SAS data sets and joins them into a single observation.	X		X	When using MERGE with BY, the data must be sorted or indexed on the BY variable.
MODIFY	Processes observations in a SAS data set in place. (Contrast with UPDATE.)	X	X	X	Sorted or indexed data are not required for use with BY, but are recommended for performance.
SET	Reads an observation from one or more SAS data sets.	X	X	X	Use KEY= or POINT= statement options for directly accessing data.
UPDATE	Applies transactions to observations in a master SAS data set. UPDATE does not update observations in place; it produces an updated copy of the current data set.	X		X	Both the master and transaction data sets must be sorted by or indexed on the BY variable.
PROC APPEND	Adds the observations from one SAS data set to the end of another SAS data set.	X			
PROC SQL*	Reads an observation from one or more SAS data sets; reads observations from up to 32 SAS data sets and joins them into single observations; manipulates observations in a SAS data set in place; easily produces a Cartesian product.	X	X	X	All three access methods are available in PROC SQL, but the access method is chosen by the internal optimizer.

\* PROC SQL is the SAS implementation of Structured Query Language. In addition to expected SQL capabilities, PROC SQL includes additional capabilities specific to SAS, such as the use of formats and SAS macro language.

## Using Error Checking

You can use the \_IORC\_ automatic variable and the SYSRC autocall macro to perform error checking in a DATA step. Use these tools with the MODIFY statement or with the SET statement and the KEY= option. For more information about these tools, see “[Error Checking When Using Indexes to Randomly Access or Update Data](#)” on page 555.

---

# How to Prepare Your Data Sets

## Guidelines to Prepare Your Data Sets

Before combining SAS data sets, follow these guidelines to produce the results that you want:

- Know the structure and the contents of the data sets.
- Look at sources of common problems.
- Ensure that observations are in the correct order, or that they can be retrieved in the correct order (for example, by using an index).
- Test your program.

## Knowing the Structure and Contents of the Data Sets

To help determine how your data is related, look at the structure of the data sets. To see the data set structure, execute the DATASETS procedure, the CONTENTS procedure, or access the SAS Explorer window in your windowing environment to display the descriptor information. Descriptor information includes the following information:

- the number of observations in each data set
- the name and attributes of each variable
- an alphabetic list of extended attributes (including data set and variable extended attributes)
- a list of indexes and index attributes

To print a sample of the observations, use the PRINT procedure or the REPORT procedure.

You can also use functions such as VTYPE, VLENGTH, and VLENGTHX to show specific descriptor information. For complete information about these functions, see [SAS Functions and CALL Routines: Reference](#).

## Looking at Sources of Common Problems

If your program does not execute correctly, review your input data for the following errors:

- variables that have the same name but that represent different data

SAS includes only one variable of a given name in the new data set. If two data sets have variables with the same names but different data, the values from the last data set that was read are written over the values from the previously read data sets.

To correct the error, you can rename variables before you combine the data sets by using the RENAME= data set option in the SET, UPDATE, or MERGE statement. Or you can use the DATASETS procedure.

- common variables with the same data but different attributes

The way SAS handles these differences depends on which attributes are different:

- type attribute

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. The SAS statements that you use depend on the nature of the variable.

- length attribute

If the length attribute is different, SAS takes the length from the first data set that contains the variable. In the following example, all data sets that are listed in the MERGE statement contain the variable Mileage. In Quarter1, the length of the variable Mileage is four bytes; in Quarter2, it is eight bytes and in Quarter3 and Quarter4, it is six bytes. In the output data set Yearly, the length of the variable Mileage is four bytes, which is the length derived from Quarter1.

```
data yearly;
  merge quarter1 quarter2 quarter3 quarter4;
  by Account;
run;
```

To override the default and set the length yourself, specify the appropriate length in a LENGTH statement that precedes the SET, MERGE, or UPDATE statement.

**Note:** If the length of a variable changes as a result of combining data sets, SAS prints a warning message to the log and issues a nonzero return code. For example, on z/OS, the value for SYSRC would be 4. If you do not want SAS to issue a warning, you can turn it off by setting the VARLENCHK system option to NOWARN. For example, if you expect truncation of data because you are removing insignificant blanks from the end of a character value, you might not want the warnings. For more information, see ["VARLENCHK= System Option" in SAS System Options: Reference](#).

- label, format, and informat attributes

If any of these attributes are different, SAS takes the attribute from the first data set that contains the variable with that attribute. However, any label, format, or informat that you explicitly specify overrides a default. If all data sets contain explicitly specified attributes, the one specified in the first data set overrides the others. To ensure that the new output data set has the attributes that you prefer, use an ATTRIB statement.

You can also use SAS File I/O functions, such as VLABEL, VLABELX, and other Variable Information functions to access this information. For complete information about these functions, see [SAS Functions and CALL Routines: Reference](#).

- extended attributes

Like formats and labels, extended attributes are automatically passed from the input data set to the output data set in a DATA step. If two input data sets contain extended attributes, then SAS preserves the extended attributes from the first data set read and applies those attributes to the output data set. To ensure that the new output data set has the extended attributes that you prefer, use the DATASETS procedure to add, delete, remove, set, and update extended attributes. For more information about the DATASETS procedure see “[DATASETS Procedure](#)” in *Base SAS Procedures Guide*.

## Ensuring Correct Order

If you use BY-group processing with the UPDATE, SET, and MERGE statements to combine data sets, ensure that the observations in the data sets are sorted in the order of the variables that are listed in the BY statement, or that the data sets have an appropriate index. If you use BY-group processing in a MODIFY statement, your data does not need to be sorted, but sorting the data improves efficiency. The BY variable or variables must be common to both data sets, and they must have the same attributes. For more information, see [Chapter 22, “BY-Group Processing in the DATA Step,” on page 491](#).

## Testing Your Program

As a final step in preparing your data sets, you should test your program. Create small temporary SAS data sets that contain a sample of observations that test all of your program’s logic. If your logic is faulty and you get unexpected output, you can use the DATA step debugger to debug your program. For complete information about the DATA Step Debugger, see [SAS Data Set Options: Reference](#).

---

# Combining SAS Data Sets: Methods

---

## Concatenating

### Definition

Concatenating data sets is the combining of two or more data sets, one after the other, into a single data set. The number of observations in the new data set is the sum of the number of observations in the original data sets. The order of observations is sequential. All observations from the first data set are followed by all observations from the second data set, and so on.

In the simplest case, all input data sets contain the same variables. If the input data sets contain different variables, observations from one data set have missing values for variables defined only in other data sets. In either case, the variables in the new data set are the same as the variables in the old data sets.

## Syntax

Use this form of the SET statement to concatenate data sets:

`SET data-set(s);`

where

*data-set*

specifies any valid SAS data set name.

For a complete description of valid SAS data set names, see the SET statement in [SAS DATA Step Statements: Reference](#).

## DATA Step Processing during Concatenation

### Compilation phase

SAS reads the descriptor information of each data set that is named in the SET statement and then creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step.

### Execution — Step 1

SAS reads the first observation from the first data set into the program data vector. It processes the first observation and executes other statements in the DATA step. It then writes the contents of the program data vector to the new data set.

The SET statement does not reset the values in the program data vector to missing, except for variables whose value is calculated or assigned during the DATA step. Variables that are created by the DATA step are set to missing at the beginning of each iteration of the DATA step. Variables that are read from a data set are not.

### Execution — Step 2

SAS continues to read one observation at a time from the first data set until it finds an end-of-file indicator. The values of the variables in the program data vector are then set to missing, and SAS begins reading observations from the second data set, and so on, until it reads all observations from all data sets.

## Example 1: Concatenation Using the DATA Step

In this example, each data set contains the variables Common and Number, and the observations are arranged in the order of the values of Common. Generally, you concatenate SAS data sets that have the same variables. In this case, each data set also contains a unique variable to show the effects of combining data sets more clearly. The following shows the Animal and the Plant input data sets in the library that is referenced by the libref Example:

Animal				Plant			
OBS	Common	Animal	Number	OBS	Common	Plant	Number
1	a	Ant	5	1	g	Grape	69
2	b	Bird		2	h	Hazelnut	55
3	c	Cat	17	3	i	Indigo	.
4	d	Dog	9	4	j	Jicama	14
5	e	Eagle		5	k	Kale	5

6	f	Frog
76		
6	1	Lentil
		77

The following program uses a SET statement to concatenate the data sets and then prints the results:

```
data concatenation;
    set animal plant;
run;

proc print data=concatenation;
    var Common Animal Plant Number;
    title 'Data Set CONCATENATION';
run;
```

**Output 23.1** Concatenated Data Sets (DATA Step)

Data Set CONCATENATION				
Obs	Common	Animal	Plant	Number
1	a	Ant		5
2	b	Bird		.
3	c	Cat		17
4	d	Dog		9
5	e	Eagle		.
6	f	Frog		76
7	g		Grape	69
8	h		Hazelnut	55
9	i		Indigo	.
10	j		Jicama	14
11	k		Kale	5
12	l		Lentil	77

The resulting data set CONCATENATION has 12 observations, which is the sum of the observations from the combined data sets. The program data vector contains all variables from all data sets. The values of variables found in one data set but not in another are set to missing.

## Example 2: Concatenation Using SQL

You can also use the SQL language to concatenate tables. In this example, SQL reads each row in both tables and creates a new table named Combined. The following shows the YEAR1 and YEAR2 input tables:

YEAR1	YEAR2
Date1	Date2

2009	
2010	2010
2011	2011
2012	2012
	2013
	2014

The following SQL code creates and prints the table Combined.

```
proc sql;
  title 'SQL Table Combined';
  create table combined as
    select * from year1
    union all
    select * from year2;
    select * from combined;
quit;
```

**Output 23.2** Concatenated Tables (SQL)

Year
2009
2010
2011
2012
2010
2011
2012
2013
2014

## Appending Files

Instead of concatenating data sets or tables, you can append them and produce the same results as concatenation. SAS concatenates data sets (DATA step) and tables (SQL) by reading each row of data to create a new file. To avoid reading all the records, you can append the second file to the first file by using the APPEND procedure:

```
proc append base=year1 data=year2;
run;
```

The YEAR1 file contains all rows from both tables.

**Note:** You cannot use PROC APPEND to add observations to a SAS data set in a sequential library.

## Efficiency

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.

# Interleaving

## Definition

Interleaving uses a SET statement and a BY statement to combine multiple data sets into one new data set. The number of observations in the new data set is the sum of the number of observations from the original data sets. However, the observations in the new data set are arranged by the values of the BY variable or variables and, within each BY group, by the order of the data sets in which they occur. You can interleave data sets either by using a BY variable or by using an index.

## Syntax

Use this form of the SET statement to interleave data sets when you use a BY variable:

`SET data-set(s);`

`BY variable(s);`

where

***data-set***

specifies a one-level name, a two-level name, or one of the special SAS data set names.

***variable***

specifies each variable by which the data set is sorted. These variables are referred to as BY variables for the current DATA or PROC step.

Use this form of the SET statement to interleave data sets when you use an index:

`SET data-set-1 . . . data-set-n KEY= index;`

where

***data-set***

specifies a one-level name, a two-level name, or one of the special SAS data set names.

***index***

provides nonsequential access to observations in a SAS data set, which are based on the value of an index variable or key.

For a complete description of the SET statement, including SET with the KEY= option, see the SET statement in [SAS DATA Step Statements: Reference](#).

## Sort Requirements

Before you can interleave data sets, the observations must be sorted or grouped by the same variable or variables that you use in the BY statement, or you must have an appropriate index for the data sets.

## DATA Step Processing during Interleaving

### Compilation phase

- SAS reads the descriptor information of each data set that is named in the SET statement and then creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step.
- SAS creates the FIRST.variable and LAST.variable for each variable listed in the BY statement.

### Execution — Step 1

SAS compares the first observation from each data set that is named in the SET statement to determine which BY group should appear first in the new data set. It reads all observations from the first BY group from the selected data set. If this BY group appears in more than one data set, it reads from the data sets in the order in which they appear in the SET statement. The values of the variables in the program data vector are set to missing each time SAS starts to read a new data set and when the BY group changes.

### Execution — Step 2

SAS compares the next observations from each data set to determine the next BY group and then starts reading observations from the selected data set in the SET statement that contains observations for this BY group. SAS continues until it has read all observations from all data sets.

## Example 1: Interleaving in the Simplest Case

In this example, each data set contains the BY variable Common, and the observations are arranged in order of the values of the BY variable. The following shows the Animal and the Plant input data sets in the library that is referenced by the libref 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 uses SET and BY statements to interleave the data sets, and prints the results:

```
data interleaving;
  set animal plant;
  by Common;
run;
```

```
proc print data=interleaving;
  title 'Data Set INTERLEAVING';
  run;
```

**Output 23.3 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

The resulting data set INTERLEAVING has 12 observations, which is the sum of the observations from the combined data sets. The new data set contains all variables from both data sets. The value of variables found in one data set but not in the other are set to missing, and the observations are arranged by the values of the BY variable.

## Example 2: Interleaving with Duplicate Values of the BY Variable

If the data sets contain duplicate values of the BY variables, the observations are written to the new data set in the order in which they occur in the original data sets. This example contains duplicate values of the BY variable Common. The following shows the Animal1 and Plant1 input data sets:

Animal1			Plant1		
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 uses SET and BY statements to interleave the data sets, and prints the results:

```
data interleaving2;
  set animal1 plant1;
  by Common;
run;

proc print data=interleaving2;
  title 'Data Set INTERLEAVING2: Duplicate BY Values';
run;
```

**Output 23.4 Interleaved Data Sets with Duplicate Values of the BY Variable**

Data Set INTERLEAVING2: Duplicate BY Values			
Obs	Common	Animal1	Plant1
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

The number of observations in the new data set is the sum of the observations in all the data sets. The observations are written to the new data set in the order in which they occur in the original data sets.

### Example 3: Interleaving with Different BY Values in Each Data Set

The data sets Animal2 and Plant2 both contain values that are present in one data set but not in the other. The following shows the Animal2 and the Plant2 input data sets:

Animal2	Plant2				
OBS	Common	Animal2	OBS	Common	Plant2
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

This program uses SET and BY statements to interleave these data sets, and prints the results:

```

data interleaving3;
  set animal2 plant2;
  by Common;
run;

proc print data=interleaving3;
  title 'Data Set INTERLEAVING3: Different BY Values';
run;

```

*Output 23.5 Interleaving Data Sets with Different BY Values*

## Data Set INTERLEAVING3: Different BY Values

Obs	Common	Animal2	Plant2
1	a	Ant	
2	a		Apple
3	b		Banana
4	c	Cat	
5	c		Coconut
6	d	Dog	
7	e	Eagle	
8	e		Eggplant
9	f		Fig

The resulting data set has nine observations arranged by the values of the BY variable.

## Comments and Comparisons

- In other languages, the term merge is often used to mean interleave. SAS reserves the term merge for the operation in which observations from two or more data sets are combined into one observation. The observations in interleaved data sets are not combined; they are copied from the original data sets in the order of the values of the BY variable.
- If one table has multiple rows with the same BY value, the DATA step preserves the order of those rows in the result.
- To use the DATA step, the input tables must be appropriately sorted or indexed. SQL does not require the input tables to be in order.

---

## One-to-One Reading

### Definition

One-to-one reading combines observations from two or more data sets into one observation by using two or more SET statements to read observations independently from each data set. This process is also called one-to-one matching. 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 the values that were read in from earlier data sets.

### Syntax

Use this form of the SET statement for one-to-one reading:

```
SET data-set-1;  
SET data-set-2;
```

where

*data-set-1*

specifies a one-level name, a two-level name, or one of the special SAS data set names. *data-set-1* is the first file that the DATA step reads.

*data-set-2*

specifies a one-level name, a two-level name, or one of the special SAS data set names. *data-set-2* is the second file that the DATA step reads.

**CAUTION! Use care when you combine data sets with multiple SET statements.**

Using multiple SET statements to combine observations can produce undesirable results. Test your program on representative samples of the data sets before using this method to combine them.

For a complete description, see SET Statement in [SAS DATA Step Statements: Reference](#).

### DATA Step Processing during a One-to-One Reading

#### Compilation phase

SAS reads the descriptor information of each data set named in the SET statement and then creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step.

#### Execution — Step 1

When SAS executes the first SET statement, SAS reads the first observation from the first data set into the program data vector. The second SET statement reads the first observation from the second data set into the program data vector. If both data sets contain the same variables, the values from the second data set replace the values from the first data set, even if the value is missing. After reading the first observation from the last data set and executing any other statements in the DATA step, SAS writes the contents of the program data vector to the new data set. The SET statement does not reset the values in the program

data vector to missing, except for those variables that were created or assigned values during the DATA step.

#### Execution — Step 2

SAS continues reading from one data set and then the other until it detects an end-of-file indicator in one of the data sets. SAS stops processing with the last observation of the shortest data set and does not read the remaining observations from the longer data set.

### Example 1: One-to-One Reading: Processing an Equal Number of Observations

The SAS data sets Animal and Plant both contain the variable Common, and are arranged by the values of that variable. The following shows the Animal and the Plant 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	g	Fig

The following program uses two SET statements to combine observations from Animal and Plant, and prints the results:

```

data twosets;
  set animal;
  set plant;
run;

proc print data=twosets;
  title 'Data Set TWOSETS - Equal Number of Observations';
run;
```

**Output 23.6 Data Set Created from Two Data Sets That Have Equal Observations**

Data Set TWOSETS - 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

Each observation in the new data set contains all the variables from all the data sets. Note, however, that the Common variable value in observation 6 contains a "g." The value of Common in observation 6 of the Animal data set was overwritten by the value in Plant, which was the data set that SAS read last.

## Comments and Comparisons

- The results that are obtained by reading observations using two or more SET statements are similar to those that are obtained by using the MERGE statement with no BY statement. However, with one-to-one reading, SAS stops processing before all observations are read from all data sets if the number of observations in the data sets is not equal.
- Using multiple SET statements with other DATA step statements makes the following applications possible:
  - merging one observation with many
  - conditionally merging observations
  - reading from the same data set twice

## One-to-One Merging

### Definition

One-to-one merging combines observations from two or more SAS data sets into a single observation in a new data set. To perform a one-to-one merge, use the MERGE statement without a BY statement. SAS combines the first observation from all data sets in the MERGE statement into the first observation in the new data set, the second observation from all data sets into the second observation in the new data set, and so on. In a one-to-one merge, the number of observations in the new data set equals the number of observations in the largest data set that was named in the MERGE statement.

If you use the MERGENOBY= SAS system option, you can control whether SAS issues a message when MERGE processing occurs without an associated BY statement.

## Syntax

Use this form of the MERGE statement to merge SAS data sets:

MERGE *data-set(s)*;

where

*data-set*

names at least two existing SAS data sets.

**CAUTION! Avoid using duplicate values or different values of common variables.**

One-to-one merging with data sets that contain duplicate values of common variables can produce undesirable results. If a variable exists in more than one data set, the value from the last data set that is read is the one that is written to the new data set. The variables are combined exactly as they are read from each data set. Using a one-to-one merge to combine data sets with different values of common variables can also produce undesirable results. If a variable exists in more than one data set, the value from the last data set read is the one that is written to the new data set even if the value is missing. Once SAS has processed all observations in a data set, all subsequent observations in the new data set have missing values for the variables that are unique to that data set.

For a complete description of the MERGE statement, see the MERGE statement in [SAS DATA Step Statements: Reference](#).

## DATA Step Processing during One-to-One Merging

### Compilation phase

SAS reads the descriptor information of each data set that is named in the MERGE statement. Then, SAS creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step.

### Execution — Step 1

SAS reads the first observation from each data set into the program data vector, reading the data sets in the order in which they appear in the MERGE statement. If two data sets contain the same variables, the values from the second data set replace the values from the first data set. After reading the first observation from the last data set and executing any other statements in the DATA step, SAS writes the contents of the program data vector to the new data set. Only those variables that are created or assigned values during the DATA step are set to missing.

### Execution — Step 2

SAS continues until it has read all observations from all data sets.

## Example 1: One-to-One Merging with an Equal Number of Observations

The SAS data sets Animal and Plant both contain the variable Common, and the observations are arranged by the values of Common. The following shows the Animal and the Plant 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	g	Fig

The following program merges these data sets and prints the results:

```
data combined;
    merge animal plant;
run;

proc print data=combined;
    title 'Data Set Combined';
run;
```

#### Output 23.7 Merged Data Sets That Have an Equal Number of Observations

Data Set COMBINED			
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

Each observation in the new data set contains all variables from all data sets. If two data sets contain the same variables, the values from the second data set replace the values from the first data set, as shown in observation 6.

## Example 2: One-to-One Merging with an Unequal Number of Observations

The SAS data sets Animal1 and Plant1 both contain the variable Common, and the observations are arranged by the values of Common. The Plant1 data set has fewer observations than the Animal1 data set. The following shows the Animal1 and the Plant1 input data sets:

Animal1			Plant1		
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			

```

5      e      Eagle
6      f      Frog

```

The following program merges these unequal data sets and prints the results:

```

data combined1;
    merge animal1 plant1;
run;

proc print data=combined1;
    title 'Data Set Combined1';
run;

```

*Output 23.8 Merged Data Sets That Have an Unequal Number of Observations*

Data Set COMBINED1			
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	

Note that observations 4 through 6 contain missing values for the variable Plant.

### Example 3: One-to-One Merging with Duplicate Values of Common Variables

The following example shows the undesirable results that you can obtain by using one-to-one merging with data sets that contain duplicate values of common variables. The value from the last data set that is read is the one that is written to the new data set. The variables are combined exactly as they are read from each data set. In the following example, the data sets Animal1 and Plant1 contain the variable Common, and each data set contains observations with duplicate values of Common. The following shows the Animal1 and the Plant1 input data sets:

Animal1			Plant1		
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 following program produces the data set MERGE1 data set and prints the results:

```
/* This program illustrates undesirable results. */
data merge1;
  merge animal1 plant1;
  run;

proc print data=merge1;
  title 'Data Set MERGE1';
  run;
```

**Output 23.9 Undesirable Results with Duplicate Values of Common Variables**

Data Set MERGE1			
Obs	Common	Animal1	Plant1
1	a	Ant	Apple
2	b	Ape	Banana
3	c	Bird	Coconut
4	c	Cat	Celery
5	d	Dog	Dewberry
6	e	Eagle	Eggplant

The number of observations in the new data set is six. Note that observations 2 and 3 contain undesirable values. SAS reads the second observation from data set Animal1. It then reads the second observation from data set Plant1 and replaces the values for the variables Common and Plant1. The third observation is created in the same way.

#### Example 4: One-to-One Merging with Different Values of Common Variables

The following example shows the undesirable results obtained from using the one-to-one merge to combine data sets with different values of common variables. If a variable exists in more than one data set, the value from the last data set that is read is the one that is written to the new data set even if the value is missing. Once SAS processes all observations in a data set, all subsequent observations in the new data set have missing values for the variables that are unique to that data set. In this example, the data sets Animal2 and Plant2 have different values of the Common variable. The following shows the Animal2 and the Plant2 input data sets:

Animal2			Plant2		
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

The following program produces the data set MERGE2 and prints the results:

```
/* This program illustrates undesirable results. */
data merge2;
    merge animal2 plant2;
    run;

proc print data=merge2;
    title 'Data Set MERGE2';
    run;
```

*Output 23.10 Undesirable Results with Different Values of Common Variables*

Data Set MERGE2			
Obs	Common	Animal2	Plant2
1	a	Ant	Apple
2	b	Cat	Banana
3	c	Dog	Coconut
4	e	Eagle	Eggplant
5	f		Fig

## Comments and Comparisons

The results from a one-to-one merge are similar to the results obtained from using two or more SET statements to combine observations. However, with the one-to-one merge, SAS continues processing all observations in all data sets that were named in the MERGE statement.

## Match-Merging

### Definition

Match-merging combines observations from two or more SAS data sets into a single observation in a new data set according to the values of a common variable. The number of observations in the new data set is the sum of the largest number of observations in each BY group in all data sets. To perform a match-merge, use the MERGE statement with a BY statement. Before you can perform a match-merge, all data sets must be sorted by the variables that you specify in the BY statement or they must have an index.

### Syntax

Use this form of the MERGE statement to match-merge data sets:

MERGE *data-set(s)*;  
 BY *variable(s)*;  
 where  
*data-set*  
 names at least two existing SAS data sets from which observations are read.  
*variable*  
 names each variable by which the data set is sorted or indexed. These variables are referred to as BY variables.

For a complete description of the MERGE and the BY statements, see [SAS DATA Step Statements: Reference](#).

## DATA Step Processing during Match-Merging

### Compilation phase

SAS reads the descriptor information of each data set that is named in the MERGE statement and then creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step. SAS creates the FIRST.variable and LAST.variable for each variable that is listed in the BY statement.

### Execution – Step 1

SAS looks at the first BY group in each data set that is named in the MERGE statement to determine which BY group should appear first in the new data set. The DATA step reads into the program data vector the first observation in that BY group from each data set, reading the data sets in the order in which they appear in the MERGE statement. If a data set does not have observations in that BY group, the program data vector contains missing values for the variables unique to that data set.

### Execution – Step 2

After processing the first observation from the last data set and executing other statements, SAS writes the contents of the program data vector to the new data set. SAS retains the values of all variables in the program data vector except those variables that were created by the DATA step; SAS sets those values to missing. SAS continues to merge observations until it writes all observations from the first BY group to the new data set. When SAS has read all observations in a BY group from all data sets, it sets all variables in the program data vector (except those created by SAS) to missing. SAS looks at the next BY group in each data set to determine which BY group should appear next in the new data set.

### Execution – Step 3

SAS repeats these steps until it reads all observations from all BY groups in all data sets.

## Example 1: Combining Observations Based on a Criterion

The SAS data sets Animal and Plant each contain the BY variable Common, and the observations are arranged in order of the values of the BY variable. The following shows the Animal and the Plant 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 merges the data sets according to the values of the BY variable Common, and prints the results:

```
data combined;
    merge animal plant;
    by Common;
run;

proc print data=combined;
    title 'Data Set Combined';
run;
```

*Output 23.11 Data Sets Combined by Match-Merging*

Data Set COMBINED			
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

Each observation in the new data set contains all the variables from all the data sets.

## Example 2: Match-Merge with Duplicate Values of the BY Variable

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. In the following example, the data sets Animal1 and Plant1 contain duplicate values of the BY variable Common. The following shows the Animal1 and the Plant1 input data sets:

Animal1			Plant1		
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 produces the merged data set MATCH1, and prints the results:

```

data match1;
  merge animal1 plant1;
  by Common;
run;

proc print data=match1;
  title 'Data Set MATCH1';
run;

```

**Output 23.12** Match-Merged Data Set with Duplicate BY Values

Data Set MATCH1			
Obs	Common	Animal1	Plant1
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

In observation 2 of the output, the value of the variable Plant1 is retained until all observations in the BY group are written to the new data set. Match-merging also produced duplicate values in Animal1 for observations 4 and 5.

**Note:** The MERGE statement does not produce a Cartesian product on a many-to-many match-merge. Instead, it performs a one-to-one merge while there are observations in the BY group in at least one data set. When all observations in the BY group have been read from one data set and there are still more observations in another data set, SAS performs a one-to-many merge until all observations have been read for the BY group.

### Example 3: Match-Merge with Nonmatched Observations

When SAS performs a match-merge with nonmatched observations in the input data sets, SAS retains the values of all variables in the program data vector even if the value is missing. The data sets Animal2 and Plant2 do not contain all values of the BY variable Common. The following shows the Animal2 and the Plant2 input data sets:

Animal2	Plant2
---------	--------

OBS	Common	Animal2	OBS	Common	Plant2
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

Fig

The following program produces the merged data set MATCH2, and prints the results:

```
data match2;
    merge animal2 plant2;
    by Common;
run;

proc print data=match2;
    title 'Data Set MATCH2';
run;
```

*Output 23.13 Match-Merged Data Set with Nonmatched Observations*

Data Set MATCH2				
Obs	Common	Animal2	Plant2	
1	a	Ant	Apple	
2	b		Banana	
3	c	Cat	Coconut	
4	d	Dog		
5	e	Eagle	Eggplant	
6	f		Fig	

As the output shows, all values of the variable Common are represented in the new data set, including missing values for the variables that are in one data set but not in the other.

## Updating with the UPDATE and the MODIFY Statements

### Definitions

Updating a data set refers to the process of applying changes to a master data set. To update data sets, you work with two input data sets. The data set containing the original information is the master data set, and the data set containing the new information is the transaction data set.

You can update data sets by using the UPDATE statement or the MODIFY statement:

#### UPDATE

uses observations from the transaction data set to change the values of corresponding observations from the master data set. You must use a BY statement with the UPDATE statement because all observations in the transaction data set are keyed to observations in the master data set according to the values of the BY variable.

#### MODIFY

can replace, delete, and append observations in an existing data set. Using the MODIFY statement can save disk space because it modifies data in place, without creating a copy of the data set.

The number of observations in the new data set is the sum of the number of observations in the master data set and the number of unmatched observations in the transaction data set.

For complete information about the UPDATE and the MODIFY statements, see [SAS DATA Step Statements: Reference](#).

## Syntax of the UPDATE Statement

Use this form of the UPDATE statement to update a master data set:

`UPDATE master-data-set transaction-data-set;`

`BY variable-list;`

where

`master-data-set`

names the SAS data set that is used as the master file.

`transaction-data-set`

names the SAS data set that contains the changes to be applied to the master data set.

`variable-list`

specifies the variables by which observations are matched.

If the transaction data set contains duplicate values of the BY variable, SAS applies both transactions to the observation. The last values that are copied into the program data vector are written to the new data set. If your data is in this form, use the MODIFY statement instead of the UPDATE statement to process your data.

**CAUTION! Values of the BY variable must be unique for each observation in the master data set.** If the master data set contains two observations with the same value of the BY variable, the first observation is updated and the second observation is ignored. SAS writes a warning message to the log when the DATA step executes.

For complete information about the UPDATE statement, see [SAS DATA Step Statements: Reference](#).

## Syntax of the MODIFY Statement

This form of the MODIFY statement is used in the examples that follow:

`MODIFY master-data-set;`

`BY variable-list;`

where

***master-data-set***

specifies the SAS data set that you want to modify.

***variable-list***

names each variable by which the data set is ordered.

**Note:** The MODIFY statement does not support changing the descriptor portion of a SAS data set, such as adding a variable.

For complete information, see MODIFY Statement in the [SAS DATA Step Statements: Reference](#).

## DATA Step Processing with the UPDATE Statement

### Compilation phase

- SAS reads the descriptor information of each data set that is named in the UPDATE statement and creates a program data vector that contains all the variables from all data sets as well as variables created by the DATA step.
- SAS creates the FIRST.variable and LAST.variable for each variable that is listed in the BY statement.

### Execution – Step 1

SAS looks at the first observation in each data set that is named in the UPDATE statement to determine which BY group should appear first. If the transaction BY value precedes the master BY value, SAS reads from the transaction data set only and sets the variables from the master data set to missing. If the master BY value precedes the transaction BY value, SAS reads from the master data set only and sets the unique variables from the transaction data set to missing. If the BY values in the master and transaction data sets are equal, it applies the first transaction by copying the nonmissing values into the program data vector.

### Execution – Step 2

After completing the first transaction, SAS looks at the next observation in the transaction data set. If SAS finds one with the same BY value, it applies that transaction too. The first observation then contains the new values from both transactions. If no other transactions exist for that observation, SAS writes the observation to the new data set and sets the values in the program data vector to missing. SAS repeats these steps until it has read all observations from all BY groups in both data sets.

## Updating with Nonmatched Observations, Missing Values, and New Variables

In the UPDATE statement, if an observation in the master data set does not have a corresponding observation in the transaction data set, SAS writes the observation to the new data set without modifying it. Any observation from the transaction data set that does not correspond to an observation in the master data set is written to the program data vector and becomes the basis for an observation in the new data set. The data in the program data vector can be modified by other transactions before it is written to the new data set. If a master data set observation does not need updating, the corresponding observation can be omitted from the transaction data set.

SAS does not replace existing values in the master data set with missing values if those values are coded as periods (for numeric variables) or blanks (for character variables) in the transaction data set. To replace existing values with missing values, you must either create a transaction data set in which missing values are coded with

the special missing value characters, or use the UPDATEMODE=NOMISSINGCHECK statement option.

With UPDATE, the transaction data set can contain new variables to be added to all observations in the master data set.

To view a sample program, see “[Example 3: Using UPDATE for Processing Nonmatched Observations, Missing Values, and New Variables](#)” on page 549.

## Sort Requirements for the UPDATE Statement

If you do not use an index, both the master data set and the transaction data set must be sorted by the same variable or variables that you specify in the BY statement that accompanies the UPDATE statement. The values of the BY variable should be unique for each observation in the master data set. If you use more than one BY variable, the combination of values of all BY variables should be unique for each observation in the master data set. The BY variable or variables should be ones that you never need to update.

**Note:** The MODIFY statement does not require sorted files. However, sorting the data improves efficiency.

## Using an Index with the MODIFY Statement

The MODIFY statement maintains the index. You do not have to rebuild the index like you do for the UPDATE statement.

## Choosing between UPDATE or MODIFY with BY

Using the UPDATE statement is comparable to using MODIFY with BY to apply transactions to a data set. MODIFY is a more powerful tool with several other applications, but UPDATE is still the tool of choice in some cases. The following table helps you choose whether to use UPDATE or MODIFY with BY.

**Table 23.3** MODIFY with BY versus UPDATE

Issue	MODIFY with BY	UPDATE
Disk space	saves disk space because it updates data in place	requires more disk space because it produces an updated copy of the data set
Sort and index	sorted input data sets are not required, although for good performance, it is strongly recommended that both data sets be sorted and that the master data set be indexed	requires only that both data sets be sorted
When to use	use only when you expect to process a SMALL portion of the data set	use if you expect to need to process most of the data set
Where to specify the modified data set	specify the updated data set in both the DATA and the MODIFY statements	specify the updated data set in the DATA and the UPDATE statements
Duplicate BY-values	allows duplicate BY-values in both the master and the transaction data sets	allows duplicate BY-values in the transaction data set only (If duplicates exist in the master data set, SAS issues a warning.)

Issue	MODIFY with BY	UPDATE
Scope of changes	cannot change the data set descriptor information, so changes such as adding or deleting variables, variable labels, and so on, are not valid	can make changes that require a change in the descriptor portion of a data set, such as adding new variables, and so on
Error checking	has error-checking capabilities using the <code>_IORC_</code> automatic variable and the <code>SYSRC</code> autocall macro	needs no error checking because transactions without a corresponding master record are not applied but are added to the data set
Data set integrity	data might be only partially updated due to an abnormal task termination	no data loss occurs because UPDATE works on a copy of the data

For more information about tools for combining SAS data sets, see [Table 23.2 on page 520](#).

## Primary Uses of the MODIFY Statement

The MODIFY statement has three primary uses:

- modifying observations in a single SAS data set
- modifying observations in a single SAS data set directly, either by observation number or by values in an index
- modifying observations in a master data set, based on values in a transaction data set. MODIFY with BY is similar to using the UPDATE statement

Several of the examples that follow demonstrate these uses.

## Example 1: Using UPDATE for Basic Updating

In this example, the data set Master contains original values of the variables Animal and Plant. The data set NEWPlant is a transaction data set with new values of the variable Plant. The following shows the Master and the NEWPlant input data sets:

Master				NEWPlant		
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 following program updates Master with the transactions in the data set NEWPlant, writes the results to `UPDATE_FILE`, and prints the results:

```
data update_file;
  update master newplant;
  by common;
run;
```

```
proc print data=update_file;
   title 'Data Set Update_File';
   run;
```

**Output 23.14** Master Data Set Updated by Transaction Data Set

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

Each observation in the new data set contains a new value for the variable Plant.

## Example 2: Using UPDATE with Duplicate Values of the BY Variable

If the master data set contains two observations with the same value of the BY variable, the first observation is updated and the second observation is ignored. SAS writes a warning message to the log. If the transaction data set contains duplicate values of the BY variable, SAS applies both transactions to the observation. The last values copied into the program data vector are written to the new data set. The following shows the Master1 and the DupPlant input data sets.

Master1				DupPlant		
OBS	Common	Animal1	Plant1	OBS	Common	Plant1
1	a	Ant	Apple	1	a	Apricot
2	b	Bird	Banana	2	b	Barley
3	b	Bird	Banana	3	c	Cactus
4	c	Cat	Coconut	4	d	Date
5	d	Dog	Dewberry	5	d	Dill
6	e	Eagle	Eggplant	6	e	Escarole
7	f	Frog	Fig	7	f	Fennel

The following program applies the transactions in DupPlant to Master1 and prints the results:

```
data update1;
   update master1 dupplant;
   by Common;
   run;

proc print data=update1;
   title 'Data Set Update1';
   run;
```

**Output 23.15** Updating Data Sets with Duplicate BY Values

Data Set Update1			
Obs	Common	Animal1	Plant1
1	a	Ant	Apricot
2	b	Bird	Barley
3	b	Bird	Banana
4	c	Cat	Cactus
5	d	Dog	Dill
6	e	Eagle	Escarole
7	f	Frog	Fennel

When this DATA step executes, SAS generates a warning message stating that there is more than one observation for a BY group. However, the DATA step continues to process, and the data set Update1 is created.

The resulting data set has seven observations. Observations 2 and 3 have duplicate values of the BY variable Common. However, the value of the variable Plant1 was not updated in the second occurrence of the duplicate BY value.

### Example 3: Using UPDATE for Processing Nonmatched Observations, Missing Values, and New Variables

In this example, the data set Master2 is a master data set. It contains a missing value for the variable Plant2 in the first observation, and not all of the values of the BY variable Common are included. The transaction data set NONPlant contains a new variable Mineral, a new value of the BY variable Common, and missing values for several observations. The following shows the Master2 and the NONPlant input data sets:

Master2				NONPlant			
OBS	Common	Animal2	Plant2	OBS	Common	Plant2	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 data set Master2 and prints the results:

```
data update2_file;
  update master2 nonplant;
  by Common;
run;
```

```
proc print data=update2_file;
  title 'Data Set Update2_File';
run;
```

**Output 23.16 Results of Updating with New Variables, Nonmatched Observations, and Missing Values**

Data Set Update2_File				
Obs	Common	Animal2	Plant2	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

As shown, all observations now include values for the variable Mineral. The value of Mineral is set to missing for some observations. Observations 2 and 6 in the transaction data set did not have corresponding observations in Master2, and they have become new observations. Observation 3 from the master data set was written to the new data set without change, and the value for Plant2 in observation 4 was not changed to missing. Three observations in the new data set contain updated values for the variable Plant2.

The following program uses the UPDatemode statement option in the UPDATE statement, and prints the results:

```
data update2_file;
  update master2 nonplant updatemode=nomissingcheck;
    by Common;
  run;

  proc print data=update2_file;
    title 'Data Set Update2_File - UPDatemode Option';
  run;
```

**Output 23.17 Results of Updating with the UPDatemode Option**

<b>Obs</b>	<b>Common</b>	<b>Animal2</b>	<b>Plant2</b>	<b>Mineral</b>
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

The value of Plant2 in observation 5 is set to missing because the UPDatemode=NOMISSINGCHECK option is in effect.

For detailed examples for updating data sets, see *Combining and Modifying SAS Data Sets: Examples*.

#### Example 4: Updating a Master Data Set By Adding Observations

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 PartNumber. The *master* data set, Inventory, holds a record of the warehouse's inventory. The master data set is updated to reflect changes when the warehouse receives a new shipment of items. The *transaction* data set contains the new information about the items being added to the inventory (new types of tools), as well as changes to the existing inventory items.

To begin this example, first create the Inventory master data set and the Add\_Inventory transaction data set. Use the PRINT procedure to view the data sets as tables in HTML.

```
data Inventory;
input PartNumber $ PartName $ Amount_in_Stock
      Price ReceivedDate date9. ;
format Price comma12.2 ReceivedDate mmddyy10. ;
datalines;
K89R seal    34  245.00 07jul1998
M4J7 sander  98   45.88 20jun1998
LK43 filter  121   10.99 19may1999
MN21 brace   43   27.87 10aug1999
BC85 clamp   80    9.55 16aug1999
NCF3 valve  198   24.50 20mar1999
```

```

KJ66 cutter   6   19.77 18jun1999
UYN7 rod     211  11.55 09sep1999
JD03 switch  383  13.99 09jan2000
BV1E timer   26   34.50 03aug2000
;
proc sort data=inventory; by PartNumber; run;
proc print data=inventory;
   title "Inventory Data Set Sorted By PartNumber";
run;

data add_Inventory;
input PartNumber $ PartName $ Add_New_Stock New_Price;
format New_Price comma12.2;
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
;
proc sort data=add_Inventory; by PartNumber; run;
proc print data=add_Inventory;
   title "Add_Inventory Data Set Sorted By PartNumber";
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.

**Inventory Data Set Sorted by PartNumber  
(Master)**

Obs	PartNumber	PartName	Amount_in_Stock	Price	ReceivedDate
1	BC85	clamp	80	9.55	08/16/1999
2	BV1E	timer	26	34.50	08/03/2000
3	JD03	switch	383	13.99	01/09/2000
4	K89R	seal	34	245.00	07/07/1998
5	KJ66	cutter	6	19.77	06/18/1999
6	LK43	filter	121	10.99	05/19/1999
7	M4J7	sander	98	45.88	06/20/1998
8	MN21	brace	43	27.87	08/10/1999
9	NCF3	valve	198	24.50	03/20/1999
10	UYN7	rod	211	11.55	09/09/1999

**Add\_Inventory Data Set Sorted By PartNumber  
(Transaction)**

Obs	PartNumber	PartName	Add_New_Stock	New_Price
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, Add\_Inventory, do not exist in the master data set, Inventory. Also, notice that values for the variables Amount\_in\_Stock, Price, and ReceivedDate in the transaction data set contain new values.

Now, modify the master data set based on the new information in the transaction data set:

```

data Inventory;
  modify Inventory add_Inventory; /* 1 */
    by PartNumber;
    select (_iorc_); /* 2 */
      *** The observation exists in the master data set */;
      when (%sysrc(_sok)) do; /* 3 */
        Amount_in_Stock = Amount_in_Stock + Add_New_Stock;
        ReceivedDate = today();
        replace; /* 4 */
      end;
      *** The observation does not exist in the master data set*/
      when (%sysrc(_dsenmr)) do; /* 5 */
        Amount_in_Stock=Add_New_Stock;
        ReceivedDate=today();
        Price>New_Price;
        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 "Updated 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 **%SYSPROC 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 Amount\_in\_Stock and Price. The Amount\_in\_Stock values are updated based on the values for Add\_New\_Stock in the transaction data set. The Price values are updated based on the values for New\_Price 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 3 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 do 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 Amount\_in\_Stock and Price.

**Figure 23.10 Results for the Inventory Master Data Set Sorted by PartNumber**

Updated Inventory Data Set Sorted by PartNumber					
Obs	PartNumber	PartName	Amount_in_Stock	Price	ReceivedDate
1	AA11	hammer	55	32.26	03/27/2018
2	BB22	wrench	21	17.35	03/27/2018
3	BC85	clamp	80	9.55	08/16/1999
4	BV1E	timer	56	36.50	08/03/2000
5	CC33	socket	7	22.19	03/27/2018
6	JD03	switch	383	13.99	01/09/2000
7	K89R	seal	40	247.50	07/07/1998
8	KJ66	cutter	16	24.50	06/18/1999
9	LK43	filter	121	10.99	05/19/1999
10	M4J7	sander	98	45.88	06/20/1998
11	MN21	brace	43	27.87	08/10/1999
12	NCF3	valve	198	24.50	03/20/1999
13	UYN7	rod	211	11.55	09/09/1999

■ New inventory item (new observation)  
■ Update to existing inventory item

**Note:** Using the OUTPUT or REPLACE statements 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.

For more information about the statements in this program, see the following:

- “[Error Checking When Using Indexes to Randomly Access or Update Data](#)” on page 555
- “[%SYSRC Autocall Macro](#)” in *SAS Macro Language: Reference*
- “[SELECT Statement](#)” in *SAS DATA Step Statements: Reference*
- “[Error-Checking Tools](#)” on page 555
- “[REPLACE Statement](#)” in *SAS DATA Step Statements: Reference*
- “[OUTPUT Statement](#)” in *SAS DATA Step Statements: Reference*

---

## 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 write them to the output data set. If they do not match, however, you might want SAS to 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 23.4** 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_);
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 the 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. The result is to write the observation to the output data set. 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



# Using DATA Step Component Objects

<i>Introduction to DATA Step Component Objects</i>	565
<b><i>Using the Hash Object</i></b>	<b>566</b>
Why Use the Hash Object?	566
Declaring and Instantiating a Hash Object	567
Initializing Hash Object Data Using a Constructor	567
Defining Keys and Data	568
Non-Unique Key and Data Pairs	569
Storing and Retrieving Data	570
Maintaining Key Summaries	572
Replacing and Removing Data in the Hash Object	575
Saving Hash Object Data in a Data Set	577
Comparing Hash Objects	578
Using Hash Object Attributes	578
<b><i>Using the Hash Iterator Object</i></b>	<b>579</b>
About the Hash Iterator Object	579
Declaring and Instantiating a Hash Iterator Object	579
Example: Retrieving Hash Object Data By Using the Hash Iterator	580
<b><i>Using the Java Object</i></b>	<b>582</b>
About the Java Object	582
CLASSPATH and Java Options	583
Restrictions and Requirements for Using the Java Object	584
Declaring and Instantiating a Java Object	584
Accessing Object Fields	585
Accessing Object Methods	585
Type Issues	586
Java Objects and Arrays	588
Passing Java Object Arguments	589
Java Exceptions	591
Java Standard Output	591
Java Object Examples	592

---

## Introduction to DATA Step Component Objects

SAS provides these five predefined component objects for use in a DATA step:

#### hash and hash iterator objects

enable you to quickly and efficiently store, search, and retrieve data based on lookup keys. The hash object keys and data are DATA step variables. Key and data values can be directly assigned constant values or values from a SAS data set. For information about the hash and hash iterator object language elements, see “[Dictionary of Hash and Hash Iterator Object Language Elements](#)” in *SAS Component Objects: Reference*.

#### Java object

provides a mechanism that is similar to the Java Native Interface (JNI) for instantiating Java classes and accessing fields and methods on the resultant objects. For more information, see “[Dictionary of Java Object Language Elements](#)” in *SAS Component Objects: Reference*.

#### logger and appender objects

enable you to record logging events and write these events to the appropriate destination. For more information, see “[Component Object Reference](#)” in *SAS Logging: Configuration and Programming Reference*.

The DATA step Component Interface enables you to create and manipulate these component objects using statements, attributes, operators, and methods. You use the DATA step object dot notation to access the component object's attributes and methods. For detailed information about dot notation and the DATA step objects' statements, attributes, methods, and operators, see the Dictionary of Component Language Elements in *SAS Component Objects: Reference*.

**Note:** The DATA step component object statement, attributes, methods, and operators are limited to those defined for these objects. You cannot use the SAS Component Language functionality with these predefined DATA step objects.

## Using the Hash Object

### Why Use the Hash Object?

The hash object provides an efficient, convenient mechanism for quick data storage and retrieval. The hash object stores and retrieves data based on lookup keys.

To use the DATA step Component Object Interface, follow these steps:

- 1 Declare the hash object.
- 2 Create an instance of (instantiate) the hash object.
- 3 Initialize lookup keys and data.

After you declare and instantiate a hash object, you can perform many tasks, including these:

- Store and retrieve data.
- Maintain key summaries.
- Replace and remove data.
- Compare hash objects.

- Output a data set that contains the data in the hash object.

For example, suppose you have a large data set that contains numeric lab results corresponding to a unique patient number and weight. And suppose you have a small data set that contains patient numbers (a subset of those in the large data set). You can load the large data set into a hash object using the unique patient number as the key and the weight values as the data. A single pass is made over the small data set using the patient number to look up the current patient in the hash object whose weight is over a certain value and output that data to a different data set.

Depending on the number of lookup keys and the size of the data set, the hash object lookup can be significantly faster than a standard format lookup. If you are just looking up keys, you have a lot of memory, and you want fast performance, load the large data set first. If you do not want to use a lot of memory, load the small data set first.

## Declaring and Instantiating a Hash Object

You declare a hash object using the DECLARE statement. After you declare the new hash object, use the \_NEW\_ operator to instantiate the object. For example:

```
declare hash myhash;
myhash = _new_ hash();
```

The DECLARE statement tells the compiler that the object reference MyHash is of type hash. At this point, you have declared only the object reference MyHash. It has the potential to hold a component object of type hash. You should declare the hash object only once. The \_NEW\_ operator creates an instance of the hash object and assigns it to the object reference MyHash.

There is an alternative to the two-step process of using the DECLARE statement and the \_NEW\_ operator to declare and instantiate a component object. You can use the DECLARE statement to declare and instantiate the component object in one step.

```
declare hash myhash();
```

The above statement is equivalent to the following code:

```
declare hash myhash;
myhash = _new_ hash();
```

For more information, see “[DECLARE Statement: Hash and Hash Iterator Objects](#)” in *SAS Component Objects: Reference* and the “[Hash and Hash Iterator Operator: Objects](#)” in *SAS Component Objects: Reference*.

## Initializing Hash Object Data Using a Constructor

When you create a hash object, you might want to provide initialization data. A constructor is a method that you can use to instantiate a hash object and initialize the hash object data.

The hash object constructor can have either of the following formats:

- `declare hash object_name(argument_tag-1: value-1 <, ...argument_tag-n: value-n>);`

```
■ object_name = _new_ hash(argument_tag-1: value-1
<, ...argument_tag-n: value-n>);
```

For more information, see the “[DECLARE Statement: Hash and Hash Iterator Objects](#)” in *SAS Component Objects: Reference* and the “[Hash and Hash Iterator Operator: Objects](#)” in *SAS Component Objects: Reference*.

## Defining Keys and Data

The hash object uses lookup keys to store and retrieve data. The keys and the data are DATA step variables that you use to initialize the hash object by using dot notation method calls. A key is defined by passing the key variable name to the DEFINEKEY method. Data is defined by passing the data variable name to the DEFINEDATA method. After you have defined all key and data variables, the DEFINEDONE method is called. Keys and data can consist of any number of character or numeric DATA step variables.

For example, the following code initializes a character key and a character data variable:

```
length d $20;
length k $20;

if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
end;
```

You can have multiple key and data variables, but the entire key must be unique, unless you create the hash object with the MULTIDATA:“YES” argument tag. For more information, see “[Non-Unique Key and Data Pairs](#)” on page 569.

You can store more than one data item with a particular key. For example, you could modify the previous example to store auxiliary numeric values with the character key and data. In this example, each key and each data item consists of a character value and a numeric value:

```
length d1 8;
length d2 $20;
length k1 $20;
length k2 8;

if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k1', 'k2');
  rc = h.defineData('d1', 'd2');
  rc = h.defineDone();
end;
```

For more information, see the “[DEFINEDATA Method](#)” in *SAS Component Objects: Reference*, “[DEFINEDONE Method](#)” in *SAS Component Objects: Reference*, and the “[DEFINEKEY Method](#)” in *SAS Component Objects: Reference*.

**Note:** The hash object does not assign values to key variables (for example, `h.find(key: 'abc')`), and the SAS compiler cannot detect the data variable assignments that are performed by the hash object and the hash iterator. If you

declare a key or data variable in the program, but do not assign that key or data variable an initial value, SAS issues a note stating that the variable is uninitialized. To avoid receiving these notes, you can perform one of the following actions:

- Set the NONOTES system option.
- Provide an initial assignment statement (typically to a missing value) for each key and data variable.
- Use the CALL MISSING routine with all the key and data variables as parameters. Here is an example.

```
length d $20;
length k $20;

if _N_ = 1 then do;
   declare hash h();
   rc = h.defineKey('k');
   rc = h.defineData('d');
   rc = h.defineDone();
   call missing(k, d);
end;
```

If you use a key or data variable without declaring or initializing that key or data variable outside the hash object, an error occurs.

## Non-Unique Key and Data Pairs

By default, all of the keys in a hash object are unique. This means one set of data variables exists for each key. In some situations, you might want to have duplicate keys in the hash object, that is, associate more than one set of data variables with a key.

For example, assume that the key is a patient ID and the data is a visit date. If the patient were to visit multiple times, multiple visit dates would be associated with the patient ID. When you create a hash object with the MULTIDATA:“YES” argument tag, multiple sets of the data variables are associated with the key.

If the data set contains duplicate keys, by default, the first instance is stored in the hash object and subsequent instances are ignored. To store the last instance in the hash object, use the DUPLICATE argument tag. The DUPLICATE argument tag also writes an error to the SAS log if there is a duplicate key.

However, the hash object allows storage of multiple values for each key if you use the MULTIDATA argument tag in the DECLARE statement or \_NEW\_ operator. The hash object keeps the multiple values in a list that is associated with the key. This list can be traversed and manipulated by using several methods such as HAS\_NEXT or FIND\_NEXT.

To traverse a multiple data item list, you must know the current list item. Start by calling the FIND method for a given key. The FIND method sets the current list item. Then to determine whether the key has multiple data values, call the HAS\_NEXT method. After you have determined that the key has another data value, you can retrieve that value with the FIND\_NEXT method. The FIND\_NEXT method sets the current list item to the next item in the list and sets the corresponding data variable or variables for that item.

In addition to moving forward through the list for a given key, you can loop backward through the list by using the HAS\_PREV and FIND\_PREV methods in a similar manner.

When you have a hash object that has multiple values for a single key, you can use the DO\_OVER method in an iterative DO loop to traverse through the duplicate keys. The DO\_OVER method reads the key on the first method call and continues to iterate over the duplicate key list until it reaches the end.

**Note:** The items in a multiple data item list are maintained in the order in which you insert them.

For more information about these and other methods associated with non-unique key and data pairs, see “[Dictionary of Hash and Hash Iterator Object Language Elements](#)” in *SAS Component Objects: Reference*.

## Storing and Retrieving Data

### How to Store and Retrieve Data

After you initialize the hash object's key and data variables, you can store data in the hash object using the ADD method, or you can use the *dataset* argument tag to load a data set into the hash object. If you use the *dataset* argument tag, and if the data set contains more than one observation with the same value of the key, by default, SAS keeps the first observation in the hash table and ignores subsequent observations. To store the last instance in the hash object or to send an error to the log if there is a duplicate key, use the DUPLICATE argument tag. To allow duplicate values for each key, use the MULTIDATA argument tag.

You can then use the FIND method to search and retrieve data from the hash object if one data value exists for each key. Use the FIND\_NEXT and FIND\_PREV methods to search and retrieve data if multiple data items exist for each key.

For more information, see “[ADD Method](#)” in *SAS Component Objects: Reference*, “[FIND Method](#)” in *SAS Component Objects: Reference*, “[FIND\\_NEXT Method](#)” in *SAS Component Objects: Reference*, and the “[FIND\\_PREV Method](#)” in *SAS Component Objects: Reference*.

You can consolidate a FIND method and ADD method using the REF method. In the following example, you can reduce the amount of code from this:

```
rc = h.find();
if (rc != 0) then
  rc = h.add();
```

to a single method call:

```
rc = h.ref();
```

For more information, see the “[REF Method](#)” in *SAS Component Objects: Reference*.

**Note:** You can also use the hash iterator object to retrieve the hash object data, one data item at a time, in forward and reverse order. For more information, see “[Using the Hash Iterator Object](#)” on page 579.

## Example 1: Using the ADD and FIND Methods to Store and Retrieve Data

The following example uses the ADD method to store the data in the hash object and associate the data with the key. The FIND method is then used to retrieve the data that is associated with the key value **Homer**.

```

data _null_;
length d $20;
length k $20;

/* Declare the hash object and key and data variables */
if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
end;

/* Define constant value for key and data */
k = 'Homer';
d = 'Odyssey';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Define constant value for key and data */
k = 'Joyce';
d = 'Ulysses';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

k = 'Homer';
/* Use the FIND method to retrieve the data associated with 'Homer' key */
rc = h.find();
if (rc = 0) then
  put d=;
else
  put 'Key Homer not found.';
run;

```

The FIND method assigns the data value **Odyssey**, which is associated with the key value **Homer**, to the variable D.

## Example 2: Loading a Data Set and Using the FIND Method to Retrieve Data

Assume the data set Small contains two numeric variables K (key) and S (data) and another data set, LARGE, contains a corresponding key variable K. The following code loads the Small data set into the hash object, and then searches the hash object for key matches on the variable K from the LARGE data set.

```

data match;
length k 8;
length s 8;
if _N_ = 1 then do;
  /* load SMALL data set into the hash object */
  declare hash h(dataset: "work.small");
  /* define SMALL data set variable K as key and S as value */
  h.defineKey('k');
  h.defineData('s');
  h.defineDone();
  /* avoid uninitialized variable notes */
  call missing(k, s);
end;

/* use the SET statement to iterate over the LARGE data set using */
/* keys in the LARGE data set to match keys in the hash object */
set large;
rc = h.find();
if (rc = 0) then output;
run;

```

The *dataset* argument tag specifies the Small data set whose keys and data are read and loaded by the hash object during the DEFINEDONE method. The FIND method is then used to retrieve the data.

## Maintaining Key Summaries

You can maintain a summary count for a hash object key by using the SUMINC argument tag when you declare the hash object. The tag value is a string expression that resolves to the name of a numeric DATA step variable: the SUMINC variable.

This SUMINC tag instructs the hash object to allocate internal storage for maintaining a summary value for each key.

The summary value of a hash key is initialized to the value of the SUMINC variable whenever the ADD or REPLACE method is used.

The summary value of a hash key is incremented by the value of the SUMINC variable whenever the FIND, CHECK, or REF method is used.

Note that the SUMINC variable can be negative, positive, or zero valued. The variable does not need to be an integer. The SUMINC value for a key is zero by default.

In the following example, the initial ADD method sets the summary count for K=99 to 1 before the ADD. Then each time a new COUNT value is given, the following FIND method adds the value to the key summary. In this example, one data value exists for each key. The SUM method retrieves the current value of the key summary and the value is stored in the DATA step variable TOTAL. If multiple items exist for each key, the SUMDUP method retrieves the current value of the key summary.

```

data _null_;
length k count 8;
length total 8;
dcl hash myhash(suminc: 'count');

```

```

myhash.defineKey('k');
myhash.defineDone();

k = 99;
count = 1;
myhash.add();

/* COUNT is given the value 2.5 and the */
/* FIND sets the summary to 3.5*/
count = 2.5;
myhash.find();

/* The COUNT of 3 is added to the FIND and */
/* sets the summary to 6.5. */
count = 3;
myhash.find();

/* The COUNT of -1 sets the summary to 5.5. */
count = -1;
myhash.find();

/* The SUM method gives the current value of */
/* the key summary to the variable TOTAL. */
myhash.sum(sum: total);

/* The PUT statement prints total=5.5 in the log. */
put total=;
run;

```

In this example, a summary is maintained for each key value K=99 and K=100:

```

k = 99;
count = 1;
myhash.add();
/* key=99 summary is now 1 */

k = 100;
myhash.add();
/* key=100 summary is now 1 */

k = 99;
myhash.find();
/* key=99 summary is now 2 */

count = 2;
myhash.find();
/* key=99 summary is now 4 */

k = 100;
myhash.find();
/* key=100 summary is now 3 */

myhash.sum(sum: total);
put 'total for key 100 = 'total;

```

```

k = 99;

myhash.sum(sum:total);
put 'total for key 99 = ' total;

```

The first PUT statement prints the summary for K=100:

```
total for key 100 = 3
```

And the second PUT statement prints the summary for K=99:

```
total for key 99 = 4
```

You can use key summaries in conjunction with the *dataset* argument tag. As the data set is read into the hash object using the DEFINEDONE method, all key summaries are set to the SUMINC value. And, all subsequent FIND, CHECK, or ADD methods change the corresponding key summaries.

```
declare hash myhash(suminc: "keycount", dataset: "work.mydata");
```

You can use key summaries for counting the number of occurrences of given keys. In the following example, the data set MyData is loaded into a hash object and uses key summaries to keep count of the number of occurrences for each key in the data set Keys. (The SUMINC variable is not set to a value, so the default initial value of zero is used.)

```

data mydata;
  input key;
  datalines;
1
2
3
4
5
;
run;
```

```

data keys;
  input key;
  datalines;
1
2
1
3
5
2
3
2
4
1
5
1
;
run;
```

```

data count;
length total key 8;
keep key total;
```

```

declare hash myhash(suminc: "count", dataset:"mydata");
myhash.defineKey('key');
myhash.defineDone();
count = 1;

do while (not done);
  set keys end=done;
  rc = myhash.find();
end;

done = 0;
do while (not done);
  set mydata end=done;
  rc = myhash.sum(sum: total);
  output;
end;
stop;
run;

```

Here is the output for the resulting data set.

*Output 24.1 Key Summary Output*

The SAS System		
Obs	total	key
1	4	1
2	3	2
3	2	3
4	1	4
5	2	5

**Note:** The KEYSUM constructor in the DECLARE statement or \_NEW\_ operator declares a variable that tracks the key summary for all keys. The KEYSUM variable is part of the output data set and works when one or more data items exist for a key.

For more information, see the “SUM Method” in *SAS Component Objects: Reference* and the “SUMDUP Method” in *SAS Component Objects: Reference*.

## Replacing and Removing Data in the Hash Object

You can remove or replace data that is stored in the hash object using any of the following methods:

- Use the REMOVE method to remove all data items.
- Use the REPLACE method to replace all data items.
- Use the REMOVEDUP method to remove only the current data item.
- Use the REPLACEDUP method to replace only the current data item.

In the following example, the REPLACE method replaces the data `odyssey` with `Iliad`, and the REMOVE method deletes the entire data entry associated with the `Joyce` key from the hash object.

```

data _null_;
length d $20;
length k $20;

/* Declare the hash object and key and data variables */
if _N_ = 1 then do;
  declare hash h();
  rc = h.defineKey('k');
  rc = h.defineData('d');
  rc = h.defineDone();
end;

/* Define constant value for key and data */
k = 'Joyce';
d = 'Ulysses';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Define constant value for key and data */
k = 'Homer';
d = 'Odyssey';
/* Use the ADD method to add the key and data to the hash object */
rc = h.add();
if (rc ne 0) then
  put 'Add failed.';

/* Use the REPLACE method to replace 'Odyssey' with 'Iliad' */
k = 'Homer';
d = 'Iliad';
rc = h.replace();
if (rc = 0) then
  put d=;
else
  put 'Replace not successful.';

/* Use the REMOVE method to remove the 'Joyce' key and data */
k = 'Joyce';
rc = h.remove();
if (rc = 0) then
  put k 'removed from hash object';
else
  put 'Deletion not successful.';

run;

```

The following lines are written to the SAS log.

```

d=Iliad
Joyce removed from hash object

```

**Note:** If an associated hash iterator is pointing to the key, the REMOVE method does not remove the key or data from the hash object. An error message is issued to the log.

For more information, see the “REMOVE Method” in *SAS Component Objects: Reference*, “REMOVEDUP Method” in *SAS Component Objects: Reference*, “REPLACE Method” in *SAS Component Objects: Reference*, and the “REPLACEDUP Method” in *SAS Component Objects: Reference*.

## Saving Hash Object Data in a Data Set

You can create a data set that contains the data in a specified hash object by using the OUTPUT method. In the following example, two keys and data are added to the hash object and then output to the Work.Out data set.

```
options pageno=1 nodate;

data test;
length d1 8;
length d2 $20;
length k1 $20;
length k2 8;

/* Declare the hash object and two key and data variables */
if _N_ = 1 then do;
    declare hash h();
    rc = h.defineKey('k1', 'k2');
    rc = h.defineData('d1', 'd2');
    rc = h.defineDone();
end;

/* Define constant value for key and data */
k1 = 'Joyce';
k2 = 1001;
d1 = 3;
d2 = 'Ulysses';
rc = h.add();

/* Define constant value for key and data */
k1 = 'Homer';
k2 = 1002;
d1 = 5;
d2 = 'Odyssey';
rc = h.add();

/* Use the OUTPUT method to save the hash object data to the OUT data set */
rc = h.output(dataset: "work.out");
run;

proc print data=work.out;
run;
```

The following output shows the report that PROC PRINT generates.

**Output 24.2** Data Set Created from the Hash Object

The SAS System		
Obs	d1	d2
1	5	Odyssey
2	3	Ulysses

Note that the hash object keys are not automatically stored as part of the output data set. The keys can be defined as data items by using the DEFINEDATA method to be included in the output data set. In addition, if no data items are defined by using the DEFINEDATA method, the keys are written to the data set specified in the OUTPUT method. In the previous example, the DEFINEDATA method would be written this way:

```
rc = h.defineData('k1', 'k2', 'd1', 'd2');
```

For more information, see the “[OUTPUT Method](#)” in [SAS Component Objects: Reference](#).

## Comparing Hash Objects

You can compare one hash object to another by using the EQUALS method. In the following example, two hash objects are being compared. Note that the EQUALS method has two argument tags. The HASH argument tag is the name of the second hash object. The RESULTS argument tag is a numeric variable name that holds the result of the comparison (1 if equal and zero if not equal).

```
length eq k 8;

declare hash myhash1();
myhash1.defineKey('k');
myhash1.defineDone();

declare hash myhash2();
myhash2.defineKey('k');
myhash2.defineDone();

rc = myhash1.equals(hash: 'myhash2', result: eq);
```

For more information, see the “[EQUALS Method](#)” in [SAS Component Objects: Reference](#).

## Using Hash Object Attributes

You can use the DATA Step Component Interface to retrieve information from a hash object using an attribute. Use the following syntax for an attribute:

```
attribute_value=obj.attribute_name;
```

There are two attributes available to use with hash objects. NUM\_ITEMS returns the number of items in a hash object and ITEM\_SIZE returns the size (in bytes) of an item. The following example retrieves the number of items in a hash object:

```
n = myhash.num_items;
```

The following example retrieves the size of an item in a hash object:

```
s = myhash.item_size;
```

You can obtain an idea of how much memory the hash object is using with the ITEM\_SIZE and NUM\_ITEMS attributes. The ITEM\_SIZE attribute does not reflect the initial overhead that the hash object requires, nor does it take into account any necessary internal alignments. Therefore, the use of ITEM\_SIZE does not provide exact memory usage, but it gives a good approximation.

For more information, see the “[NUM\\_ITEMS Attribute](#)” in *SAS Component Objects: Reference* and the “[ITEM\\_SIZE Attribute](#)” in *SAS Component Objects: Reference*.

## Using the Hash Iterator Object

### About the Hash Iterator Object

Use the hash iterator object to store and search data based on lookup keys. The hash iterator object enables you to retrieve the hash object data in either forward or reverse key order.

### Declaring and Instantiating a Hash Iterator Object

You declare a hash iterator object by using the DECLARE statement. After you declare the new hash iterator object, use the \_NEW\_ operator to instantiate the object. Use the hash object name as an argument tag. For example:

```
declare hiter myiter;
myiter = _new_ hiter('h');
```

The DECLARE statement tells the compiler that the object reference MyIter is of type hash iterator. At this point, you have declared only the object reference MyIter. It has the potential to hold a component object of type hash iterator. You should declare the hash iterator object only once. The \_NEW\_ operator creates an instance of the hash iterator object and assigns it to the object reference MyIter. The hash object, H, is passed as a constructor argument. The hash object, not the hash object variable, is specifically assigned to the hash iterator.

As an alternative to the two-step process of using the DECLARE statement and the \_NEW\_ operator to declare and instantiate a component object, you can declare and instantiate a hash iterator object in one step by using the DECLARE statement as a constructor method. The syntax is as follows:

```
declare hiter object_name(hash_object_name);
```

In the above example, the hash object name must be enclosed in single or double quotation marks.

For example:

```
declare hiter myiter('h');
```

The previous statement is equivalent to these:

```
declare hiter myiter;
myiter = _new_ hiter('h');
```

**Note:** You must declare and instantiate a hash object before you create a hash iterator object. For more information, see “[Declaring and Instantiating a Hash Object](#)” on page 567.

For example:

```
if _N_ = 1 then do;
  length key $10;
  declare hash myhash(dataset:"work.x", ordered: 'yes');
  declare hiter myiter('myhash');
  myhash.defineKey('key');
  myhash.defineDone();
end;
```

This code creates an instance of a hash iterator object with the variable name MyIter. The hash object, MyHash, is passed as the constructor argument. Because the hash object was created with the ORDERED argument tag set to 'yes', the data is returned in ascending key-value order.

For more information about the DECLARE statement and the \_NEW\_ operator, see the [SAS DATA Step Statements: Reference](#).

## Example: Retrieving Hash Object Data By Using the Hash Iterator

Using the data set ASTRO that contains astronomical data, the following code creates the data set that contains Messier objects (OBJ) whose right-ascension (RA) values are greater than 12. The FIRST and NEXT methods are used to retrieve the data in ascending order. For more information about the FIRST and NEXT methods, see [SAS Component Objects: Reference](#).

```
data astro;
  input obj $1-4 ra $6-12 dec $14-19;
  datalines;
M31 00 42.7 +41 16
M71 19 53.8 +18 47
M51 13 29.9 +47 12
M98 12 13.8 +14 54
M13 16 41.7 +36 28
M39 21 32.2 +48 26
M81 09 55.6 +69 04
M100 12 22.9 +15 49
M41 06 46.0 -20 44
M44 08 40.1 +19 59
M10 16 57.1 -04 06
M57 18 53.6 +33 02
```

```

M3 13 42.2 +28 23
M22 18 36.4 -23 54
M23 17 56.8 -19 01
M49 12 29.8 +08 00
M68 12 39.5 -26 45
M17 18 20.8 -16 11
M14 17 37.6 -03 15
M29 20 23.9 +38 32
M34 02 42.0 +42 47
M82 09 55.8 +69 41
M59 12 42.0 +11 39
M74 01 36.7 +15 47
M25 18 31.6 -19 15
;
run;

data out;
if _N_ = 1 then do;
length obj $10;
length ra $10;
length dec $10;
/* Read ASTRO data set and store in asc order in hash obj */
declare hash h(dataset:"work.astro", ordered: 'yes');
/* Define variables RA and OBJ as key and data for hash object */
declare hiter iter('h');
h.defineKey('ra');
h.defineData('ra', 'obj');
h.defineDone();
/* Avoid uninitialized variable notes */
call missing(obj, ra, dec);
end;
/* Retrieve RA values in ascending order */
rc = iter.first();
do while (rc = 0);
/* Find hash object keys greater than 12 and output data */
if ra GE '12' then
output;
rc = iter.next();
end;
run;

proc print data=work.out;
var ra obj;
title 'Messier Objects Greater than 12 Sorted by Right Ascension Values';
run;

```

The following output shows the report that PROC PRINT generates.

**Messier Objects Greater than 12 Sorted by Right Ascension Values**

Obs	ra	obj
1	12 13.8	M98
2	12 22.9	M100
3	12 29.8	M49
4	12 39.5	M68
5	12 42.0	M59
6	13 29.9	M51
7	13 42.2	M3
8	16 41.7	M13
9	16 57.1	M10
10	17 37.6	M14
11	17 56.8	M23
12	18 20.8	M17
13	18 31.6	M25
14	18 36.4	M22
15	18 53.6	M57
16	19 53.8	M71
17	20 23.9	M29
18	21 32.2	M39

---

## Using the Java Object

---

### About the Java Object

The Java object provides a mechanism that is similar to the Java Native Interface (JNI) for instantiating Java classes and accessing fields and methods on the resultant objects. You can create hybrid applications that contain both Java and DATA step code.

## CLASSPATH and Java Options

In previous versions of SAS, Java classes were found using the JREOPTIONS system option.

In SAS 9.2 and later releases, you must set the CLASSPATH environment variable so that the Java object can find your Java classes. The Java object represents an instance of a Java class that is found in the current Java classpath. Any class that you use must appear in the classpath. If the class is in a .jar file, then the .jar filename must appear in the classpath.

How you set the CLASSPATH environment variable depends on your operating environment. For most operating systems, you can set the CLASSPATH environment variable either locally (for use only in your SAS session) or globally. Table 24.1 on page 583 shows methods and examples for different operating environments. For more information, see the SAS documentation for your operating environment.

**Table 24.1** Setting the CLASSPATH Environment Variable in Different Operating Environments

Operating System	Method	Example
<b>Windows</b>		
Globally	Windows System Environment Variable in Control Panel	<b>Control Panel</b> $\Rightarrow$ <b>System and Security</b> $\Rightarrow$ <b>System</b> $\Rightarrow$ <b>Advanced system settings</b> $\Rightarrow$ <b>Advanced tab</b> $\Rightarrow$ <b>Environment Variables button</b> $\Rightarrow$ <b>System variables</b> (Windows 10)
	SAS configuration file	set classpath c:\HelloWorld.jar
Locally	SAS command line	-set classpath c:\HelloWorld.jar
<b>UNIX</b>		
Globally	SAS configuration file	set CLASSPATH ~/HelloWorld.jar
Locally	EXPORT command*	export CLASSPATH=~/HelloWorld.jar
<b>z/OS</b>		
Globally	TKMSENV data set	set TKJNI_OPT_CLASSPATH=/u/userid/java:/u/userid/java/test.jar: asis
Locally	Not available	
<b>VMS</b>		

Operating System	Method	Example
Globally	Command line**  <i>detach_template.com</i> script that is generated in sas \$root:[misc.base] at installation	\$ define java\$classpath disk:[subdir] abc.jar, disk:[subdir2]def.jar  define java\$classpath disk:[subdir] abc.jar, disk:[subdir2]def.jar
Locally	Not available	

\* The syntax depends on the shell.

\*\* The command line should be defined before you invoke SAS so that the process that the JVM actually runs in gets the definition as well.

## Restrictions and Requirements for Using the Java Object

The following restrictions and requirements apply when using the Java object:

- The Java object is designed to call Java methods from SAS. The Java object is not intended to extend the SAS library of functions. Calling PROC FCMP functions is much more efficient for fast in-process extensions to the DATA step, especially when large data sets are involved. Using the Java object to perform this type of processing with large data sets takes significantly more time.
- The only Java Runtime Environments (JREs) that are supported by SAS are those that are explicitly required during the installation of the SAS software.
- The only Java options that are supported by SAS are those that are set when SAS is installed.
- Ensure that your Java application runs correctly before using it with the Java object.
- The use of a percent character (%) in the first byte of text output by Java to the SAS log is reserved by SAS. If you need to output a % in the first byte of a Java text line, it must be escaped with another percent immediately next to it (%%).

## Declaring and Instantiating a Java Object

You declare a Java object by using the DECLARE statement. After you declare the new Java object, use the \_NEW\_ operator to instantiate the object, using the Java object name as an argument tag.

```
declare javaobj j;
j = _new_ javaobj("somejavaclass");
```

In this example, the DECLARE statement tells the compiler that the object reference J is of type Java. That is, the instance of the Java object is stored in the variable J. At this point, you have declared only the object reference J. It has the potential to hold a component object of type Java. You should declare the Java object only

once. The `_NEW_` operator creates an instance of the Java object and assigns it to the object reference J. The Java class name, `SOMEJAVACLASS`, is passed as a constructor argument, which is the first-and-only argument that is required for the Java object constructor. All other arguments are constructor arguments to the Java class itself.

As an alternative to the two-step process of using the `DECLARE` statement and the `_NEW_` operator to declare and instantiate a Java object, you can declare and instantiate a Java object in one step by using the `DECLARE` statement as a constructor method. The syntax is as follows:

**DECLARE JAVAOBJ***object-name*(“`java-class`”, <argument-1, ... argument-n>);

For more information, see the “[DECLARE Statement: Java Object](#)” in *SAS Component Objects: Reference* and the “[\\_NEW\\_ Operator: Java Object](#)” in *SAS Component Objects: Reference*.

## Accessing Object Fields

Once you instantiate a Java object, you can access and modify its public and class fields in a `DATA` step through method calls on the Java object. Public fields are non-static and declared as public in the Java class. Class fields are static and accessed from Java classes.

Method calls to access object fields have one of these forms, depending on whether you are accessing non-static or static fields:

**GET***typeFIELD*(“`field-name`”, `value`);  
**GETSTATIC***typeFIELD*(“`field-name`”, `value`);

Method calls to modify object fields have one of these forms, depending on whether you access static or non-static fields:

**SET***typeFIELD*(“`field-name`”, `value`);  
**SETSTATIC***typeFIELD*(“`field-name`”, `value`);

**Note:** The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see “[Type Issues](#)” on page 586.

The *field-name* argument specifies the type for the Java field, and *value* specifies the value that is returned or set by the method.

For more information and examples, see “[Dictionary of Java Object Language Elements](#)” in *SAS Component Objects: Reference*.

## Accessing Object Methods

Once you instantiate a Java object, you can access its public and class methods in a `DATA` step through method calls on the Java object. Public methods are non-static and declared as public in the Java class. Class methods are static and accessed from Java classes.

Method calls to access Java methods have one of these forms, depending on whether you are accessing non-static or static methods:

*object.CALL**typeMETHOD* (“`method-name`”, <`method-argument-1` ..., `method-argument-n`>,

```
<return value>);  
object.CALLSTATICtypeMETHOD ("method-name",  
<method-argument-1 ... , method-argument-n>, <return value>);
```

**Note:** The *type* argument represents a Java data type. For more information about how Java data types relate to SAS data types, see “[Type Issues](#)” on page 586.

For more information and examples, see “[Dictionary of Java Object Language Elements](#)” in *SAS Component Objects: Reference*.

## Type Issues

The Java type set is a superset of the SAS data types. Java has data types such as BYTE, SHORT, and CHAR in addition to the standard numeric and character values. SAS has only two data types: numeric and character.

The following table describes how Java data types are mapped to SAS data types when using the Java object method calls.

**Table 24.2** How Java Data Types Map to SAS Data Types

Java Data Type	SAS Data Type
BOOLEAN	numeric
BYTE	numeric
CHAR	numeric
DOUBLE	numeric
FLOAT	numeric
INT	numeric
LONG	numeric
SHORT	numeric
STRING	character*

\* Java string data types are mapped to SAS character data types as UTF-8 strings.

Other than STRING, it is not possible to return objects from Java classes to the DATA step. However, it is possible to pass objects to Java methods. For more information, see “[Passing Java Object Arguments](#)” on page 589.

Some Java methods that return objects can be handled by creating wrapper classes to convert the object values. In the following example, the Java hash table returns object values. However, you can still use the hash table from the DATA step by creating simple Java wrapper classes to handle the type conversions. Then you can access the `dhash` and `shash` classes from the DATA step.

```
/* Java code */  
import java.util.*;
```

```

public class dhash
{
    private Hashtable table;

    public dhash()
    {
        table = new Hashtable ();
    }

    public void put(double key, double value)
    {
        table.put(new Double(key), new Double(value));
    }

    public double get(double key)
    {
        Double ret = table.get(new Double(key));
        return ret.doubleValue();
    }
}

import java.util.*;

public class shash
{
    private Hashtable table;

    public shash()
    {
        table = new Hashtable ();
    }

    public void put(double key, String value)
    {
        table.put(new Double(key), value);
    }

    public String get(double key)
    {
        return table.get(new Double(key));
    }
}

/* DATA step code */
data _null_;
dcl javaobj sh('shash');
dcl javaobj dh('dhash');
length d 8;
length s $20;

do i = 1 to 10;
    dh.callvoidmethod('vput', i, i * 2);
end;

do i = 1 to 10;
    sh.callvoidmethod('put', i, 'abc' || left(trim(i)));
end;

```

```

do i = 1 to 10;
    dh.calldoublemethod('get', i, d);
    sh.callstringmethod('get', i, s);
    put d= s=;
end;
run;

```

The following lines are written to the SAS log:

```

d=2 s=abc1
d=4 s=abc2
d=6 s=abc3
d=8 s=abc4
d=10 s=abc5
d=12 s=abc6
d=14 s=abc7
d=16 s=abc8
d=18 s=abc9
d=20 s=abc10

```

## Java Objects and Arrays

You can pass DATA step arrays to Java objects.

In the following example, the arrays **d** and **s** are passed to the Java object **j**.

```

/* Java code
*/
import java.util.*;
import java.lang.*;
class jtest
{
    public void dbl(double args[])
    {
        for(int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }

    public void str(String args[])
    {
        for(int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}

/* DATA Step code */
data _null_;
dcl javaobj j("jtest");
array s{3} $20 ("abc", "def", "ghi");
array d{10} (1:10);
j.callVoidMethod("dbl", d);
j.callVoidMethod("str", s);
run;

```

The following lines are written to the SAS log:

1.0

```

2.0
3.0
4.0
5.0
6.0
7.0
8.0
9.0
10.0
abc
def
ghi

```

Only one-dimensional array parameters are supported. However, it is possible to pass multidimensional array arguments by taking advantage of the fact that the arrays are passed in row-major order. You must handle the dimensional indexing manually in the Java code. That is, you must declare a one-dimensional array parameter and index to the subarrays accordingly.

## Passing Java Object Arguments

While it is not possible to return objects from Java classes to the DATA step, it is possible to pass objects, as well as strings, to Java class methods.

For example, suppose you have the following wrapper classes for `java/util/Vector` and its iterator:

```

/* Java code */
import java.util.*;

class mVector extends Vector
{
    public mVector()
    {
        super();
    }

    public mVector(double d)
    {
        super((int)d);
    }

    public void addElement(String s)
    {
        addElement((Object)s);
    }
}

import java.util.*;
public class mIterator
{
    protected mVector m_v;
    protected Iterator iter;

    public mIterator(mVector v)

```

```

{
    m_v = v;
    iter = v.iterator();
}

public boolean hasNext()
{
    return iter.hasNext();
}

public String next()
{
    String ret = null;
    ret = (String)iter.next();
    return ret;
}
}

```

These wrapper classes are useful for performing type conversions (for example, the `mVector` constructor takes a DOUBLE argument). Overloading the constructor is necessary because `java/util/Vector`'s constructor takes an integer value, but the DATA step has no integer type.

The following DATA step program uses these classes. The program creates and fills a vector, passes the vector to the iterator's constructor, and then lists all the values in the vector. Note that you must create the iterator after the vector is filled. The iterator keeps a copy of the vector's modification count at the time of creation, and this count must stay in synchronization with the vector's current modification count. The code would throw an exception if the iterator were created before the vector was filled.

```

/* DATA step code */
data _null_;
length b 8;
length val $200;
dcl javaobj v("mVector");

v.callVoidMethod("addElement", "abc");
v.callVoidMethod("addElement", "def");
v.callVoidMethod("addElement", "ghi");
dcl javaobj iter("mIterator", v);

iter.callBooleanMethod("hasNext", b);
do while(b);
    iter.callStringMethod("next", val);
    put val=;
    iter.callBooleanMethod("hasNext", b);
end;

m.delete();
v.delete();
iter.delete();
run;

```

The following lines are written to the SAS log:

```

val=abc
val=def
val=ghi

```

One current limitation to passing objects is that the JNI method lookup routine does not perform a full class lookup based on a given signature. This means that you could not change the `mIterator` constructor to take a `Vector` as shown in the following code:

```
/* Java code */
public mIterator(Vector v)
{
    m_v = v;
    iter = v.iterator();
}
```

Even though `mVector` is a subclass of `Vector`, the method lookup routine cannot find the constructor. Currently, the only solution is to manage the types in Java by adding new methods or by creating wrapper classes.

## Java Exceptions

Java exceptions are handled through the `EXCEPTIONCHECK`, `EXCEPTIONCLEAR`, and `EXCEPTIONDESCRIBE` methods.

The `EXCEPTIONCHECK` method is used to determine whether an exception occurred during a method call. If you call a method that can throw an exception, it is strongly recommended that you check for an exception after the call. If an exception is thrown, you should take appropriate action and then clear the exception by using the `EXCEPTIONCLEAR` method.

The `EXCEPTIONDESCRIBE` method is used to turn exception debug logging on or off. If exception debug logging is on, exception information is printed to the JVM standard output. By default, JVM standard output is redirected to the SAS log. Exception debugging is off by default.

For more information, see the “[“EXCEPTIONCHECK Method” in SAS Component Objects: Reference](#)”, “[“EXCEPTIONCLEAR Method” in SAS Component Objects: Reference](#)”, and the “[“EXCEPTIONDESCRIBE Method” in SAS Component Objects: Reference](#)”.

## Java Standard Output

Output from statements in Java that are directed to standard output such as the following are sent to the SAS log by default.

```
System.out.println("hello");
```

The Java output that is directed to the SAS log is flushed when the DATA step ends. This flushing causes the Java output to appear after any output that was generated while the DATA step was running. Use the `FLUSHJAVAOUTPUT` method to synchronize the output so that it appears in the order of execution.

---

## Java Object Examples

### Example 1: Calling a Simple Java Method

This Java class creates a simple method that sums three numbers.

```
/* Java code */
class MyClass
{
    double compute(double x, double y, double z)
    {
        return (x + y + z);
    }
}

/* DATA step code */
data _null_;
dcl javaobj j("MyClass");

rc = j.callDoubleMethod("compute", 1, 2, 3, r);

put rc= r=;
run;
```

The following line is written to the SAS log:

```
rc=0 rc=6
```

### Example 2: Creating a User Interface

In addition to providing a Java component access mechanism, you can use the Java object to create a simple Java user interface.

This Java class creates a simple user interface with several buttons. The user interface also maintains a queue of values that represent the sequence of button choices that are entered by a user.

```
/* Java code */
import java.awt.*;
import java.util.*;
import java.awt.event.*;

class colorsUI extends Frame
{
    private Button red;
    private Button blue;
    private Button green;
    private Button quit;
    private Vector list;
    private boolean d;
    private colorsButtonListener cbl;

    public colorsUI()
    {
```

```

d = false;
list = new Vector();
cbl = new colorsButtonListener();

setBackground(Color.lightGray);
setSize(320,100);
setTitle("New Frame");
setVisible(true);
setLayout(new FlowLayout(FlowLayout.CENTER, 10, 15));
addWindowListener(new colorsUIListener());

red = new Button("Red");
red.setBackground(Color.red);
red.addActionListener(cbl);

blue = new Button("Blue");
blue.setBackground(Color.blue);
blue.addActionListener(cbl);

green = new Button("Green");
green.setBackground(Color.green);
green.addActionListener(cbl);

quit = new Button("Quit");
quit.setBackground(Color.yellow);
quit.addActionListener(cbl);

this.add(red);
this.add(blue);
this.add(green);
this.add(quit);

show();
}

public synchronized void enqueue(Object o)
{
    synchronized(list)
    {
        list.addElement(o);
        notify();
    }
}

public synchronized Object dequeue()
{
    try
    {
        while(list.isEmpty())
            wait();

        if (d)
            return null;

        synchronized(list)
        {

```

```

        Object ret = list.elementAt(0);
        list.removeElementAt(0);
        return ret;
    }
}
catch(Exception e)
{
    return null;
}
}

public String getNext()
{
    return (String)dequeue();
}

public boolean done()
{
    return d;
}

class colorsButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        Button b;
        String l;
        b = (Button)e.getSource();
        l = b.getLabel();
        if ( l.equals("Quit") )
        {
            d = true;
            hide();
            l = "";
        }
        enqueue(l);
    }
}

class colorsUIListener extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        Window w;
        w = e.getWindow();
        d = true;
        enqueue("");
        w.hide();
    }
}

public static void main(String s[])
{
    colorsUI cui;
    cui = new colorsUI();
}

```

```

}

/* DATA step code */
data colors;
  length s $10;
  length done 8;
  drop done;

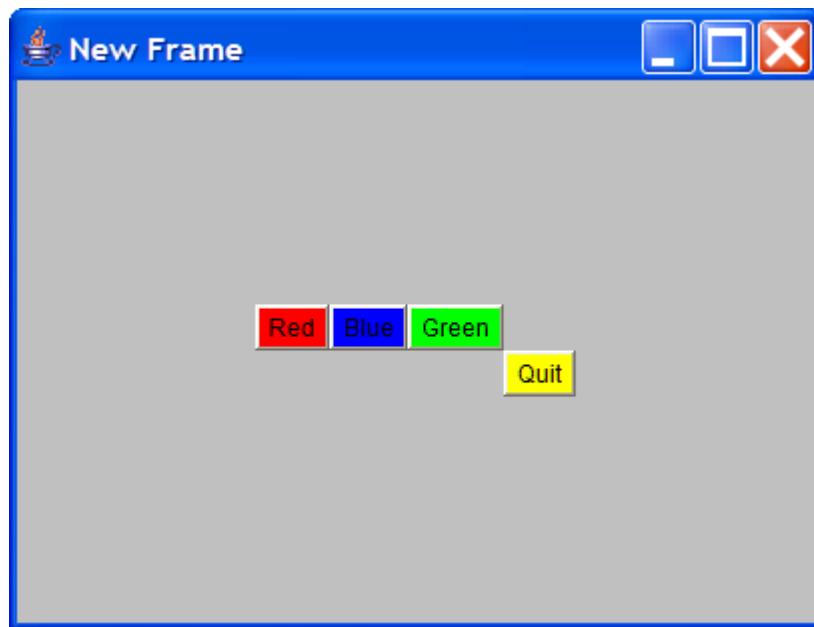
  if (_n_ = 1) then do;
    /* Declare and instantiate colors object (from colorsUI.class) */
    dcl javaobj j("colorsUI");
  end;

  /*
   * colorsUI.class will display a simple UI and maintain a
   * queue to hold color choices.
   */

  /* Loop until user hits quit button */
do while (1);
  j.callBooleanMethod("done", done);
  if (done) then
    leave;
  else do;
    /* Get next color back from queue */
    j.callStringMethod("getNext", s);
    if s ne "" then
      output;
    end;
  end;
run;
proc print data=colors;
run;
quit;

```

In the DATA step code, the `colorsUI` class is instantiated and the user interface is displayed. You enter a loop that is terminated when you click **Quit**. This action is communicated to the DATA step through the `Done` variable. While looping, the DATA step retrieves the values from the Java class's queue and writes the values successively to the output data set.

**Figure 24.1** User Interface Created by the Java Object

### Example 3: Creating a Custom Class Loader

You might not want to put all your Java classes in the classpath. You can write your own class loader to find the classes and load them. The following example illustrates how you can create a custom class loader.

In this example, you create a class, `x`, which resides in a folder or directory, `y`. You call the methods in this class by using the Java object with the classpath that includes the `y` folder.

```
/* Java code */
package com.sas;

public class x
{
    public void m()
    {
        System.out.println("method m in y folder");
    }

    public void m2()
    {
        System.out.println("method m2 in y folder");
    }
}

/* DATA step code */
data _null_;
dcl javaobj j('com/sas/x');
j.callvoidmethod('m');
j.callvoidmethod('m2');
run;
```

The following lines are written to the SAS log.

```
method m in y folder
method m2 in y folder
```

Suppose you have another class, **x**, that is stored in a different folder, **z**.

```
/* Java code
*/
package com.sas;

public class z
{
    public void m()
    {
        System.out.println("method m in y folder");
    }

    public void m2()
    {
        System.out.println("method m2 in y folder");
    }
}
```

You can call methods in this class instead of the class in folder **y** by changing the classpath, but this requires restarting SAS. The following method allows for more dynamic control of how classes are loaded.

To create a custom class loader, first you create an interface that contains all the methods that you will call through the Java object—in this program, **m** and **m2**.

```
/* Java
code */
public interface apiInterface
{
    public void m();
    public void m2();
}
```

Then you create a class for the actual implementation.

```
/* Java code */
import com.sas.x;

public class apiImpl implements apiInterface
{
    private x x;

    public apiImpl()
    {
        x = new x();
    }

    public void m()
    {
        x.m();
    }

    public void m2()
    {
        x.m2();
    }
}
```

```
}
```

These methods are called by delegating to the Java object instance class. Note that the code to create the `apiClassLoader` custom class loader is provided later in this section.

```
/* Java code */
public class api
{
    /* Load classes from the z folder */
    static ClassLoader customLoader = new apiClassLoader("C:\\\\z");
    static String API_IMPL = "apiImpl";
    apiInterface cp = null;

    public api()
    {
        cp = load();
    }

    public void m()
    {
        cp.m();
    }

    public void m2()
    {
        cp.m2();
    }

    private static apiInterface load()
    {
        try
        {
            Class aClass = customLoader.loadClass(API_IMPL);
            return (apiInterface) aClass.newInstance();
        }
        catch (Exception e)
        {
            e.printStackTrace();
            return null;
        }
    }
}
```

The following DATA step program calls these methods by delegating through the `api` Java object instance class. The Java object instantiates the `api` class, which creates a custom class loader to load classes from the `z` folder. The `api` class calls the custom loader and returns an instance of the `apiImpl` interface implementation class to the Java object. When methods are called through the Java object, the `api` class delegates them to the implementation class.

```
/* DATA step code */
data _null_;
dcl javaobj j('api');
j.callvoidmethod('m');
j.callvoidmethod('m2');
run;
```

The following lines are written to the SAS log:

```
method m is z folder
method m2 in z folder
```

In the previous Java code, you could also use .jar files to augment the classpath in the `ClassLoader` constructor.

```
static ClassLoader customLoader = new apiClassLoader("C:\\z;C:\\temp\\some.jar");
```

In this case, the Java code for the custom class loader is as follows. This code for this class loader can be added to or modified as needed.

```
import java.io.*;
import java.util.*;
import java.util.jar.*;
import java.util.zip.*;

public class apiClassLoader extends ClassLoader
{
    //class repository where findClass performs its search
    private List classRepository;

    public apiClassLoader(String loadPath)
    {
        super(apiClassLoader.class.getClassLoader());
        initLoader(loadPath);
    }

    public apiClassLoader(ClassLoader parent, String loadPath)
    {
        super(parent);
        initLoader(loadPath);
    }

    /**
     * This method will look for the class in the class repository. If
     * the method cannot find the class, the method will delegate to its parent
     * class loader.
     *
     * @param className A String specifying the class to be loaded
     * @return A Class object loaded by the apiClassLoader
     * @throws ClassNotFoundException if the method is unable to load the class
     */
    public Class loadClass(String name) throws ClassNotFoundException
    {
        // Check if the class is already loaded
        Class loadedClass = findLoadedClass(name);

        // Search for class in local repository before delegating
        if (loadedClass == null)
        {
            loadedClass = myFindClass(name);
        }

        // If class not found, delegate to parent
        if (loadedClass == null)
        {
            loadedClass = this.getClass().getClassLoader().loadClass(name);
        }
    }
}
```

```

        return loadedClass;
    }

private Class myFindClass(String className) throws ClassNotFoundException
{
    byte[] classBytes = loadFromCustomRepository(className);
    if(classBytes != null)
    {
        return defineClass(className,classBytes,0,classBytes.length);
    }
    return null;
}

/**
 * This method loads binary class file data from the classRepository.
 */
private byte[] loadFromCustomRepository(String classFileName)
throws ClassNotFoundException
{
    Iterator dirs = classRepository.iterator();
    byte[] classBytes = null;
    while (dirs.hasNext())
    {
        String dir = (String) dirs.next();

        if (dir.endsWith(".jar"))
        {
            // Look for class in jar

            String jclassFileName = classFileName;

            jclassFileName = jclassFileName.replace('.', '/');
            jclassFileName += ".class";

            try
            {
                JarFile j = new JarFile(dir);
                for (Enumeration e = j.entries(); e.hasMoreElements() ;)
                {
                    Object n = e.nextElement();

                    if (jclassFileName.equals(n.toString()))
                    {
                        ZipEntry zipEntry = j.getEntry(jclassFileName);
                        if (zipEntry == null)
                        {
                            return null;
                        }
                        else
                        {
                            // read file
                            InputStream is = j.getInputStream(zipEntry);
                            classBytes = new byte[is.available()];
                            is.read(classBytes);
                            break;
                        }
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
catch (Exception e)
{
    System.out.println("jar file exception");
    return null;
}
}
else
{
    // Look for class in directory
    String fclassName = className;

    fclassName = fclassName.replace('.', File.separatorChar);
    fclassName += ".class";

    try
    {
        File file = new File(dir,fclassName);
        if(file.exists()) {
            //read file
            InputStream is = new FileInputStream(file);
            classBytes = new byte[is.available()];
            is.read(classBytes);
            break;
        }
    }
    catch(IOException ex)
    {
        System.out.println("IOException raised while reading class
file data");
        ex.printStackTrace();
        return null;
    }
}
return classBytes;
}

private void initLoader(String loadPath)
{
/*
 * loadPath is passed in as a string of directories/jar files
 * separated by the File.pathSeparator
 */
classRepository = new ArrayList();
if((loadPath != null) && !(loadPath.equals(""))))
{
    StringTokenizer tokenizer =
        new StringTokenizer(loadPath,File.pathSeparator);
    while(tokenizer.hasMoreTokens())
    {
        classRepository.add(tokenizer.nextToken());
    }
}
}

```

```
    }
```

# 25

## Array Processing

---

<i>Definitions for Array Processing</i>	<b>603</b>
<i>A Conceptual View of Arrays</i>	<b>604</b>
One-Dimensional Array	604
Two-Dimensional Array	605
<i>Syntax for Defining and Referencing an Array</i>	<b>605</b>
<i>Processing Simple Arrays</i>	<b>606</b>
Grouping Variables in a Simple Array	606
Using a DO Loop to Repeat an Action	607
Using a DO Loop to Process Selected Elements in an Array	607
Selecting the Current Variable	608
Defining the Number of Elements in an Array	609
Rules for Referencing Arrays	609
<i>Variations on Basic Array Processing</i>	<b>610</b>
Determining the Number of Elements in an Array Efficiently	610
DO WHILE and DO UNTIL Expressions	611
Using Variable Lists to Define an Array Quickly	611
<i>Multidimensional Arrays: Creating and Processing</i>	<b>612</b>
Grouping Variables in a Multidimensional Array	612
Using Nested DO Loops	612
<i>Specifying Array Bounds</i>	<b>614</b>
Identifying Upper and Lower Bounds	614
Determining Array Bounds: LBOUND and HBOUND Functions	615
When to Use the HBOUND Function Instead of the DIM Function	615
Specifying Bounds in a Two-Dimensional Array	616
<i>Examples of Array Processing</i>	<b>616</b>
Example 1: Using Character Variables in an Array	616
Example 2: Assigning Initial Values to the Elements of an Array	617
Example 3: Creating an Array for Temporary Use in the Current DATA Step	618
Example 4: Performing an Action on All Numeric Variables	619

---

## 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:

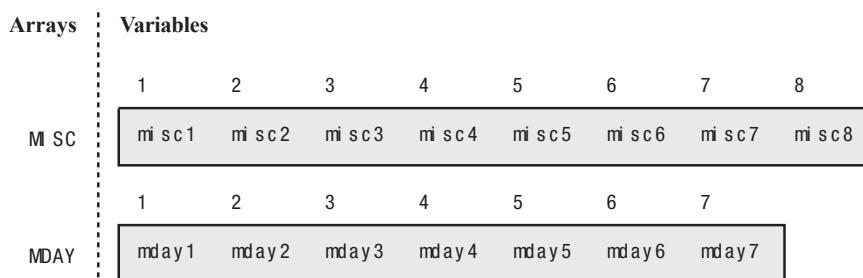
- grouping variables into arrays
- selecting a current variable for an action
- repeating an action

## A Conceptual View of Arrays

### One-Dimensional Array

The following figure is a conceptual representation of two one-dimensional arrays, Misc and Mday.

**Figure 25.1** 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 25.2 Example of a Two-Dimensional Array**

First Dimension Expense Categories	Second Dimension 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	carrrt1	carrrt2	carrrt3	carrrt4	carrrt5	carrrt6	carrrt7	carrrt8
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 25.1** 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 25.2 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 25.1 Using an Array Statement to Process Missing Data Values*

<b>Number of Books Sold</b>				
<b>Obs</b>	<b>Reference</b>	<b>Usage</b>	<b>Introduction</b>	
<b>1</b>	45	63	113	
<b>2</b>	0	75	150	
<b>3</b>	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  $\{n, \dots\}$  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 25.3* 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 25.2 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(temprrg);
    do j=1 to dim2(temprrg);
      temprrg{i,j}=round(temprrg{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 of Array Processing

### Example 1: Using Character Variables in an Array

You can specify character variables and their lengths in ARRAY statements. The following example groups variables into two arrays, NAMES and CAPITALS. The dollar sign (\$) tells SAS to create the elements as character variables. If the variables have already been declared as character variables, a dollar sign in the array is not necessary. The INPUT statement reads all the variables in array NAMES.

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.

```
options linesize=80 pagesize=60;

data text;
array names{*} $ n1-n5;
array capitals{*} $ c1-c5;
input names{*};
do i=1 to 5;
    capitals{i}=upcase(names{i});
end;
datalines;
smithers michaels gonzalez hurth frank
;

proc print data=text;
title 'Names Changed from Lowercase to Uppercase';
run;
```

The following output shows the TEXT data set.

**Output 25.3 Using Character Variables in an Array**

<b>Names Changed from Lowercase to Uppercase</b>												
<b>Obs</b>	<b>n1</b>	<b>n2</b>	<b>n3</b>	<b>n4</b>	<b>n5</b>	<b>c1</b>	<b>c2</b>	<b>c3</b>	<b>c4</b>	<b>c5</b>	<b>i</b>	
<b>1</b>	smithers	michaels	gonzalez	hurth	frank	SMITHERS	MICHAELS	GONZALEZ	HURTH	FRANK	6	

## Example 2: 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. If the value of the element in Score is greater than or equal to the value of the element in Test, the variable NewScore is assigned the value in the element Score. The OUTPUT statement writes the observation to the SAS data set.

The INPUT statement reads a value for the variable named ID and then reads values for all the variables in the Score array.

```
options linesize=80 pagesize=60;

data score1(drop=i);
array test{3} t1-t3 (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=score1;
title 'Data Set SCORE1';
run;

```

The following output shows the Score1 data set.

**Output 25.4** 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 3: 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 25.5 Using \_TEMPORARY\_ Arrays**

<b>Data Set SCORE2</b>				
<b>s1</b>	<b>s2</b>	<b>s3</b>	<b>id</b>	<b>NewScore</b>
99	60	82	1234	99
99	60	82	1234	82
80	85	75	5678	85
80	85	75	5678	75

## Example 4: Performing an Action on All Numeric Variables

This example multiplies all 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 25.6 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

**PART 5****SAS Files Concepts**

<i>Chapter 26</i>	<i>SAS Libraries</i>	623
<i>Chapter 27</i>	<i>SAS Data Sets</i>	639
<i>Chapter 28</i>	<i>SAS Data Files</i>	655
<i>Chapter 29</i>	<i>SAS Views</i>	721
<i>Chapter 30</i>	<i>Stored Compiled DATA Step Programs</i>	731
<i>Chapter 31</i>	<i>DICTIONARY Tables</i>	741
<i>Chapter 32</i>	<i>SAS Catalogs</i>	747
<i>Chapter 33</i>	<i>About SAS/ACCESS Software</i>	757
<i>Chapter 34</i>	<i>Processing Data Using Cross-Environment Data Access (CEDA)</i>	765
<i>Chapter 35</i>	<i>Cross-Release Compatibility and Migration</i>	779
<i>Chapter 36</i>	<i>File Protection</i>	785
<i>Chapter 37</i>	<i>SAS Engines</i>	803
<i>Chapter 38</i>	<i>SAS File Management</i>	815
<i>Chapter 39</i>	<i>External Files</i>	821



# SAS Libraries

---

<i>Definition of a SAS Library</i> .....	623
<i>Library Engines</i> .....	625
<i>Library Names</i> .....	625
Physical Names and Logical Names (Librefs) .....	625
Assigning Librefs .....	626
Associating and Clearing Logical Names (Librefs) with the LIBNAME Statement .....	627
Reserved Librefs .....	627
Accessing Remote SAS Libraries on SAS/CONNECT, SAS/SHARE, and WebDAV Servers .....	628
<i>Library Concatenation</i> .....	629
Definition of Library Concatenation .....	629
How SAS Concatenates Library Members .....	629
Rules for Library Concatenation .....	630
<i>Permanent and Temporary Libraries</i> .....	631
<i>Definition of a Metadata-Bound Library</i> .....	632
<i>SAS System Libraries</i> .....	632
Introduction to SAS System Libraries .....	632
Work Library .....	632
User Library .....	633
Sashelp Library .....	634
Sasuser Library .....	635
<i>Sequential Data Libraries</i> .....	635
<i>Tools for Managing Libraries</i> .....	636
SAS Utilities .....	636
Library Directories .....	637
Accessing Permanent SAS Files without a Libref .....	637
Operating Environment Commands .....	638

---

## Definition of a SAS Library

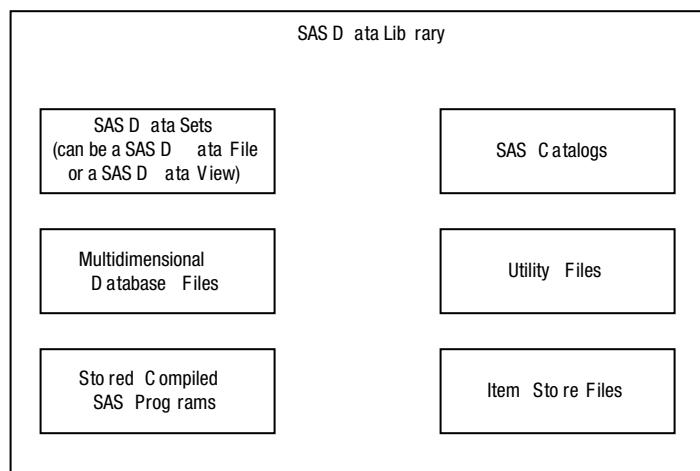
The logical concept of a SAS library remains constant, regardless of the operating environment. In any operating environment where SAS can be installed, the structure for organizing, locating, and managing SAS files is the same.

At the operating environment level, however, a SAS library has different physical implementations. Most SAS libraries implement the storage of files in a manner similar to how the operating environment stores and accesses files.

For example, in directory-based operating environments, a SAS library is a group of SAS files that are stored in the same directory and accessed by the same engine. Other files can be stored in the directory, but only the files with file extensions that are assigned by SAS are recognized as part of the SAS library. Under z/OS, a SAS library can be implemented as either a bound library in a traditional OS data set or as a directory under UNIX System Services.

SAS files can be any of the following file types:

- SAS data set (SAS data file or SAS view)
- SAS catalog
- stored compiled SAS program
- SAS utility file
- access descriptors
- multi-dimensional database files such as MDDB, FDB, and DMDB files
- item store files



Each SAS file, in turn, stores information in smaller units that are characteristic of the **SAS file type**. For example, SAS data sets store information as variables and observations, while SAS catalogs store information in units called **entries**. SAS determines the type of a file from the context of the SAS program in which the file is created or specified. Therefore, a library can contain files with the same name but with different member types.

SAS libraries can contain files that you create, or they can be one of several special libraries that SAS provides for convenience, support, and customizing capability such as the Work library. SAS does not limit the number of SAS files that you can store in a SAS library.

---

## Library Engines

Each SAS library is associated with a library engine. SAS library engines are software components that form the interface between SAS and the SAS library. It is the SAS library engine that locates files in a SAS library and renders the file contents to SAS in a form that it can recognize. Library engines perform such tasks as:

- reading and writing data
- listing the files in the library
- deleting and renaming files

SAS has a Multi Engine Architecture in order to read to and write from files in different formats. Each SAS engine has specific processing characteristics, such as the ability to

- process a SAS file generated by an older version of SAS
- read database files created by other software programs
- store and access files on disk or tape
- determine how variables and observations are placed in a file
- place data into memory from its physical location
- transport SAS files between operating environments

You generally are not aware of the particular type of engine that is processing data at any given time. If you issue an instruction that is not supported by the engine, an error message is displayed in the SAS log. When needed, you can select a specific engine to perform a task. But usually, you do not have to specify an engine, because SAS automatically selects the appropriate one.

More than one engine might be involved in processing a DATA step; for example, one engine might be used to input data, and another engine might be used to write observations to the output data set.

For more information about library engines, including a list of engines available in Base SAS, see “[About Library Engines](#)” on page 808.

---

## Library Names

---

### Physical Names and Logical Names (Librefs)

Before you can use a SAS library, you must tell SAS where it is. SAS recognizes SAS libraries based on either operating environment naming conventions or SAS naming conventions. There are two ways to define SAS libraries:

- a physical location name that the operating environment recognizes
- a logical name (libref) that you assign using the LIBNAME statement, LIBNAME function, or the New Library window

The physical location name of the SAS library is a name that identifies your SAS files to the operating environment. The physical location name must conform to the naming conventions of your operating environment. The physical location name fully identifies the directory, or operating environment data set that contains the SAS library.

The logical name, or libref, is the way you identify a group of files to SAS. A libref is a name that you associate with the physical location of the SAS library.

## Assigning Librefs

Librefs can be assigned using the following methods:

- LIBNAME statement
- LIBNAME function
- New Library window that is available in your toolbar
- operating environment commands

Once the libref is assigned, you can read, create, or update files in a SAS library. A libref is valid only for the current SAS session, unless it is assigned using the New Library window with the **Enable at start-up** box checked.

A libref can have a maximum length of eight characters. You can use the LIBREF function to verify that a libref has been assigned. Librefs can be referenced repeatedly within a SAS session. SAS does not limit the number of librefs that you can assign during a session. However, your operating environment or site might set limitations. If you are running in batch mode, the library must exist before you can allocate or assign it. In interactive mode, you might be allowed to create it if it does not already exist.

**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.

*Table 26.1 Syntax for Assigning a Libref*

Operating Environment	Examples
DOS, Windows	<code>libname mylibref 'c:\root\mystuff\sastuff\work';</code>
UNIX	<code>libname mylibref '/u/mystuff/sastuff/work';</code>
z/OS	<code>libname mylibref 'userid.mystuff.sastuff.work';</code> <code>libname mylibref '/mystuff/sastuff/work';</code>

You can also access files without using a libref. See “[Accessing Permanent SAS Files without a Libref](#)” on page 637.

---

## Associating and Clearing Logical Names (Librefs) with the LIBNAME Statement

You can assign or clear a physical name with a libref using the LIBNAME statement, which is described in [SAS DATA Step Statements: Reference](#), or the LIBNAME function, which is described in [SAS Functions and CALL Routines: Reference](#).

**Operating Environment Information:** For some operating environments, you can use operating environment commands to associate a libref with a SAS library. When using operating environment commands to assign librefs to a SAS library, the association might persist beyond the SAS session in which the libref was created. For some operating environments, you can use only the LIBNAME statement or function. See the SAS documentation for your operating environment for more information about assigning librefs.

The most common form of the LIBNAME statement is used in this example to associate the libref Annual with the physical name of the SAS library.

```
libname annual 'SAS-library';
```

If you use the LIBNAME statement to assign the libref, SAS clears (deassigns) the libref automatically at the end of each SAS session. If you want to clear the libref Annual before the end of the session, you can issue the following form of the LIBNAME statement:

```
libname annual clear;
```

SAS also provides a New Library window to assign or clear librefs and SAS Explorer to view, add, or delete SAS libraries. You can select the **New Library** or the **SAS Explorer** icon from the Toolbar.

---

## Reserved Librefs

SAS reserves a few names for special uses. You should not use Sashelp, Sasuser, or Work as librefs, except as intended. The purpose and content of these libraries are discussed in [“Permanent and Temporary Libraries” on page 631](#).

**Operating Environment Information:** There are other librefs reserved for SAS under some operating environments. In addition, your operating environment might have reserved certain words that cannot be used as librefs. See the SAS documentation for your operating environment for more information.

---

## Accessing Remote SAS Libraries on SAS/CONNECT, SAS/SERVE, and WebDAV Servers

### Remote Library Access for SAS/CONNECT and SAS/SERVE

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 gets overwritten in subsequent client requests for server data.

The LIBNAME statement can be used to access SAS data sets across computers that have different architectures. The LIBNAME statement also provides Read-Only access to some SAS catalog entry types across computers that have different architectures.

The LIBNAME statement provides access to remote server data by associating a SAS library reference (libref) with a permanent SAS library.

SAS/CONNECT Example:

The SAS/CONNECT client creates a LIBNAME statement to access a server library that is located on a SAS/CONNECT server. The client creates the new libref, Reports.

```
signon rempc;
libname reports 'd:\prod\reports' server=rempc;
```

The SAS/CONNECT client signs on to the SAS/CONNECT server named REMPC. A server library is assigned to the client session. The value for SERVER= is the same as the server session ID that is used in the SIGNON statement.

For more information about SAS/CONNECT, see [SAS/CONNECT User's Guide](#).

SAS/SERVE Example:

The SAS/SERVE client uses a LIBNAME statement to access a server library via the existing libref, Sales, which was pre-defined at the SAS/SERVE server for client access.

```
libname sales server=server1;
```

For more information about SAS/SERVE, see [SAS/SERVE User's Guide](#).

### Remote Library Access for WebDAV Servers

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. SAS supports the WebDAV protocol under the UNIX and Windows operating environments.

You use a LIBNAME statement to access WebDAV servers, as shown in the following example:

```
libname davdata v9 "http://www.webserver.com/users/mydir/datadir"
webdav user="mydir" pw="12345";
```

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 SAS Work directory, unless you use the LOCALCACHE= option in the LIBNAME statement, which specifies a different directory 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.

For more information, see “[WHEREUP= Data Set Option](#)” in *SAS Data Set Options: Reference*.

---

## Library Concatenation

---

### Definition of Library Concatenation

Concatenation is the logical combining of two or more libraries. Concatenation enables you to access the SAS data sets in several libraries with one libref.

You can concatenate two or more libraries by specifying their librefs or physical names in the LIBNAME statement or function.

Physical names must be enclosed in single or double quotation marks in a LIBNAME statement. Otherwise, SAS looks for a previously assigned libref with the same name.

In the following examples, Summer, Winter, Spring, Fall, and Annual are previously defined librefs:

```
libname annual (summer winter spring fall);  
  
libname annual ('SAS-library-1' 'SAS-library-2' 'SAS-library-3');  
  
libname annual ('SAS-library' winter spring fall);  
  
libname total (annual 'SAS-library');
```

---

### How SAS Concatenates Library Members

When there are members of the same name in more than one library, the first occurrence of the member is used for input and update processes. Output always goes to the first library.

This example contains three SAS libraries, and each library contains two SAS data files:

```
Lib1  
    Apples and Pears  
  
Lib2  
    Apples and Oranges
```

## Lib3

Oranges and Plums

The LIBNAME statement concatenates Lib1, Lib2, and Lib3:

```
libname fruit (lib1 lib2 lib3);
```

The concatenated library Fruit has the following members:

- Apples
- Pears
- Oranges
- Plums

**Note:** Output always goes to the first library. For example, the following statement writes to the first library in the concatenation, Lib1:

```
data fruit.oranges;
```

Note that in this example, if the file Apples in Lib1 was different from the file Apples in Lib2, and if an update to Apples was specified, it is updated only in Lib1 because that is the first occurrence of the member Apples.

For complete documentation on library concatenation, see the “[LIBNAME Statement](#)” in *SAS Global Statements: Reference*.

**Operating Environment Information:** For more information about how specific operating environments handle concatenation, see the SAS documentation for your operating environment.

## Rules for Library Concatenation

After you create a library concatenation, you can specify the libref in any context that accepts a simple (nonconcatenated) libref. These rules determine how SAS files (that is, members of SAS libraries) are located among the concatenated libraries:

- If you specify any options or an engine, the options apply only to the libraries that you specified with the physical name, not to any library that you specified with a libref.
- When a SAS file is opened for input or update, the concatenated libraries are searched and the first occurrence of the specified file is used.
- When a SAS file is opened for output, it is created in the first library that is listed in the concatenation.
- When you delete or rename a SAS file, only the first occurrence of the file is affected.
- Any time a list of SAS files is displayed, only one occurrence of a filename is shown, even if the name occurs multiple times in the concatenation. For example, if library One contains A.DATA and library Two contains A.DATA, only A.DATA from library One is listed because it is the first occurrence of the filename.

In addition, a SAS file that is logically connected to another file (such as an index to a data file) is listed only if the parent file is the first (or only) occurrence of the filename. For example, if library One contains A.DATA and library Two contains

A.DATA and A.INDEX, only A.DATA from library One is listed. A.DATA and A.INDEX from library Two are not listed.

- If any library in the concatenation is sequential, then the concatenated library is considered sequential by applications that require random access. For example, the DATASETS procedure cannot process sequential libraries, and therefore cannot process a concatenated library that contains one or more sequential libraries.
- The attributes of the first library that is specified determine the attributes of the concatenation. For example, if the first SAS library that is listed is “read only,” then the entire concatenated library is “read only.”
- Once a libref has been assigned in a concatenation, any changes made to the libref does not affect the concatenation.
- Changing a data set name to an existing name in the concatenation will fail.

---

## Permanent and Temporary Libraries

SAS libraries are generally stored as permanent data libraries. However, SAS provides a temporary or scratch library where you can store files for the duration of a SAS session or job.

A permanent SAS library is one that resides on the external storage medium of your computer and is not deleted when the SAS session terminates. Permanent SAS libraries are stored until you delete them. The library is available for processing in subsequent SAS sessions. When working with files in a permanent SAS library, you generally specify a libref as the first part of a two-level SAS filename. The libref tells SAS where to find or store the file.

**Note:** You can also skip using a libref and point directly to the file that you want to use, using syntax that your operating system understands. An example of this in the Windows environment is

```
data 'C:\root\sasfiles\myfile.ext';
```

**Operating Environment Information:** Files are specified differently in various operating environments. See the SAS documentation for your operating environment for more information.

A temporary SAS library is one that exists only for the current SAS session or job. SAS files that are created during the session or job are held in a special work space that might or might not be an external storage medium. This work space is generally assigned the default libref Work. Files in the temporary Work library can be used in any DATA step or SAS procedure during the SAS session, but they are typically not available for subsequent SAS sessions. Normally, you specify that data sets be stored in or retrieved from this library by specifying a one-level name. Files held in the Work library are deleted at the end of the SAS session if it ends normally.

There are a number of SAS system options that enable you to customize how you name and work with your permanent and temporary SAS libraries. See the USER=, WORK=, WORKINIT, and WORKTERM system options in *SAS System Options: Reference* for more information.

---

## Definition of a Metadata-Bound Library

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*.

---

## SAS System Libraries

---

### Introduction to SAS System Libraries

Four special libraries supplied by SAS provide convenience, support, and customization capability:

- Work
- User
- Sashelp
- Sasuser

---

## Work Library

### Definition of Work Library

The Work library is the temporary (scratch) library that is automatically defined by SAS at the beginning of each SAS session. The Work library stores two types of temporary files: those that you create and those that are created internally by SAS as part of normal processing. Typically, the Work library is deleted at the end of each SAS session if the session terminates normally.

## Using the Work Library

To store or retrieve SAS files in the Work library, specify a one-level name in your SAS program statements. The libref Work is automatically assigned to these files as a system default unless you have assigned the User libref. The following examples contain valid names for SAS data sets stored in the Work library:

```
data test2;
data work.test2;
proc contents data=testdata;
proc contents data=worktestdata;
```

**Operating Environment Information:** The Work library is implemented differently in various operating environments. See the SAS documentation for your operating environment for more information.

## Relation to the User Library

While the Work library is designed to hold temporary files used during the current SAS session, the User library is designed to hold files after the SAS session is over. If you associate the libref User with a SAS library, use a one-level name to create and access files that are not deleted at the end of your SAS session. When SAS encounters a one-level filename, it looks first in the User library, if it has been defined, and then it looks in Work. If you want to place a file in the User library, so that it is not deleted after your SAS session is over, any single-level file goes there by default. At that point, if you want to create a temporary file in Work, you must use a two-level name, such as Work.Name.

## User Library

### Definition of User Library

The User library enables you to read, create, and write to files in a SAS library other than Work without specifying a libref as part of the SAS filename. Once you associate the libref User with a SAS library, SAS stores any file with a one-level name in that library. Unlike the Work library, files stored in this library are not deleted by SAS when the session terminates.

### Ways to Assign the User Libref

You can assign the User libref using one of the following methods:

- LIBNAME statement
- LIBNAME function
- USER= system option
- operating environment command

In this example, the LIBNAME statement is used with a DATA step, which stores the data set Region in a permanent SAS library.

```

libname user 'SAS-library';
data region;
... more DATA step statements ...
run;

```

In this example, the LIBNAME function assigns the User libref:

```

data _null_;
x=libname ('user', 'SAS-library');
run;

```

When assigning a libref using the USER= system option, you must first assign a libref to a SAS library, and then use the USER= system option to specify that library as the default for one-level names. In this example, the DATA step stores the data set Prochlor in the SAS library Testlib.

```

libname testlib 'SAS-library';
options user=testlib;
data prochlor;
... more DATA step statements ...
run;

```

**Operating Environment Information:** The methods and results of assigning the User libref vary slightly from one operating environment to another. See the SAS documentation for your operating environment for more information.

## Relation to Work Library

The User libref overrides the default libref Work for one-level names. When you refer to a file by a one-level name, SAS looks first for the libref User. If User is assigned to a SAS library, files with one-level names are stored there. If you have not assigned the libref User to a library, the files with one-level names are stored in the temporary library Work. To refer to SAS files in the Work library while the User libref is assigned, you must specify a two-level name with Work as the libref. Data files that SAS creates internally still go to the Work library.

## Sashelp Library

Each SAS site receives the Sashelp library, which contains a group of catalogs and other files containing information that is used to control various aspects of your SAS session. The defaults stored in this library are for everyone using SAS at your installation. Your personal settings are stored in the Sasuser library, which is discussed later in this section.

If SAS products other than Base SAS are installed at your site, the Sashelp library contains catalogs that are used by those products. In many instances, the defaults in this library are customized for your site by your on-site SAS support personnel. You can list the catalogs stored at your site by using one of the file management utilities discussed later in this section.

---

## Sasuser Library

The Sasuser library contains SAS catalogs that enable you to customize features of SAS for your needs. If the defaults in the Sashelp library are not suitable for your applications, you can modify them and store your personalized defaults in your Sasuser library. For example, in Base SAS, you can store your own defaults for function key settings or window attributes in a personal Profile catalog named Sasuser.Profile.

SAS assigns the Sasuser library during system initialization, according to the information supplied by the Sasuser system option.

A system option called RSASUSER enables the system administrator to control the mode of access to the Sasuser library at installations that have one Sasuser library for all users and that want to prevent users from modifying it.

**Operating Environment Information:** In most operating environments, the Sasuser library is created if it does not already exist. However, the Sasuser library is implemented differently in various operating environments. See the SAS documentation for your operating environment for more information.

---

## Sequential Data Libraries

SAS provides a number of features and procedures for reading from and writing to files that are stored on sequential format devices, either disk or tape. Before you store SAS libraries in sequential format, you should consider the following:

- You cannot use random access methods with sequential SAS data sets.
- You can access only one of the SAS files in a sequential library, or only one of the SAS files on a tape, at any point in a SAS job.

For example, you cannot read two or more SAS data sets in the same library or on the same tape at the same time in a single DATA step. However, you can access:

- two or more SAS files in different sequential libraries, or on different tapes at the same time, if there are enough tape drives available
- a SAS file during one DATA or PROC step, and then access another SAS file in the same sequential library or on the same tape during a later DATA or PROC step

Also, when you have more than one SAS data set on a tape or in a sequential library in the same DATA or PROC step, one SAS data set file might be opened during the compilation phase. The additional SAS data sets are opened during the execution phase. For more information, see the “[SET Statement](#)” in [SAS DATA Step Statements: Reference](#).

- For some operating environments, you can read from or write to SAS data sets only during a DATA or PROC step. However, you can always use the COPY procedure to transfer all members of a SAS library to tape for storage and backup purposes.

- Considerations specific to your site can affect your use of tape. For example, it might be necessary to manually mount a tape before the SAS libraries become available. Consult your operations staff if you are not familiar with using tape storage at your location.

For information about sequential engines, see [Chapter 37, “SAS Engines,” on page 803](#).

**Operating Environment Information:** The details for storing and accessing SAS files in sequential format vary with the operating environment. See the SAS documentation for your operating environment for more information.

## Tools for Managing Libraries

### SAS Utilities

The SAS utilities that are available for SAS file management enable you to work with more than one SAS file at a time, as long as the files belong to the same library. The advantage of learning and using SAS Explorer, functions, options, and procedures is that they automatically copy, rename, or delete any index files or integrity constraints, audit trails, backups, and generation data sets that are associated with your SAS data files. Another advantage is that SAS utility procedures work on any operating environment at any level.

There are several SAS window options, functions, and procedures available for performing file management tasks. You can use the following features alone or in combination, depending on what works best for you. See [“Choosing the Right Procedure” in Base SAS Procedures Guide](#) for detailed information about SAS utility procedures. The SAS windowing environment and how to use it for managing SAS files is discussed in [Chapter 16, “Introduction to the SAS Windowing Environment,” on page 385](#) and [Chapter 17, “Managing Your Data in the SAS Windowing Environment,” on page 405](#) as well as in the online Help.

#### CATALOG procedure

provides catalog management utilities with the COPY, CONTENTS, and APPEND procedures.

#### DATASETS procedure

provides all library management functions for all member types except catalogs. If your site does not use the SAS Explorer, or if SAS executes in batch or interactive line mode, using this procedure can save you time and resources.

#### SAS Explorer

includes windows that enable you to perform most file management tasks without submitting SAS program statements. Type LIBNAME, CATALOG, or DIR in the Toolbar window to use SAS Explorer, or select the Explorer icon from the **Toolbar** menu.

#### DETAILS system option

Sets the default display for file information when using the CONTENTS or DATASETS procedure. When enabled, DETAILS provides additional information about files, depending on which procedure or window you use.

---

## Library Directories

SAS Explorer and SAS procedures enable you to obtain a list, or directory, of the members in a SAS library. Each directory contains the name of each member and its member type. For the member type DATA, the directory indicates whether an index, audit trail, backup, or generation data set is associated with the data set. The directory also describes some attributes of the library, but the amount and nature of this information vary with the operating environment.

**Note:** SAS libraries can also contain various SAS utility files. These files are not listed in the library directory and are for internal processing.

---

## Accessing Permanent SAS Files without a Libref

SAS provides another method of accessing files in addition to assigning a libref with the LIBNAME statement or using the New Library window. To use this method, enclose the filename, or the filename and the SAS library, in single quotation marks.

For example, in a directory-based system, if you want to create a data set named MyData in your default directory, that is, in the directory that you are running SAS in, you can write the following line of code:

```
data 'mydata';
```

SAS creates the data set and remembers its location for the duration of the SAS session.

If you omit the single quotation marks, SAS creates the data set MyData in the temporary Work subdirectory, named Work.MyData:

```
data mydata;
```

If you want to create a data set named MyData in a library other than the directory in which you are running SAS, enclose the entire pathname in quotation marks, following the naming conventions of your operating environment. For example, the following DATA step creates a data set named Foo in the directory C:\sasrun\mydata

```
data 'c:\sasrun\mydata\foo';
```

This method of accessing files works on all operating environments and in most contexts where a *libref.data-set-name* is accepted as a SAS data set. Most data set options can be specified with a quoted name.

You cannot use quoted names for the following:

- SAS catalogs
- MDDB and FDB references
- contexts that do not accept a libref, such as the SELECT statement of PROC COPY and most PROC DATASETS statements
- PROC SQL
- DATA step stored programs, or SAS views
- SAS Component Language (SCL) open function

The following table shows some examples of DATA statements that access SAS data files without using a libref.

**Table 26.2 Example DATA Statements That Access SAS Files without Using a Libref**

Operating Environment	Examples
DOS, Windows	data 'c:\root\mystuff\sasstuff\work\myfile';
UNIX	data '/u/root/mystuff/sastuff/work/myfile';
z/OS	data 'user489.mystuff.saslib(member1)'; /* bound SAS library */  data '/mystuff/sasstuff/work/myfile'; /* UNIX file system library */

## Operating Environment Commands

You can use operating environment commands to copy, rename, and delete the operating environment file or files that make up a SAS library. However, to maintain the integrity of your files, you must know how the SAS library model is implemented in your operating environment. For example, in some operating environments, SAS data sets and their associated indexes can be copied, deleted, or renamed as separate files. If you rename the file that contains the SAS data set, but not its index, the data set is marked as damaged.

**CAUTION! Using operating environment commands can damage files.** You can avoid problems by always using SAS utilities to manage SAS files.

# SAS Data Sets

---

<i>Definition of a SAS Data Set</i> .....	639
<i>Descriptor Information for a SAS Data Set</i> .....	640
<i>Data Set Names</i> .....	642
Where to Use Data Set Names .....	642
How and When SAS Data Set Names Are Assigned .....	642
Parts of a Data Set Name .....	642
Two-level SAS Data Set Names .....	643
One-level SAS Data Set Names .....	643
<i>Data Set Lists</i> .....	644
<i>Special SAS Data Sets</i> .....	645
Null Data Sets .....	645
Default Data Sets .....	645
Automatic Naming Convention .....	645
<i>Sorted Data Sets</i> .....	646
The Sort Indicator .....	646
How SAS Uses the Sort Indicator to Improve Performance .....	651
Validating That a Data Set Is Sorted .....	652
<i>Tools for Managing Data Sets</i> .....	652
<i>Viewing and Editing SAS Data Sets</i> .....	653

---

## Definition of a SAS Data Set

A SAS data set is a SAS file stored in a SAS library that SAS creates and processes. A SAS data set contains data values that are organized as a table of observations (rows) and variables (columns) that can be processed by SAS software. A SAS data set also contains descriptor information such as the data types and lengths of the variables, as well as which engine was used to create the data.

A SAS data set can be one of the following:

**SAS data file**

contains both the data and the descriptor information. SAS data files have a member type of DATA. For specific information, see [Chapter 28, “SAS Data Files,” on page 655](#).

**SAS view**

is a virtual data set that points to data from other sources. SAS views have a member type of VIEW. For specific information, see [Chapter 29, “SAS Views,” on page 721](#).

**Note:** The term SAS data set is used when a SAS view and a SAS data file can be used in the same manner.

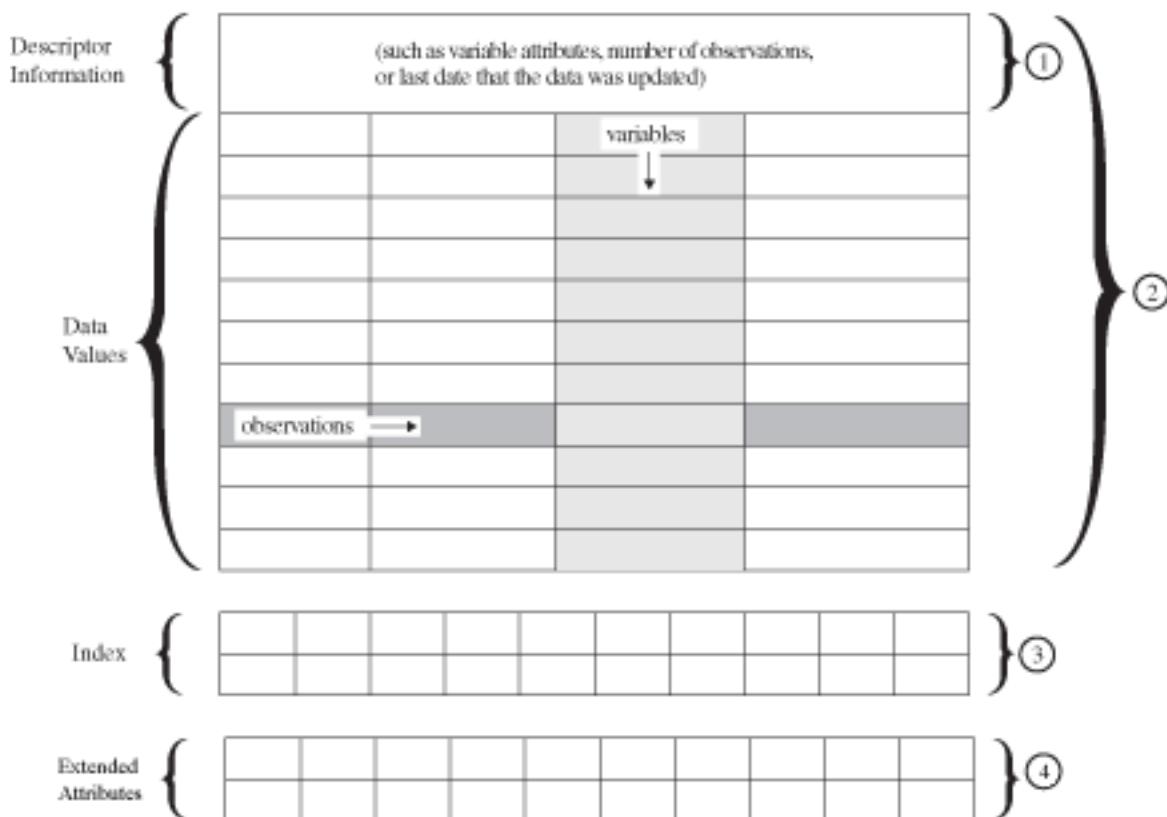
---

## Descriptor Information for a SAS Data Set

The descriptor information for a SAS data set makes the file self-documenting. That is, each data set can supply the attributes of the data set and of its variables. Once the data is in the form of a SAS data set, you do not have to specify the attributes of the data set or the variables in your program statements. SAS obtains the information directly from 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.

The following figure illustrates the logical components of a SAS data set:

**Figure 27.1** Logical Components of a SAS Data Set

The following items correspond to the numbers in the figure above:

- 1 A SAS view (member type VIEW) contains descriptor information and uses data values from one or more data sets.
- 2 A SAS data file (member type DATA) contains descriptor information and data values. SAS data sets can be a member type DATA (SAS data file) or VIEW (SAS view).
- 3 An index is a separate file that you can create for a SAS data file in order to provide direct access to specific observations. The index file has the same name as its data file and a member type of INDEX. Indexes can provide faster access to specific observations, particularly when you have a large data set.
- 4 Extended attributes are metadata that is defined on a data set or on a variable (column). Extended attributes are represented as name-value pairs and are created using the DATASETS procedure.

---

# Data Set Names

---

## Where to Use Data Set Names

You can use SAS data sets as input for DATA or PROC steps by specifying the name of the data set in the following:

- a SET statement
- a MERGE statement
- an UPDATE statement
- a MODIFY statement
- the DATA= option of a SAS procedure
- the OPEN function

---

## How and When SAS Data Set Names Are Assigned

You name SAS data sets when you create them. 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 an OUTPUT statement. If you do not specify a name for an output data set, SAS assigns a default name.

If you are creating SAS views, you assign the data set name using one of the following:

- the SQL procedure
- the ACCESS procedure
- the VIEW= option in the DATA statement

**Note:** Because you can specify both SAS data files and SAS views in the same program statements but cannot specify the member type, SAS cannot determine from the program statement which one you want to process. This is why SAS prevents you from giving the same name to SAS views and SAS data sets in the same library.

---

## Parts of a Data Set Name

The complete name of every SAS data set has three elements. You assign the first two; SAS supplies the third. The form for SAS data set names is as follows:

*libref.SAS-data-set.membertype*

The elements of a SAS data set name include the following:

*libref*

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

*SAS-data-set*

is the 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.

*membertype*

is assigned by SAS. The member type is DATA for SAS data files and VIEW for SAS views.

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. A two-level name consists of both the libref and the data set name. A one-level name consists of just the data set name.

## Two-level SAS Data Set Names

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:

*libref.SAS-data-set*

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. The following examples show the use of two-level names in SAS statements:

```
data revenue.sales;
proc sort data=revenue.sales;
```

## One-level SAS Data Set Names

You can omit the libref, and refer to data sets with a one-level name in the following form:

*SAS-data-set*

Data sets with one-level names are automatically assigned to one of two SAS libraries: Work or User. 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 [Chapter 26, "SAS Libraries," on page 623](#) for more information about using the User and Work libraries. The following examples show how one-level names are used in SAS statements.

```
/* create perm data set in location of USER=option*/
options user='c:\temp'
data test3;
```

```

/* create perm data set in current directory */
data 'test3';

/* create a temp data set in WORK directory if USER= is not specified*/
data stratifiedsample1;

```

## Data Set Lists

In the DATASETS procedure and the DATA step MERGE and SET statements, data set lists provide a quick way to reference existing groups of data sets. These data set lists can be numbered range lists or colon (name prefix) lists.

### **Numbered range lists**

require a series of data sets with the same name, except for the last character or characters, which are consecutive numbers. In a numbered range list, you can begin with any number and end with any number. For example, the following two lists refer to the same data sets:

```

sales1 sales2 sales3 sales4

sales1-sales4

```

If the numeric suffix of the first data set name contains leading zeros, the number of digits in the last data set name must be greater than or equal to the number of digits in the first data set name. Otherwise, an error occurs. For example, the data set lists *sales001–sales99* and *sales01–sales9* cause an error to occur. The data set list *sales001–sales999* is valid. If the numeric suffix of the first data set name does not contain leading zeros, then the number of digits in the numeric suffix of the first and last data set names does not have to be equal. For example, the data set list *sales1–sales999* is valid.

### **Colon (name prefix) lists**

require a series of data sets with the same starting character or characters. For example, the following two lists refer to the same data sets:

```

abc:

abc1 abc2 abcr abcx

```

In the DATASETS procedure data set lists can be used with the following statements:

- COPY SELECT
- COPY EXCLUDE
- DELETE
- REPAIR
- REBUILD
- the variables that are specified in MODIFY SORTEDBY

In the DATA step, data set lists can be used in the following SAS statements:

- [MERGE statement](#)
- [SET statement](#).

---

# Special SAS Data Sets

---

## Null Data Sets

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.

---

## Default Data Sets

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 procedure steps. Issuing the `_LAST_=` system option enables you to avoid specifying the SAS data set name in each procedure statement. The following OPTIONS statement specifies a default SAS data set:

```
options _last_=schedule.january;
```

---

## Automatic Naming Convention

If you do not specify a SAS data set name or the reserved name `_NULL_` in a DATA statement, SAS automatically creates data sets with the names DATA1, DATA2, and so on, to successive data sets in the Work or User library. This feature is referred to as the DATA*n* naming convention. The following statement produces a SAS data set using the DATA*n* naming convention:

```
data;
```

---

# Sorted Data Sets

---

## The Sort Indicator

### What Is a Sort Indicator?

After a data set is sorted, a sort indicator is added to the data set descriptor information. The sort indicator is updated without a permanent sort of the data set by using the SORTEDBY= data set option. The **Sortedby** and **Validated** sort information is updated when the SORTEDBY= data set option is used.

The sort indicator contains some or all of the following sort information of a SAS data set:

- how the data set is sorted by which variable or variables
- whether the order for a variable is descending or ascending
- the character set used for character variables
- the collating sequence used for ordering character data
- collation rules if the data set is sorted linguistically
- whether there is only one observation for any given BY group (use of NODUPKEY option)
- whether there are no adjacent duplicate observations (use of NODUPREC option)
- whether the data set is validated

The sort indicator is set when a data set is sorted by a SORT procedure, an SQL procedure with an ORDER BY clause, a DATASETS procedure MODIFY statement, or a SORTEDBY= data set option. If the SORT or SQL procedures were used to sort the data set, which is being sorted by SAS, the CONTENTS procedure output indicates that the **Validated** sort information is YES. If the SORTEDBY= data set option was used to sort the data set, which is being sorted by the user, the CONTENTS procedure output indicates the **Validated** sort information is set to NO and the **Sortedby** sort information is updated with the variable or variables specified by the data set option.

Data sets can be sorted outside of SAS. In that case, you might use the SORTEDBY= data set option or the DATASETS procedure MODIFY statement to add the sort order to the sort indicator. In this case, they are not validated. For more information, see “[Validating That a Data Set Is Sorted](#)” on page 652.

To view the sort indicator information, use the CONTENTS procedure or the CONTENTS statement in the DATASETS procedure. The following three examples show the sort indicator information in the CONTENTS procedure output.

## Example 1: Using No Sorting

The first example is a data set created without any type of sort:

```
options yearcutoff=1920;
libname myfiles 'C:\My Documents';

data myfiles.sorttest1;
    input priority 1. +1 indate date7.
        +1 office $ code $;
    format indate date7.;
    datalines;
1 03may11 CH J8U
1 21mar11 LA M91
1 01dec11 FW L6R
1 27feb10 FW Q2A
2 15jan11 FW I9U
2 09jul11 CH P3Q
3 08apr10 CH H5T
3 31jan10 FW D2W
;
proc contents data=myfiles.sorttest1;
run;
```

Note that the CONTENTS procedure output indicates there was no sort. SAS did not sort the data set, and the user did not specify that the data is sorted.

*Output 27.1 Contents of Sorttest1 Data Set – No Sorting*

The SAS System			
The CONTENTS Procedure			
Data Set Name	MYFILES.SORTTEST1	Observations	8
Member Type	DATA	Variables	4
Engine	V9	Indexes	0
Created	04/10/2014 12:50:17	Observation Length	32
Last Modified	04/10/2014 12:50:17	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		

## Example 2: Using the SORTEDBY= Data Set Option

In the second example, the data set is created using the SORTEDBY= data set option in the DATA statement.

```
options yearcutoff=1920;
libname myfiles 'C:\My Documents';

data myfiles.sorttest1 (sortedby=priority descending indate);
    input priority 1. +1 indate date7.
        +1 office $ code $;
    format indate date7.;
    datalines;
1 03may01 CH J8U
1 21mar01 LA M91
1 01dec00 FW L6R
1 27feb99 FW Q2A
2 15jan00 FW I9U
2 09jul99 CH P3Q
3 08apr99 CH H5T
3 31jan99 FW D2W
;
proc contents data=myfiles.sorttest1;
run;
```

Note that the CONTENTS procedure output shows that the data set is sorted. Therefore, a **Sort Information** section containing sort indicator information is created. In the **Sort Information** section, the **Sortedby** information indicates the data set is sorted by the PRIORITY variable and is in descending order by the INDATE variable. The data set is sorted using the SORTEDBY= data set option, so the **Validated** information is NO. The **Character Set** information for the data set is ANSI.

*Output 27.2 Contents of Sorttest1 Data Set – Sorted*

The SAS System			
The CONTENTS Procedure			
Data Set Name	MYFILES.SORTTEST1	Observations	8
Member Type	DATA	Variables	4
Engine	V9	Indexes	0
Created	04/10/2014 12:51:54	Observation Length	32
Last Modified	04/10/2014 12:51:54	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		

*Output 27.3 Contents of Sorttest1 Data Set – Sorted*

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Format
4	code	Char	8	
2	indate	Num	8	DATE7.
3	office	Char	8	
1	priority	Num	8	

Sort Information	
Sortedby	priority DESCENDING indate
Validated	NO
Character Set	ANSI

### Example 3: Using the SORT Procedure

In the third example, the data set is sorted using the SORT procedure.

```
options yearcutoff=1920;
libname myfiles 'C:\My Documents';
```

```

data myfiles.sorttest1;
    input priority 1. +1 indate date7.
        +1 office $ code $;
    format indate date7.;
    datalines;
1 03may01 CH J8U
1 21mar01 LA M91
1 01dec00 FW L6R
1 27feb99 FW Q2A
2 15jan00 FW I9U
2 09jul99 CH P3Q
3 08apr99 CH H5T
3 31jan99 FW D2W
;
proc sort data=myfiles.sorttest1;
    by priority descending
    indate;
run;

proc contents data=myfiles.sorttest1;
run;

```

Note that the CONTENTS procedure output shows that the data set is sorted. Therefore, a **Sort Information** section containing sort indicator information is created. In the **Sort Information** section, the **Sortedby** information indicates the data set is sorted by the PRIORITY variable and is in descending order by the INDATE variable. The data set is sorted using the SORT procedure, so the **Validated** information is YES. The **Character Set** information for the data set is ANSI.

*Output 27.4 Contents of Sorttest1 Data Set – Validated Sort*

The SAS System			
The CONTENTS Procedure			
Data Set Name	MYFILES.SORTTEST1	Observations	8
Member Type	DATA	Variables	4
Engine	V9	Indexes	0
Created	04/10/2014 12:54:33	Observation Length	32
Last Modified	04/10/2014 12:54:33	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	YES
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		

**Output 27.5** Contents of Sorttest1 Data Set – Validated Sort

Alphabetic List of Variables and Attributes				
#	Variable	Type	Len	Format
4	code	Char	8	
2	indate	Num	8	DATE7.
3	office	Char	8	
1	priority	Num	8	

Sort Information	
Sortedby	priority DESCENDING indate
Validated	YES
Character Set	ANSI

---

## How SAS Uses the Sort Indicator to Improve Performance

The sort information provided by the sort indicator is used internally for performance improvements. There are several ways to improve performance using the sort indicator:

- SAS uses the sort indicator to validate whether there was a previous sort. If there was a previous sort from a SORT procedure or an SQL procedure with an ORDER BY clause, then SAS does not perform another sort.
  - The SORT procedure sets the sort indicator when a sort occurs. The SORT procedure checks for the sort indicator before it sorts a data set so that data is not re-sorted unnecessarily. For more information, see the SORT Procedure in the *Base SAS Procedures Guide*.
  - The SQL procedure uses the sort indicator to process queries more efficiently and to determine whether an internal sort is necessary before performing a join. For more information, see the SQL procedure in the *Base SAS Procedures Guide*.
- When using the sort indicator during index creation, SAS determines whether the data is already sorted by the key variable or variables in ascending order by checking the sort indicator in the data file. If the values in the sort indicator are in ascending order, SAS does not sort the values for the index file. For more information, see “[Understanding SAS Indexes](#)” on page 692.
- When processing a WHERE expression without an index, SAS first checks the sort indicator. If the **Validated** sort information is YES, SAS stops reading the file once there are no more values that satisfy the WHERE expression.
- If an index is selected for WHERE expression processing, the sort indicator for that data set is changed to reflect the order that is specified by the index.

- For BY-group processing, if the data set is already sorted by the BY variable, SAS does not use the index, even if the data set is indexed on the BY variable.
- If the **Validated** sort information is set to YES, SAS does not need to perform another sort.

## Validating That a Data Set Is Sorted

Any SAS procedure that requires data to be sorted as part of the process checks the sort indicator information. The sort indicator is set when a data set is sorted by a SORT procedure, an SQL procedure with an ORDER BY clause, a DATASETS procedure MODIFY statement, or a SORTEDBY= data set option. If the SORT or SQL procedures were used to sort the data set, the CONTENTS procedure output indicates the **Validated** sort information is YES. If the SORTEDBY= data set option was used to sort the data set, the CONTENTS procedure output indicates the **Validated** sort information is NO. For examples of the CONTENTS procedure output, see “[Example 1: Using No Sorting](#)” on page 647, “[Example 2: Using the SORTEDBY= Data Set Option](#)” on page 648, and “[Example 3: Using the SORT Procedure](#)” on page 649.

You can use the SORTVALIDATE system option to specify whether the SORT procedure validates that a data set is sorted correctly when the data set sort indicator shows that a user has specified that the data set is sorted. The user can specify a sort order by using the SORTEDBY= data set option in a DATA statement or by using the SORTEDBY= option in the DATASETS procedure MODIFY statement. When the sort indicator is set by a user, SAS cannot be absolutely certain that a data set is sorted according to the variables in the BY statement.

If the SORTVALIDATE system option is set and the data set sort indicator was set by a user, the SORT procedure performs a sequence check on each observation 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.

At the end of a successful sequence check or at the end of a sort, the SORT procedure sets the sort indicator **Validated** sort information to YES. If a sort is performed, the SORT procedure also updates the sort indicator **Sortedby** sort information to the variables in the BY statement. If an output data set is specified, the sort indicator **Validated** sort information in the output data set is set to YES. If no sort is necessary, the data set is copied to the output data set. For more information about validated data sets, see the “[SORTVALIDATE System Option](#)” in [SAS System Options: Reference](#).

## Tools for Managing Data Sets

To copy, rename, delete, or obtain information about the contents of SAS data sets, use the same windows, procedures, functions, and options that you do for SAS libraries. For a list of those windows and procedures, see [Chapter 26, “SAS Libraries,” on page 623](#).

There are also functions available that enable you to work with your SAS data set. See each individual function for more complete information.

---

# Viewing and Editing SAS Data Sets

The VIEWTABLE window enables you to browse, edit, or create data sets. This window provides two viewing modes:

**Table View**

uses a tabular format to display multiple observations in the data set.

**Form View**

displays data one observation at a time in a form layout.

You can customize your view of a data set (for example, by sorting your data, changing the color and fonts of columns, displaying variable labels instead of variable names, or removing or adding variables). You can also load an existing DATAFORM catalog entry in order to apply a previously defined variable, data set, and viewer attributes.

To view a data set, select the following: **Tools**  $\Rightarrow$  **Table Editor**. This action brings up VIEWTABLE or FSVIEW (z/OS). You can also double-click on the data set in the Explorer window.

The following SAS files are supported within the VIEWTABLE window:

- SAS data files
- SAS views
- MDDB files

For more information, see the SAS System Help for the VIEWTABLE window in Base SAS.



# SAS Data Files

---

<i>Definition of a SAS Data File</i> .....	656
<i>Differences between Data Files and SAS Views</i> .....	657
<i>Understanding the Observation Count in a SAS Data File</i> .....	658
Definition of the Observation Count .....	658
Backward Compatibility of the Extended Observation Count Attribute .....	658
Interactions with the Extended Observation Count .....	660
Exceeding the Maximum Observation Count .....	660
<i>Understanding an Audit Trail</i> .....	662
Definition of an Audit Trail .....	662
Audit Trail Description .....	662
Defining and Using User Variables .....	664
Operation in a Shared Environment .....	664
Performance Implications .....	664
Preservation by Other Operations .....	665
Programming Considerations .....	665
Other Considerations .....	665
Initiating an Audit Trail .....	665
Controlling the Audit Trail .....	666
Reading and Determining the Status of the Audit Trail .....	666
Audit Trails and CEDA Processing .....	668
Examples of Using Audit Trails .....	668
<i>Understanding Generation Data Sets</i> .....	672
Definition of Generation Data Set .....	672
Terminology for Generation Data Sets .....	672
Invoking Generation Data Sets .....	673
Understanding How a Generation Group Is Maintained .....	673
Processing Specific Versions of a Generation Group .....	675
Managing Generation Groups .....	676
<i>Understanding Integrity Constraints</i> .....	679
Definition of Integrity Constraints .....	679
General and Referential Integrity Constraints .....	679
Preservation of Integrity Constraints .....	681
Indexes and Integrity Constraints .....	682
Locking Integrity Constraints .....	683
Encryption and Integrity Constraints .....	683
Specifying Integrity Constraints .....	683
Specifying Physical Location for Inter-Libref Referential Integrity Constraints When Sharing Disk Space .....	684
Listing Integrity Constraints .....	685
Rejected Observations .....	685

Integrity Constraints and CEDA Processing .....	685
Examples .....	686
<b>Understanding SAS Indexes .....</b>	<b>692</b>
Definition of SAS Indexes .....	692
Benefits of an Index .....	693
The Index File .....	693
Types of Indexes .....	694
Deciding Whether to Create an Index .....	697
Guidelines for Creating Indexes .....	699
Creating an Index .....	700
Using an Index for WHERE Processing .....	702
Using an Index for BY Processing .....	709
Using an Index for Both WHERE and BY Processing .....	710
Specifying an Index with the KEY= Option for SET and MODIFY Statements .....	711
Taking Advantage of an Index .....	711
Procedures and SAS Operations That Maintain Indexes .....	712
<b>Extended Attributes .....</b>	<b>716</b>
Definition .....	716
Enabling and Manipulating Extended Attributes .....	716
<b>Compressing Data Files .....</b>	<b>717</b>
Definition of Compression .....	717
Requesting Compression .....	718
Disabling a Compression Request .....	718

---

## Definition of a SAS Data File

A SAS data file is a type of SAS data set that contains both the data values and the descriptor information. SAS data files have the member type of DATA. There are two general types of SAS data files:

**native SAS data file**

stores the data values and descriptor information in a file that is formatted by SAS.

**interface SAS data file**

stores the data in a file that was formatted by software other than SAS. SAS provides engines for reading and writing data from files that were formatted by software such as ORACLE, DB2, Sybase, ODBC, BMDP, SPSS, and OSIRIS. These files are interface SAS data files, and when their data values are accessed through an engine, SAS recognizes them as SAS data sets.

**Note:** The availability of engines that can access different types of interface data files is determined by your site licensing agreement. See your system administrator to determine which engines are available. For more information about SAS Multi Engine Architecture, see [Chapter 37, “SAS Engines,” on page 803](#).

---

# Differences between Data Files and SAS Views

While the terms *SAS data files* and *SAS views* can often be used interchangeably, there are differences to consider:

The main difference is where the values are stored.

A SAS data file is a type of SAS data set that contains both descriptor information about the data and the data values themselves. SAS views contain only descriptor information and instructions for retrieving data that is stored elsewhere. Once the data is retrieved by SAS, it can be manipulated in a DATA step.

A data file is static. A SAS view is dynamic.

When you reference a data file in a later PROC step, you see the data values as they were when the data file was created or last updated. When you reference a SAS view in a PROC step, the view executes and provides an image of the data values as they currently exist, not as they existed when the view was defined.

SAS data files can be created on tape or on any other storage medium.

SAS views cannot be stored on tape. Because of their dynamic nature, SAS views must derive their information from data files on random-access storage devices, such as disk drives. SAS views cannot derive their information from files stored on sequentially accessed storage devices, such as tape drives.

SAS views are read only.

You cannot write to a SAS view, but some SQL views can be updated.

SAS data files can have an audit trail.

The audit trail is an optional SAS file that logs modifications to a SAS data file. Each time an observation is added, deleted, or updated, information is written to the audit trail about who made the modification, what was modified, and when.

SAS data files can have generations.

Generations provide the ability to keep multiple copies of a SAS data file. The multiple copies represent versions of the same data file, which is archived each time it is replaced.

SAS data files can have integrity constraints.

When you update a SAS data file, you can ensure that the data conforms to certain standards by using integrity constraints. With SAS views, you can assign integrity constraints to the data files that the views reference.

SAS data files can be indexed.

Indexing might enable SAS to find data in a SAS data file more quickly. SAS views cannot be indexed.

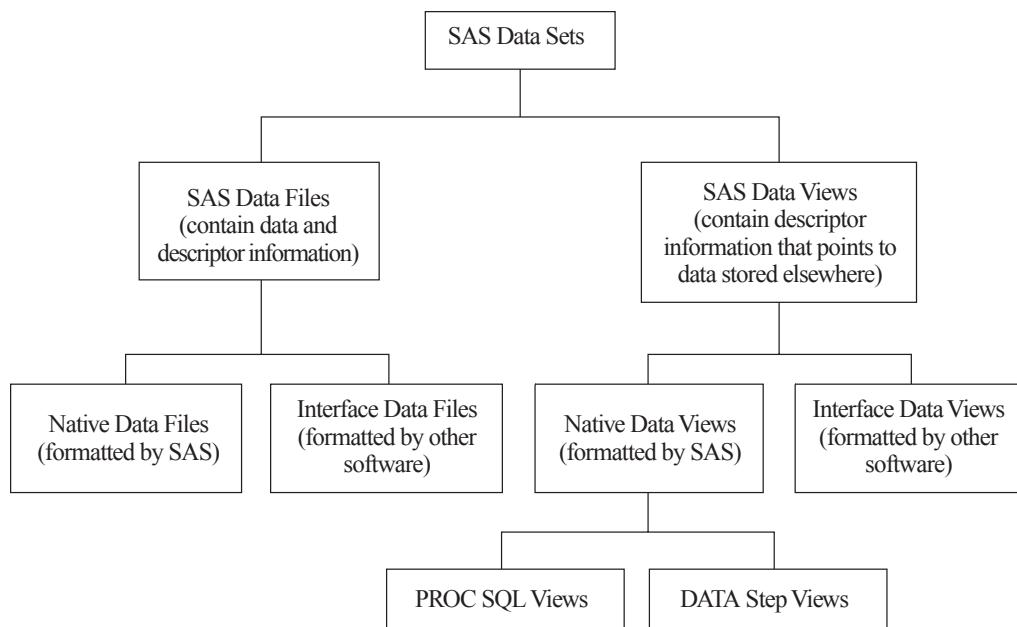
SAS data files can be encrypted.

Encryption provides an extra layer of security to physical files. SAS views cannot be encrypted.

SAS data files can be compressed.

Compression makes it possible to store physical files in less space. SAS views cannot be compressed.

The following figure illustrates native and interface SAS data files and their relationship to SAS views.

**Figure 28.1** Types of SAS Data Sets


---

## Understanding the Observation Count in a SAS Data File

---

### Definition of the Observation Count

The observation count includes both observations (rows) and deleted observations. The maximum number of observations that can be counted for a SAS data file is  $2^{63}-1$  or approximately 9.2 quintillion observations. Exceeding that number is extremely unlikely for most users.

---

### Backward Compatibility of the Extended Observation Count Attribute

#### Overview of the Extended Observation Count Attribute

In SAS 9.2 and earlier releases, the maximum observation count was much lower under some operating environments. In SAS 9.3, an extended observation count was offered to users with the option EXTENDOBSCOUNTER=YES. In SAS 9.4 and later releases, the observation count is extended by default, and SAS has the same maximum observation count under all environments.

The behavior depends on your operating environment:

- Under UNIX environments, by default the EXTENDOBS COUNTER= option is not set. The extended observation count feature is not necessary under 64-bit UNIX. However, if you specify the OUTREP= option, and the data representation is not a 64-bit UNIX operating environment, then SAS automatically sets EXTENDOBS COUNTER=YES. SAS adds the extended observation count feature for compatibility with environments other than UNIX where it might be necessary. (Many customers specify OUTREP= when they create a data set for use in a different environment. This practice can help you avoid the limitations of CEDA processing.)
- Under Windows and z/OS, by default EXTENDOBS COUNTER=YES. Files are created with the enhanced file format and the extended observation count attribute.

## When to Use the EXTENDOBS COUNTER=NO Option

The extended observation count attribute can make data sets unusable in SAS 9.2 and earlier releases, under certain circumstances. Here is an example of the error message:

ERROR: File MYFILES.EXTEND.Data not compatible with this SAS version.

The behavior depends on your operating environment:

- Under UNIX, if you specify OUTREP= and plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBS COUNTER=NO. If you do not specify OUTREP=, then you do not need to specify EXTENDOBS COUNTER=NO.
- Under Windows or z/OS, if you plan to use the file in SAS 9.2 or earlier releases, specify EXTENDOBS COUNTER=NO.

For more information, see the following:

- The EXTENDOBS COUNTER= data set option is documented in [SAS Data Set Options: Reference](#).
- The EXTENDOBS COUNTER= LIBNAME statement option is documented in [SAS Global Statements: Reference](#).
- The EXTENDOBS COUNTER= system option is documented in [SAS System Options: Reference](#).

## Viewing the ExtendObsCounter Attribute in CONTENTS Output

When you run PROC CONTENTS or the CONTENTS statement of PROC DATASETS, you can see the **ExtendObsCounter** attribute in the output. If a SAS data file does not contain the extended observation count file attribute, the **ExtendObsCounter** field is not listed.

**Output 28.1** CONTENTS Procedure Output Showing ExtendObsCounter Attribute

Engine/Host Dependent Information	
Data Set Page Size	65536
Number of Data Set Pages	1
First Data Page	1
Max Obs per Page	2039
Obs in First Data Page	2
Number of Data Set Repairs	0
ExtendObsCounter	YES
Filename	C:\My Documents\bigfile.sas7bdat
Release Created	9.0401M0
Host Created	X64_7PRO

---

## Interactions with the Extended Observation Count

Note the following details:

- SAS functionality that copies files (such as the APPEND procedure, COPY procedure, MIGRATE procedure, and SET statement) does not copy the extended observation count attribute.
- In a SAS/SERVE client session, the EXTENDOBSCOUNTER= option in the LIBNAME statement is ignored if it is specified in combination with the SERVER= option.

---

## Exceeding the Maximum Observation Count

### SAS Processing When the Maximum Observation Count Is Reached

The observation count includes both observations (rows) and deleted observations. The maximum number of observations that can be counted for a SAS data file is  $2^{63}-1$  or approximately 9.2 quintillion observations. When a SAS data file reaches the maximum observation count, continued SAS processing depends on whether the file has an index or an integrity constraint that uses an index.

- If the SAS data file has an index or an integrity constraint that uses an index (unique key, primary key, and foreign key), an error message is issued when an operation reaches the maximum observation count.

A SAS data file is not damaged when an operation attempts to exceed the maximum observation count. However, you must take explicit action to continue processing the file.

- If the SAS data file does not have an index or an integrity constraint that uses an index, sequential processing continues and additional observations are accepted. However, the file cannot store the observation count and does not maintain the observation numbers. Any operation that requires an observation number is not available. There are no messages to indicate that the file has reached or exceeded the maximum observation count.

The following list describes some of the operations and features that are limited for a SAS data file that exceeds the maximum observation count and does not have an index or an integrity constraint that uses an index. For a complete list, contact SAS Technical Support.

- SAS procedures that return an observation count (such as the PRINT procedure or the CONTENTS procedure) return a missing value, which is represented by a period (.), for the number of observations.
- SAS procedures that depend on the observation count (for example, the SORT procedure or the COMPARE procedure) can return unpredictable results.
- Operations that update the observation count cannot be submitted. You cannot reset the observation count by deleting observations.
- When you request to compress a file for which the observation count is no longer maintained, the compression percentage cannot be calculated.
- You cannot create an index or an integrity constraint.

## Recovering from an Exceeded Maximum Observation Count

If a SAS data file has reached or exceeded the maximum number of observations that can be counted and the file has an index or an integrity constraint that uses an index, then you must take explicit action to continue processing. You can delete the index or the integrity constraint and continue processing. However, because the file exceeds the maximum observation count, you have limited functionality. You can use the DATASETS procedure or the SQL procedure to delete indexes and integrity constraints. See the [Base SAS Procedures Guide](#) or the [SAS SQL Procedure User's Guide](#).

If the file does not have an index or an integrity constraint that uses an index, there are no messages to indicate that the file has reached or exceeded the maximum observation count. However, the file has limited functionality.

Here are some ways to recover from an exceeded maximum observation count:

- If the file was created in SAS 9.2 or earlier, the observation count might have been limited. Try re-creating the file in the current release to increase the maximum observation count.
- Deleted observations are included in the total observation count. If the data file has deleted observations, try re-creating it. Use a method that does not retain deleted observations, such as the COPY procedure or the DATA step with a SET statement. (The MIGRATE procedure retains deleted observations.)

# Understanding an Audit Trail

## Definition of an Audit Trail

The audit trail is an optional SAS file that you can create in order to log modifications to a SAS data file. Each time an observation is added, deleted, or updated, information is written to the audit trail about who made the modification, what was modified, and when.

Many businesses and organizations require an audit trail for security reasons. The audit trail maintains historical information about the data, which gives you the opportunity to develop usage statistics and patterns. The historical information enables you to track individual pieces of data from the moment that they enter the data file to the time they leave.

The audit trail is also the only facility in SAS that stores observations from failed Append operations and that were rejected by integrity constraints. (The integrity constraints feature is described in “[Understanding Integrity Constraints](#)” on page 679.) The audit trail enables you to write a DATA step to extract the failed or rejected observations, use information describing why the observations failed to correct them, and then reapply the observations to the data file.

## Audit Trail Description

The audit trail is created by the default Base SAS engine and has the same libref and member name as the data file, but has a type of AUDIT. It replicates the variables in the data file and also stores two types of audit variables:

- `_AT*` variables, which automatically store modification data
- user variables, which are optional variables that you can define to collect modification data

The `_AT*` variables are described in the following table.

**Table 28.1 `_AT*` Variables**

<code>_AT*</code> Variable	Description
<code>_ATDATETIME_</code>	Stores the date and time of a modification
<code>_ATUSERID_</code>	Stores the logon user ID that is associated with a modification
<code>_ATOBSNO_</code>	Stores the observation number that is affected by the modification, except when REUSE=YES (because the observation number is always 0)

<u>_AT*</u> _ Variable	Description
<u>_ATRETURNCODE_</u>	Stores the event return code
<u>_ATMESSAGE_</u>	Stores the SAS log message at the time of the modification
<u>_ATOPCODE_</u>	Stores a code that describes the type of modification

The \_ATOPCODE\_ values are listed in the following table.

*Table 28.2* \_ATOPCODE\_ Values

Code	Modification
AL	Auditing is resumed
AS	Auditing is suspended
DA	Added data record image
DD	Deleted data record image
DR	Before-update record image
DW	After-update record image
EA	Observation add failed
ED	Observation delete failed
EU	Observation update failed

The type of entries stored in the audit trail, along with their corresponding \_ATOPCODE\_ values, are determined by the options specified in the LOG statement when the audit trail is initiated. Note that if the LOG statement is omitted when the audit trail is initiated, the default behavior is to log all images.

- The A operation codes are controlled by the ADMIN\_IMAGE option.
- The DR operation code is controlled by the BEFORE\_IMAGE option.
- All other D operation codes are controlled with the DATA\_IMAGE option.
- The E operation codes are controlled by the ERROR\_IMAGE option.

The user variable is a variable that associates data values with the data file without making them part of the data file. That is, the data values are stored in the audit file, but you update them in the data file like any other variable. You might want to define a user variable to enable end users to enter a reason for each update.

A data file can have one audit file, and the audit file must reside in the same SAS library as the data file.

---

## Defining and Using User Variables

User variables are defined at audit trail initiation with the USER\_VAR statement. For example, the following code initiates an audit trail and creates a user variable Reason\_Code for data file MyLib.Sales:

```
proc datasets lib=mylib;
  audit sales;
  initiate;
  user_var reason_code $ 20;
run;
```

If you define user variables, you must store values in them in order for the variables to be meaningful. Programmatically, you can enter data values for the user variables as you would for any data variable. See “[Example of a Data File Update](#)” on page [669](#). The data values are saved to the audit trail as each observation is saved.

User variables cannot be displayed or updated in an interactive window except in the FSEDIT window of SAS/FSP software. To view the audit variables, use the TYPE=AUDIT data set option to print the audit file.

However, to rename a user variable or modify its attributes, you modify the data file, not the audit file. The following example uses PROC DATASETS to rename the user variable:

```
proc datasets lib=mylib;
  modify sales;
  rename reason_code = Reason;
  run;
quit;
```

You must also define attributes such as format and informat in the data file with PROC DATASETS.

---

## Operation in a Shared Environment

The audit trail operates similarly in local and remote environments. The only difference for applications and users networking with SAS/CONNECT and SASSHARE is that the audit trail logs events when the observation is written to permanent storage. That is, when the data is written to the remote SAS session or server. Therefore, the time that the transaction is logged might be different from the user's SAS session.

---

## Performance Implications

Because each update to the data file is also written to the audit file, the audit trail can negatively impact system performance. You might want to consider suspending the audit trail for large, regularly scheduled batch updates. Note that the audit variables are unavailable when the audit trail is suspended.

---

## Preservation by Other Operations

The audit trail is not recommended for data files that are copied, moved, sorted in place, replaced, or transferred to another operating environment. Those operations do not preserve the audit trail. In a Copy operation on the same host, you can preserve the data file and audit trail by renaming them using the generation data sets feature. However, logging stops because neither the auditing process nor the generation data sets feature saves the source program that caused the replacement. For more information about generation data sets, see “[Understanding Generation Data Sets](#)” on page 672.

---

## Programming Considerations

For data files whose audit file contains user variables, the variable list is different when browsing and updating the data file. The user variables are selected for update but not for browsing. You should be aware of this difference when you are developing your own full-screen applications.

---

## Other Considerations

Data values that are entered for user variables are not stored in the audit trail for Delete operations.

If the audit file becomes damaged, you cannot process the data file until you terminate the audit trail. Then you can initiate a new audit trail or process the data file without one. To terminate the audit trail for a generation data set, use the GENNUM= data set option in the AUDIT statement. You cannot initiate an audit trail for a generation data set.

In indexed data sets, the fast-append feature can cause some observations to be written to the audit trail twice, first with a DA operation code and then with an EA operation code. The observations with EA represent the observations rejected by index restrictions. For more information, see “[Appending to an Indexed Data Set — Fast-Append Method](#)” in *Base SAS Procedures Guide*.

---

## Initiating an Audit Trail

You initiate an audit trail in the DATASETS procedure with the AUDIT statement. For syntax information, see “[DATASETS Procedure](#)” in *Base SAS Procedures Guide*.

The audit file uses the SAS password assigned to its associated data file. Therefore, it is recommended that the data file have an ALTER password. An ALTER-level password restricts Read and Edit access to SAS files. If a password other than ALTER is used, or no password is used, the software generates a warning message that the files are not protected from accidental update or deletion.

**Note:** The initiation of an audit trail is only possible with the Base SAS engine.

---

## Controlling the Audit Trail

Once active, you can suspend and resume logging, and terminate (delete) the audit trail. The syntax for controlling the audit trail is described in the PROC DATASETS AUDIT statement documentation. Note that replacing the associated data file also deletes the audit trail.

---

## Reading and Determining the Status of the Audit Trail

The audit trail is read-only. You can read the audit trail with any component of SAS that reads a data set. To refer to the audit trail, use the TYPE= data set option. For example, issue the following statement to view the contents of the audit trail. Note that the parentheses around the TYPE= option are required.

```
proc contents data=mylib.sales (type=audit);  
run;
```

The CONTENTS procedure output is shown below. Notice that the output contains all of the variables from the corresponding data file, the \_AT\*\_\* variables, and the user variable.

**Output 28.2 PROC CONTENTS Output for Data File MyLib.Sales**

The SAS System			
The CONTENTS Procedure			
Data Set Name	MYLIB.SALES.AUDIT	Observations	10
Member Type	AUDIT	Variables	10
Engine	V9	Indexes	0
Created	08/17/2012 10:31:41	Observation Length	283
Last Modified	08/17/2012 10:35:10	Deleted Observations	0
Protection		Compressed	NO
Data Set Type	AUDIT	Sorted	NO
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		
Engine/Host Dependent Information			
Data Set Page Size	4096		
Number of Data Set Pages	1		
First Data Page	1		
Max Obs per Page	14		
Obs in First Data Page	10		
Number of Data Set Repairs	0		
Filename	C:\My Documents\sales.sas7baud		
Release Created	9.0401B0		
Host Created	W32_7PRO		
Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
5	_ATDATETIME_	Num	8
10	_ATMESSAGE_	Char	160
6	_ATOBSNO_	Num	8
9	_ATOPCODE_	Char	2
7	_ATRETURNCODE_	Num	8
8	_ATUSERID_	Char	32
2	invoice	Num	8
1	product	Char	9
4	reason_code	Char	20
3	renewal	Num	8

You can also use your favorite reporting tool, such as PROC REPORT or PROC TABULATE, on the audit trail.

## Audit Trails and CEDA Processing

When a SAS data file requires processing with CEDA, audit trails are not supported. For example, CEDA translates the file for you if you transfer a SAS data file with an initiated audit trail from one operating environment such as Windows to a different operating environment such as UNIX. However, the audit trail is not available. For information about CEDA processing, see [Chapter 34, “Processing Data Using Cross-Environment Data Access \(CEDA\),” on page 765](#).

The MIGRATE procedure retains all deleted observations in migrated data sets. Therefore, PROC MIGRATE preserves and migrates audit trails. For more information, see “[MIGRATE Procedure](#)” in [Base SAS Procedures Guide](#).

In contrast, conversion procedures such as PROC CPOR and PROC CIMPORT clean up data sets and restructure the data sets. For example, these procedures remove deleted observations to recover disk space. The restructuring is advantageous but results in a data set that is not historically accurate when trying to track changes through an audit trail. Because these conversion procedures do not keep deleted observations, the audit trails cannot be copied using these procedures. For more information, see “[CPOR Procedure](#)” in [Base SAS Procedures Guide](#) and “[CIMPORT Procedure](#)” in [Base SAS Procedures Guide](#).

**CAUTION!** If your data files contain audit trails, do not use your operating environment commands to copy, move, or delete your data files.

---

## Examples of Using Audit Trails

### Example of Initiating an Audit Trail

The following example shows the data and code that are used to create and initiate an audit trail for the data file MyLib.Sales that is used in earlier examples in this section. MyLib.Sales stores fictional invoice and renewal figures for SAS products. The audit trail records all events and stores one user variable, Reason\_Code, for users to enter a reason for the update.

Subsequent examples illustrate the effect of a data file update on the audit trail and how to use audit variables to capture observations that are rejected by integrity constraints.

```
libname mylib 'C:\My Documents';
/*-----*/
/* Create SALES data set. */
/*-----*/
data mylib.sales;
length product $9;
input product invoice renewal;
datalines;
FSP      1270.00      570
SAS      1650.00      850
```

```

STAT      570.00      0
STAT      970.82     600
OR       239.36      0
SAS       7478.71    1100
SAS      800.00     800
;

/*-----
/* Create an audit trail with a      */
/* user variable.                  */
/*-----*/
proc datasets lib=mylib nolist;
  audit sales;
  initiate;
  user_var reason_code $ 20;
quit;

```

## Example of a Data File Update

The following example inserts an observation into MyLib.Sales.Data and prints the update data in the MyLib.Sales.Audit.

```

/*-----*/
/* Do an update.                      */
/*-----*/
proc sql;
  insert into mylib.sales
    set product = 'AUDIT',
        invoice = 2000,
        renewal = 970,
        reason_code = "Add new product";
quit;

/*-----*/
/* Print the audit trail. */
/*-----*/
proc sql;
  select product,
         reason_code,
         _atopcode_,
         _atdatetime_
    from mylib.sales(type=audit);
quit;

```

*Output 28.3 Updated Data in MyLib.Sales.Audit*

The SAS System			
product	reason_code	_ATOPCODE_	_ATDATETIME_
AUDIT	Add new product	DA	10APR2014:14:35:23

## Example of Using the Audit Trail to Capture Rejected Observations

The following example adds integrity constraints to MyLib.Sales.Data and records observations that are rejected as a result of the integrity constraints in MyLib.Sales.Audit. For more information about integrity constraints, see “[Understanding Integrity Constraints](#)” on page 679.

```

/*-----*/
/* Create integrity constraints.      */
/*-----*/
proc datasets lib=mylib;
  modify sales;
  ic create null_renewal = not null (invoice)
    message = "Invoice must have a value.";
  ic create invoice_amt = check (where=((invoice > 0) and
    (renewal <= invoice)))
    message = "Invoice and/or renewal are invalid.";
run;

/*-----*/
/* Do some updates.                  */
/*-----*/
proc sql; /* this update works */
update mylib.sales
  set invoice = invoice * .9,
  reason_code = "10% price cut"
  where renewal > 800;

proc sql; /* this update fails */
insert into mylib.sales
  set product = 'AUDIT',
  renewal = 970,
  reason_code = "Add new product";

proc sql; /* this update works */
insert into mylib.sales
  set product = 'AUDIT',
  invoice = 10000,
  renewal = 970,
  reason_code = "Add new product";

proc sql; /* this update fails */
insert into mylib.sales
  set product = 'AUDIT',
  invoice = 100,
  renewal = 970,
  reason_code = "Add new product";
quit;

/*-----*/
/* Print the audit trail. */
/*-----*/
proc print data=mylib.sales(type=audit);

```

```

format _atuserid_ $6.;
var product reason_code _atopcode_ _atdatetime_;
title 'Contents of the Audit Trail';
run;

/*-----*/
/* Print the rejected records. */
/*-----*/
proc print data=mylib.sales(type=audit);
where _atopcode_ eq "EA";
format _atmessage_ $250.;
var product invoice renewal _atmessage_;
title 'Rejected Records';
run;

```

The following output shows the contents of MyLib.Sales.Audit after several updates of MyLib.Sales.Data were attempted. Integrity constraints were added to the file, and then updates were attempted.

*Output 28.4 Contents of MyLib.Sales.Audit after an Update with Integrity Constraints*

<b>Contents of the Audit Trail</b>				
<b>Obs</b>	<b>product</b>	<b>reason_code</b>	<b>_ATOPCODE_</b>	<b>_ATDATETIME_</b>
1	AUDIT	Add new product	DA	10APR2014:14:35:23
2	SAS		DR	10APR2014:14:37:13
3	SAS	10% price cut	DW	10APR2014:14:37:13
4	SAS		DR	10APR2014:14:37:13
5	SAS	10% price cut	DW	10APR2014:14:37:13
6	AUDIT		DR	10APR2014:14:37:13
7	AUDIT	10% price cut	DW	10APR2014:14:37:13
8	AUDIT	Add new product	EA	10APR2014:14:37:13
9	AUDIT	Add new product	DA	10APR2014:14:37:13
10	AUDIT	Add new product	EA	10APR2014:14:37:13

This output prints information about the rejected observations on the audit trail.

**Output 28.5 Rejected Records on the Audit Trail**

Rejected Records				
Obs	product	invoice	renewal	_ATMESSAGE_
1	AUDIT	-	970	ERROR: Invoice must have a value. Add/Update failed for data set MYLIB.SALES because data value(s) do not comply with integrity constraint null_renewal.
2	AUDIT	100	970	ERROR: Invoice and/or renewal are invalid. Add/Update failed for data set MYLIB.SALES because data value(s) do not comply with integrity constraint invoice_amt.

---

## Understanding Generation Data Sets

---

### Definition of Generation Data Set

A generation data set is an archived version of a SAS data set that is stored as part of a generation group. A generation data set is created each time the file is replaced. Each generation data set in a generation group has the same root member name, but each has a different version number. The most recent version of the generation data set is called the base version.

You can request generations for a SAS data file only. You cannot request generations for a SAS view.

**Note:** Generation data sets provide historical versions of a data set; they do not track observation updates for an individual data set. To log each time an observation is added, deleted, or updated, see “[Understanding an Audit Trail](#)” on page 662.

**CAUTION! Do not use operating system tools when managing generation data sets.** This can cause limited access to the generation group files.

---

### Terminology for Generation Data Sets

The following terms are relevant to generation data sets:

**base version**

is the most recently created version of a data set. Its name does not have the four-character suffix for the generation number.

**generation group**

is a group of data sets that represent a series of replacements to the original data set. The generation group consists of the base version and a set of historical versions.

**generation number**

is a monotonically increasing number that identifies one of the historical versions in a generation group. For example, the data set named Air#272 has a generation number of 272.

**GENMAX=**

is an output data set option that requests generations for a data set and specifies the maximum number of versions (including the base version and all historical versions) to keep for a given data set. The default is GENMAX=0, which means that the generation data sets feature is not in effect.

**GENNUM=**

is an input data set option that references a specific version from a generation group. Positive numbers are absolute references to a historical version by its generation number. Negative numbers are a relative reference to historical versions. For example, GENNUM=-1 refers to the youngest version.

**historical versions**

are the older copies of the base version of a data set. Names for historical versions have a four-character suffix for the generation number, such as #003.

**oldest version**

is the oldest version in a generation group.

**rolling over**

specifies the process of the version number moving from 999 to 000. When the generation number reaches 999, its next value is 000.

**youngest version**

is the version that is chronologically closest to the base version.

## Invoking Generation Data Sets

To invoke generation data sets and to specify the maximum number of versions to maintain, include the output data set option GENMAX= when creating or replacing a data set. For example, the following DATA step creates a new data set and requests that up to four copies be kept (one base version and three historical versions):

```
data (genmax=4);
  x=1;
  output;
run;
```

Once the GENMAX= data set option is in effect, the data set member name is limited to 28 characters (rather than 32). This happens because the last four characters are reserved for a version number. When the GENMAX= data set option is not in effect, the member name can be up to 32 characters. See the GENMAX= data set option in [SAS Data Set Options: Reference](#).

## Understanding How a Generation Group Is Maintained

The first time a data set with generations in effect is replaced, SAS keeps the replaced data set, and appends a four-character version number to its member name, which includes # and a three-digit number. That is, for a data set named A, the replaced data set becomes A#001. When the data set is replaced for the second time, the replaced data set becomes A#002. That is, A#002 is the version that is chronologically closest to the base version. After three replacements, the result is:

A

base (current) version

A#003  
     most recent (youngest) historical version

A#002  
     second most recent historical version

A#001  
     oldest historical version

With GENMAX=4, a fourth replacement deletes the oldest version, which is A#001. As replacements occur, SAS always keeps four copies. For example, after ten replacements, the result is:

A  
     base (current) version  
 A#010  
     most recent (youngest) historical version  
 A#009  
     2nd most recent historical version  
 A#008  
     oldest historical version

The limit for version numbers that SAS can append is #999. After 999 replacements, the youngest version is #999. After 1,000 replacements, SAS rolls over the youngest version number to #000. After 1,001 replacements, the youngest version number is #001. For example, using data set A with GENNUM=4, the results would be:

999 replacements  
     ■ A (current)  
     ■ A#999 (most recent)  
     ■ A#998 (2nd most recent)  
     ■ A#997 (oldest)

1,000 replacements  
     ■ A (current)  
     ■ A#000 (most recent)  
     ■ A#999 (2nd most recent)  
     ■ A#998 (oldest)

1,001 replacements  
     ■ A (current)  
     ■ A#001 (most recent)  
     ■ A#000 (2nd most recent)  
     ■ A#999 (oldest)

The following table shows how names are assigned to a generation group:

**Table 28.3** Naming Generation Group Data Sets

Time	SAS Code	Data Set Names	GENNUM= Absolute Reference	GENNUM= Relative Reference	Explanation
1	data air (genmax=3);	Air	1	0	The Air data set is created, and three generations are requested.
2	data air;	Air Air#001	2 1	0 -1	Air is replaced. Air from time 1 is renamed Air#001.
3	data air;	Air Air#002 Air#001	3 2 1	0 -1 -2	Air is replaced. Air from time 2 is renamed Air#002.
4	data air;	Air Air#003 Air#002	4 3 2	0 -1 -2	Air is replaced. Air from time 3 is renamed Air#003. Air#001 from time 1, which is the oldest, is deleted.
5	data air (genmax=2);	Air Air#004	5 4	0 -1	Air is replaced, and the number of generations is changed to two. Air from time 4 is renamed Air#004. The two oldest versions are deleted.

## Processing Specific Versions of a Generation Group

When a generation group exists, SAS processes the base version by default. For example, the following PRINT procedure prints the base version:

```
proc print data=a;
run;
```

To request a specific version from a generation group, use the GENNUM= input data set option. There are two methods that you can use:

- A positive integer (excluding zero) references a specific historical version number. For example, the following statement prints the historical version #003:

```
proc print data=a(gennum=3);
run;
```

**Note:** After 1,000 replacements, if you want historical version #000, specify GENNUM=1000.

- A negative integer is a relative reference to a version in relation to the base version, from the youngest predecessor to the oldest. For example, GENNUM=-1 refers to the youngest version. The following statement prints the data set that is three versions previous to the base version:

```
proc print data=a(gennum=-3);
run;
```

**Table 28.4 Requesting Specific Generation Data Sets**

SAS Statement	Result
proc print data=air (gennum=0); proc print data=air;	Prints the current (base) version of the Air data set.
proc print data=air (gennum=-2);	Prints the version two generations back from the current version.
proc print data=air (gennum=3);	Prints the file Air#003.
proc print data=air (gennum=1000);	After 1,000 replacements, prints the file Air#000, which is the file that is created after Air#999.

## Managing Generation Groups

### Introduction

The DATASETS procedure provides a variety of statements for managing generation groups. Note that for the DATASETS procedure, GENNUM= has the following additional functionality:

- For the PROC DATASETS and DELETE statements, GENNUM= supports the additional values ALL, HIST, and REVERT.
- For the CHANGE statement, GENNUM= supports the additional value ALL.
- For the CHANGE statement, specifying GENNUM=0 refers to all versions rather than just the base version.

**CAUTION! Do not use operating system tools when managing generation data sets.** This can cause limited access to the generation group files. Instead, use SAS tools such as the DATASETS or COPY procedure.

### Displaying Data Set Information

A variety of statements in the DATASETS procedure can process a specific historical version. For example, you can display data set version numbers for historical copies using the CONTENTS statement in PROC DATASETS:

```
proc datasets library=myfiles;
  contents data=test (gennum=2);
run;
```

### Copying Generation Groups

You can use the COPY statement in the DATASETS procedure or the COPY procedure to copy a generation group. However, you cannot copy an individual version.

For example, the following DATASETS procedure uses the COPY statement to copy a generation group for data set MyGen1 from library MyLib1 to library MyLib2.

```
libname mylib1 'SAS-library-1';
libname mylib2 'SAS-library-2';

proc datasets;
  copy in=mylib1 out=mylib2;
  select mygen1;
run;
```

## Appending Generation Groups

You can use the GENNUM= data set option to append a specific historical version. For example, the following DATASETS procedure uses the APPEND statement to append a historical version of data set B to data set A. Note that by default, SAS uses the base version for the BASE= data set.

```
proc datasets;
  append base=a data=b(gennum=2);
run;
```

## Modifying the Number of Versions

When you modify the attributes of a data set, you can increase or decrease the number of versions for an existing generation group.

For example, the following MODIFY statement in the DATASETS procedure changes the number of generations for data set MyLib.Air to 4:

```
libname mylib 'SAS-library';

proc datasets library=mylib;
  modify air(genmax=4);
run;
```

**CAUTION!** If you decrease the number of versions, SAS deletes the oldest version or versions so as not to exceed the new maximum. For example, the following MODIFY statement decreases the number of versions for MyLib.Air from 4 to 0. This decrease causes SAS to automatically delete the three historical versions:

```
proc datasets library=mylib;
  modify air (genmax=0);
run;
```

## Deleting Versions in a Generation Group

When you delete data sets, you can specify a specific version or an entire generation group to delete. The following table shows the types of Delete operations and their effects when you delete versions of a generation group.

The following examples assume that the base version of Air and two historical versions (Air#001 and Air#002) exist for each command.

**Table 28.5** Deleting Generation Data Sets

SAS Statement in PROC DATASETS	Results
<code>delete air;</code> <code>delete air(gennum=0);</code>	Deletes the base version and shifts up historical versions. Air#002 is renamed to Air and becomes the new base version.
<code>delete air(gennum=2);</code>	Deletes historical version Air#002.
<code>delete air(gennum=-2);</code>	Deletes the second youngest historical version (Air#001).
<code>delete air(gennum=all);</code>	Deletes all data sets in the generation group, including the base version.
<code>delete air(gennum=hist);</code>	Deletes all data sets in the generation group, except the base version.

**Note:** Both an absolute reference and a relative reference refer to a specific version. A relative reference does not skip deleted versions. Therefore, when you are working with a generation group that includes one or more deleted versions, using a relative reference results in an error if the referenced version has been deleted. For example, if you have the base version Air and three historical versions (Air#001, Air#002, and Air#003) and you delete Air#002, the following statements return an error, because Air#002 does not exist. SAS does not assume that you mean Air#003:

```
proc print data=air (gennum= -2);
run;
```

## Renaming Versions in a Generation Group

When you rename a data set, you can rename an entire generation group:

```
proc datasets;
  change a=newa;
run;
```

You can also rename a single version by including GENNUM=:

```
proc datasets;
  change a (gennum=2)=newa;
```

**Note:** For the CHANGE statement in PROC DATASETS, specifying GENNUM=0 refers to the entire generation group.

## Using Passwords in a Generation Group

Passwords for versions in a generation group are maintained as follows:

- If you assign a password to the base version, the password is maintained in subsequent historical versions. However, the password is not applied to any existing historical versions.
- If you assign a password to a historical version, the password applies to that individual data set only.

---

# Understanding Integrity Constraints

---

## Definition of Integrity Constraints

Integrity constraints are a set of data validation rules that you can specify in order to restrict the data values that can be stored for a variable in a SAS data file. Integrity constraints help you preserve the validity and consistency of your data. SAS enforces the integrity constraints when the values associated with a variable are added, updated, or deleted.

There are two categories of integrity constraints: general and referential.

**CAUTION! Do not use operating system tools when managing integrity constraints.** This can cause your data set to become damaged. Instead, use SAS tools such as the DATASETS procedure or the SQL procedure.

---

## General and Referential Integrity Constraints

---

### General Integrity Constraints

General integrity constraints enable you to restrict the values of variables within a single file. There are four types of general constraints:

check

limits the data values of variables to a specific set, range, or list of values. Check constraints can also be used to ensure that the data values in one variable within an observation are contingent on the data values of another variable in the same observation.

not null

requires that a variable contain a data value. Null (missing) values are not allowed.

unique

requires that the specified variable or variables contain unique data values. A null data value is allowed but is limited to a single instance.

primary key

requires that the specified variable or variables contain unique data values and that null data values are not allowed. Only one primary key can exist in a data file.

**Note:** A primary key is a general integrity constraint if it does not have any foreign key constraints referencing it.

## Referential Integrity Constraints

A referential integrity constraint is created when a primary key integrity constraint in one data file is referenced by a foreign key integrity constraint in another data file.

The foreign key constraint links the data values of one or more variables in the foreign key data file, to corresponding variables and values in the primary key data file. Data values in the foreign key data file must have a matching value in the primary key data file, or they must be null. When data is updated or deleted in the primary key data file, the modifications are controlled by a referential action that is defined as part of the foreign key constraint.

Separate referential actions can be defined for the Update and Delete operations. There are three types of referential actions:

**restrict**

prevents the data values of the primary key variables from being updated or deleted if there is a matching value in one of the foreign key data file's corresponding foreign key variables. The restrict type of action is the default action if one is not specified.

**set null**

enables the data values of the primary key variables to be updated or deleted, but matching data values in the foreign key data files are changed to null (missing) values.

**cascade**

enables the data values in the primary key variables to be updated, and also updates matching data values in the foreign key data files to the same value. The cascade type of action is supported only for Update operations.

The requirements for establishing a referential relationship are as follows:

- The primary key and foreign key must reference the same number of variables, and the variables must be in the same order.
- The variables must be of the same type (character or numeric) and length.
- If the foreign key is being added to a data file that already contains data, the data values in the foreign key data file must either match existing values in the primary key data file, or the values must be null.

The foreign key data file can exist in the same SAS library as the referenced primary key data file (intra-libref), or in a different SAS library (inter-libref). However, if the library that contains the foreign key data file is temporary, the library that contains the primary key data file must be temporary as well. In addition, referential integrity constraints cannot be assigned to data files in concatenated libraries.

There is no limit to the number of foreign keys that can reference a primary key. However, additional foreign keys can adversely impact the performance of Update and Delete operations.

When a referential constraint exists, a primary key integrity constraint is not deleted until all foreign keys that reference it are deleted. There are no restrictions on deleting foreign keys.

## Overlapping Primary Key and Foreign Key Constraints

Variables in a SAS data file can be part of both a primary key (general integrity constraint) and a foreign key (referential integrity constraint). However, there are

restrictions when you define a primary key and a foreign key constraint that use the same variables:

- The foreign key's update and delete referential actions must both be RESTRICT.
- When the same variables are used in a primary key and foreign key definition, the variables must be defined in a different order.

For an example, see [“Defining Overlapping Primary Key and Foreign Key Constraints” on page 691](#).

---

## Preservation of Integrity Constraints

These procedures preserve integrity constraints when their operation results in a copy of the original data file:

- in Base SAS software, the APPEND, COPY, CPRT, CIMPORT, MIGRATE, and SORT procedures
- in SAS/CONNECT software, the UPLOAD and DOWNLOAD procedures
- PROC APPEND
  - for an existing BASE= data file, integrity constraints in the BASE= file are preserved, but integrity constraints in the DATA= file that is being appended to the BASE= file are not preserved.
  - for a non-existent BASE= data file, general integrity constraints in the DATA= file that is being appended to the new BASE= file are preserved. Referential constraints in the DATA= file are not preserved.
- PROC SORT, PROC UPLOAD, and PROC DOWNLOAD, when an OUT= data file is not specified
- the SAS Explorer window

You can also use the CONSTRAINT= option to control whether integrity constraints are preserved for the COPY, CPRT, CIMPORT, UPLOAD, and DOWNLOAD procedures.

General integrity constraints are preserved in an active state. The state in which referential constraints are preserved depends on whether the procedure causes the primary key and foreign key data files to be written to the same or different SAS libraries (intra-libref versus inter-libref integrity constraints). Intra-libref constraints are preserved in an active state. Inter-libref constraints are preserved in an inactive state. That is, the primary key portion of the integrity constraint is enforced as a general integrity constraint but the foreign key portion is inactive. You must use the DATASETS procedure statement IC REACTIVATE to reactivate the inactive foreign keys.

The following table summarizes the circumstances under which integrity constraints are preserved.

**Table 28.6** Circumstances That Cause Integrity Constraints to Be Preserved

Procedure	Condition	Constraints That Are Preserved
APPEND	DATA= data set does not exist	General constraints Referential constraints are not affected
COPY	CONSTRAINT=yes	General constraints Intra-libref constraints are referential in an active state Inter-libref constraints are referential in an inactive state
CPORT/CIMPORT	CONSTRAINT=yes	General constraints Intra-libref constraints are referential in an active state Inter-libref constraints are referential in an inactive state
SORT	OUT= data set is not specified	General constraints Referential constraints are not affected
UPLOAD/DOWNLOAD	CONSTRAINT=yes and OUT= data set is not specified	General constraints Intra-libref constraints are referential in an active state Inter-libref constraints are referential in an inactive state
SAS Explorer window		General constraints

**CAUTION! Do not use operating system tools when managing integrity constraints.** This can cause your data set to become damaged. Instead, use SAS tools such as the DATASETS procedure or the SQL procedure.

## Indexes and Integrity Constraints

The unique, primary key, and foreign key integrity constraints store data values in an index file. If an index file already exists, it is used. Otherwise, one is created.

Consider the following points when you create or delete an integrity constraint:

- When a user-defined index exists, the index's attributes must be compatible with the integrity constraint in order for the integrity constraint to be created. For example, when you add a primary key integrity constraint, the existing index must have the UNIQUE attribute. When you add a foreign key integrity constraint, the index must not have the UNIQUE attribute.
- The unique integrity constraint has the same effect as the UNIQUE index attribute. Therefore, when one is used, the other is not necessary.

- The NOMISS index attribute and the not-null integrity constraint have different effects. The integrity constraint prevents missing values from being written to the SAS data file and cannot be added to an existing data file that contains missing values. The index attribute allows missing data values in the data file but excludes them from the index.
- When any index is created, it is marked as being “owned” by the user, the integrity constraint, or both. A user cannot delete an index that is also owned by an integrity constraint and vice versa. If an index is owned by both, the index is deleted only after both the integrity constraint and the user have requested the index's deletion. A note in the log indicates when an index cannot be deleted.

## Locking Integrity Constraints

Integrity constraints support both member-level and record-level locking. You can override the default locking level with the CNTLLEV= data set option. For more information, see the “[CNTLLEV= Data Set Option](#)” in *SAS Data Set Options: Reference*.

## Encryption and Integrity Constraints

There are two types of algorithms that SAS uses for encrypting:

- SAS Proprietary encryption is implemented with the ENCRYPT=YES data set option.
- AES (Advanced Encryption Standard) encryption is implemented with the ENCRYPT=AES data set option.

SAS Proprietary encryption has no restrictions when using integrity constraints.

AES encryption requires that 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. You must specify the ENCRYPTKEY= data set option when using ENCRYPT=AES. For more information, see “[ENCRYPT= Data Set Option](#)” in *SAS Data Set Options: Reference* and “[ENCRYPTKEY= Data Set Option](#)” in *SAS Data Set Options: Reference*.

If an encryption key was not recorded for the metadata-bound library, then the encryption key must be the same for the primary key data file and the referencing encrypted foreign key data file. For more information about metadata-bound libraries, see “[Metadata-Bound Library](#)” in *Base SAS Procedures Guide*.

## Specifying Integrity Constraints

You can create integrity constraints in the SQL procedure, the DATASETS procedure, or in SCL (SAS Component Language). The constraints can be specified when the data file is created or can be added to an existing data file. When you add integrity constraints to an existing file, SAS verifies that the existing data values conform to the constraints that are being added.

When you specify integrity constraints, you must specify a separate statement for each constraint. In addition, you must specify a separate statement for each variable to which you want to assign the not-null integrity constraint. When multiple variables are included in the specification for a primary key, foreign key, or a unique integrity constraint, a composite index is created and the integrity constraint enforces the combination of variable values. The relationship between SAS indexes and integrity constraints is described in “[Indexes and Integrity Constraints](#)” on page 715. For more information, see “[Understanding SAS Indexes](#)” on page 692.

When you add an integrity constraint in SCL, open the data set in utility mode. See “[Creating Integrity Constraints By Using SCL](#)” on page 687 for an example.

Integrity constraints must be deleted in utility open mode. For detailed syntax information, see *SAS Component Language: Reference*.

When generation data sets are used, you must create the integrity constraints in each data set generation that includes protected variables.

**CAUTION! CHECK constraints in SAS 9.2 are not compatible with earlier releases of SAS.** If you add a CHECK constraint to an existing SAS data set or create a SAS data set that includes a CHECK constraint, the data set cannot be accessed by a release prior to SAS 9.2.

## Specifying Physical Location for Inter-Libref Referential Integrity Constraints When Sharing Disk Space

When you share disk space over a network and access referential integrity constraints in which the foreign key and primary key data files are in different SAS libraries, a standard should be established for the physical location of the shared files. A standard is required when you create the shared files so that network machines use the same physical name in order to access the files. If the physical names do not match, SAS cannot open the referenced foreign key or primary key SAS data file.

For example, a standard might be established that all shared files are placed on disk T: so that network machines use the same pathname in order to access the files.

Here is an example of a problem regarding files that were created without a standard. Suppose a primary key and a foreign key SAS data file were created on machine D4064 in different directories `C:\Public\pkey_directory` and `C:\Public\fkey_directory`. The pathnames are stored in the descriptor information of the SAS data files.

To access the primary key data file from a different machine such as F2760, the following LIBNAME statement would be executed:

```
libname pkds '\\D4064\Public\pkey_directory';
```

When the primary key data file is opened for update processing, SAS automatically tries to open the foreign key data file by using the foreign key data file's physical name that is stored in the primary key data file, which is `C:\Public\fkey_directory`. However, that directory does not exist on machine F2760. Therefore, opening the foreign key data file fails.

---

## Listing Integrity Constraints

PROC CONTENTS and PROC DATASETS report integrity constraint information without special options. In addition, you can print information about integrity constraints and indexes to a data set by using the OUT2= option. In PROC SQL, the DESCRIBE TABLE and DESCRIBE TABLE CONSTRAINTS statements report integrity constraint characteristics as part of the data file definition or alone, respectively. SCL provides the ICTYPE, ICVALUE, and ICDESCRIBE functions for getting information about integrity constraints. For more information see [Base SAS Procedures Guide](#) and [SAS Component Language: Reference](#) for more information.

---

## Rejected Observations

You can customize the error message that is associated with an integrity constraint when you create the constraint by using the MESSAGE= and MSGTYPE= options. The MESSAGE= option enables you to prepend a user-defined message to the SAS error message associated with an integrity constraint. The MSGTYPE= option enables you to suppress the SAS portion of the message. For more information, see the PROC DATASETS, PROC SQL, and SCL documentation.

Rejected observations can be collected in a special file by using an audit trail.

---

## Integrity Constraints and CEDA Processing

When a SAS data file requires processing with CEDA, integrity constraints are not supported. For example, if you transfer a SAS data file with defined integrity constraints from one operating environment such as Windows to a different operating environment such as UNIX, CEDA translates the file for you, but the integrity constraints are not available. For information about CEDA processing, see [Chapter 34, “Processing Data Using Cross-Environment Data Access \(CEDA\),” on page 765](#).

The MIGRATE procedure preserves integrity constraints when migrating data files. For more information, see the “[MIGRATE Procedure](#)” in [Base SAS Procedures Guide](#). The CPORTR and CIMPORT procedures preserve integrity constraints when transporting SAS data files from one operating environment to another operating environment. The CPORTR procedure makes a copy of the data file in a transportable format. The CIMPORT procedure reads the transport file and creates a new host-specific copy of the data file. For more information, see the “[CPORTR Procedure](#)” in [Base SAS Procedures Guide](#) and “[CIMPORT Procedure](#)” in [Base SAS Procedures Guide](#).

## Examples

### Creating Integrity Constraints with the DATASETS Procedure

The following sample code creates integrity constraints by means of the DATASETS procedure. The data file TV\_Survey checks the percentage of viewing time spent on networks, PBS, and other channels, with the following integrity constraints:

- The viewership percentage cannot exceed 100%.
- Only adults can participate in the survey.
- Gender can be male or female.

```
data tv_survey(label='Validity checking');
length idnum age 4 gender $1;
input idnum gender age network pbs other;
datalines;
1 M 55 80 . 20
2 F 36 50 40 10
3 M 42 20 5 75
4 F 18 30 0 70
5 F 84 0 100 0
;

proc datasets nolist;
modify tv_survey;
  ic create val_gender = check(where=(gender in ('M','F')));
    message = "Valid values for variable GENDER are
either 'M' or 'F'.";
  ic create val_age = check(where=(age >= 18 and age = 120));
    message = "An invalid AGE has been provided.";
  ic create val_new = check(where=(network = 100));
  ic create val_pbs = check(where=(pbs = 100));
  ic create val_ot = check(where=(other = 100));
  ic create val_max = check(where=((network+pbs+other)= 100));
quit;
```

### Creating Integrity Constraints with the SQL Procedure

The following sample program creates integrity constraints by means of the SQL procedure. The data file People lists employees and contains employment information. The data file Salary contains salary and bonus information. The integrity constraints are as follows:

- The names of employees receiving bonuses must be found in the People data file.
- The names identified in the primary key must be unique.
- Gender can be male or female.
- Job status can be permanent, temporary, or terminated.

```

proc sql;
    create table people
    (
        name      char(14),
        gender    char(6),
        hired     num,
        jobtype   char(1) not null,
        status    char(10),
    constraint prim_key primary key(name),
    constraint gender check(gender in ('male' 'female')),
    constraint status check(status in ('permanent'
                                         'temporary' 'terminated'))
    );
    create table salary
    (
        name      char(14),
        salary    num not null,
        bonus     num,
    constraint for_key foreign key(name) references people
        on delete restrict on update set null
    );
quit;

```

## Creating Integrity Constraints By Using SCL

To add integrity constraints to a data file by using SCL, you must create and build an SCL catalog entry. The following sample program creates and compiles catalog entry Example.lc\_Cat\_Allics.SCL.

```

INIT:
put "Test SCL integrity constraint functions start.";
return;

MAIN:
put "Opening WORK.ONE in utility mode.";
dsid = open('work.one', 'V');/* Utility mode.*/
if (dsid = 0) then
do;
    _msg_=sysmsg();
    put _msg_=;
end;
else do;
    if (dsid > 0) then
        put "Successfully opened WORK.ONE in"
            "UTILITY mode.";
end;

put "Create a check integrity constraint named teen.";
rc = iccreate(dsid, 'teen', 'check',
'(age > 12) && (age < 20)');

if (rc > 0) then
do;

```

```

        put rc=;
        _msg_=sysmsg();
        put _msg_=;
end;
else do;
put "Successfully created a check"
    "integrity constraint.";
end;

put "Create a not-null integrity constraint named nn.";
rc = iccreate(dsid, 'nn', 'not-null', 'age');

if (rc > 0) then
do;
put rc=;
(Msg)=sysmsg();
put _msg_=;
end;
else do;
put "Successfully created a not-null"
    "integrity constraint.";
end;

put "Create a unique integrity constraint named uq.";
rc = iccreate(dsid, 'uq', 'unique', 'age');

if (rc > 0) then
do;
put rc=;
(Msg)=sysmsg();
put _msg_=;
end;
else do;
put "Successfully created a unique"
    "integrity constraint.";
end;

put "Create a primary key integrity constraint named pk.";
rc = iccreate(dsid, 'pk', 'Primary', 'name');

if (rc > 0) then
do;
put rc=;
(Msg)=sysmsg();
put _msg_=;
end;
else do;
put "Successfully created a primary key"
    "integrity constraint.";
end;

put "Closing WORK.ONE.";
rc = close(dsid);
if (rc > 0) then
do;
put rc=;

```

```

        _msg_=sysmsg();
        put _msg_=;
        end;

        put "Opening WORK.TWO in utility mode.";
        dsid2 = open('work.two', 'V');
        /*Utility mode */
        if (dsid2 = 0) then
            do;
            _msg_=sysmsg();
            put _msg_=;
            end;
            else do;
                if (dsid2 > 0) then
                    put "Successfully opened WORK.TWO in"
                    "UTILITY mode.";
                end;

        put "Create a foreign key integrity constraint named fk.";
        rc = icccreate(dsid2, 'fk', 'foreign', 'name',
        'work.one','null', 'restrict');

        if (rc > 0) then
            do;
            put rc=;
            _msg_=sysmsg();
            put _msg_=;
            end;
            else do;
                put "Successfully created a foreign key"
                "integrity constraint.";
            end;

        put "Closing WORK.TWO.";
        rc = close(dsid2);
        if (rc > 0) then
            do;
            put rc=;
            _msg_=sysmsg();
            put _msg_=;
            end;
        return;

TERM:
put "End of test SCL integrity constraint"
"functions.";
return;

```

The previous code creates the SCL catalog entry. The following code creates two data files, One and Two, and executes the SCL entry Example.Ic\_Cat\_Allics.SCL:

```

/* Submit to create data files. */

data one two;
    input name $ age;
datalines;
Morris 13

```

```

Elaine 14
Tina 15
;

/* after compiling, run the SCL program */

proc display catalog= example.ic_cat.allics.scl;
run;

```

## Removing Integrity Constraints

The following sample program segments remove integrity constraints. In the code that deletes a primary key integrity constraint, note that the foreign key integrity constraint is deleted first.

This program segment deletes integrity constraints using PROC SQL.

```

proc sql;
    alter table salary
        DROP CONSTRAINT for_key;
    alter table people
        DROP CONSTRAINT gender
        DROP CONSTRAINT _nm0001_
        DROP CONSTRAINT status
        DROP CONSTRAINT prim_key
    ;
quit;

```

This program segment removes integrity constraints using PROC DATASETS.

```

proc datasets nolist;
    modify tv_survey;
        ic delete val_max;
        ic delete val_gender;
        ic delete val_age;
run;
quit;

```

This program segment removes integrity constraints using SCL.

TERM:

```

put "Opening WORK.TWO in utility mode.";
dsid2 = open( 'work.two' , 'V' ); /* Utility mode.      */
if (dsid2 = 0) then
do;
    _msg_=sysmsg();
    put _msg_=;
end;
else do;
    if (dsid2 > 0) then
        put "Successfully opened WORK.TWO in Utility mode.";
end;

rc = icdelete(dsid2, 'fk');
if (rc > 0) then
do;
    put rc=;
    _msg_=sysmsg();
end;

```

```

else
do;
put "Successfully deleted a foreign key integrity constraint.";
end;
rc = close(dsid2);
return;

```

## Reactivating an Inactive Integrity Constraint

The following program segment reactivates a foreign key integrity constraint that has been inactivated as a result of a COPY, CPORt, CIMPORT, UPLOAD, or DOWNLOAD procedure.

```

proc datasets;
  modify SAS-data-set;
    ic reactivate fkname references
libref;
  run;
quit;

```

## Defining Overlapping Primary Key and Foreign Key Constraints

The following code illustrates defining overlapping primary key and foreign key constraints:

```

data Singers1;
  input FirstName $ LastName $ Age;
  datalines;
Tom Jones 62
Kris Kristofferson 66
Willie Nelson 69
Barbra Streisand 60
Paul McCartney 60
Randy Travis 43
;
data Singers2;
  input FirstName $ LastName $ Style $;
  datalines;
Tom Jones Rock
Kris Kristofferson Country
Willie Nelson Country
Barbra Streisand Contemporary
Paul McCartney Rock
Randy Travis Country
;
proc datasets library=work nolist;
  modify Singers1;
    ic create primary key (FirstName LastName); 1
  run;
  modify Singers2;
    ic create foreign key (FirstName LastName) references Singers1
      on delete restrict on update restrict; 2
  run;
  modify Singers2;

```

```

      ic create primary key (LastName FirstName); 3
      run;
      modify Singers1;
      ic create foreign key (LastName FirstName) references Singers2
          on delete restrict on update restrict; 4
      run;

      quit;

```

- 1 Defines a primary key constraint for data set Singers1, for variables FirstName and LastName.
- 2 Defines a foreign key constraint for data set Singers2 for variables FirstName and LastName that references the primary key defined in Step 1. Because the intention is to define a primary key using the same variables, the foreign key update and delete referential actions must both be RESTRICT.
- 3 Defines a primary key constraint for data set Singers2 for variables LastName and FirstName. Because those exact same variables are already defined as a foreign key, the order must be different.
- 4 Defines a foreign key constraint for data set Singers1 for variables LastName and FirstName that references the primary key defined in Step 3. Because those exact same variables are already defined as a primary key, the order must be different. Because a primary key is already defined using the same variables, the foreign key's update and delete referential actions must both be RESTRICT.

## Understanding SAS Indexes

### Definition of SAS Indexes

An index is an optional file that you can create for a SAS data file in order 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 value.

For example, suppose that you want the observation with SSN (Social Security number) equal to 123-45-6789:

- Without an index, SAS accesses observations sequentially in the order in which they are stored in the data file. SAS reads each observation, looking for SSN=123-45-6789 until all observations are read.
- With an index on variable SSN, SAS accesses the observation directly. SAS satisfies the condition using the index and goes straight to the observation that contains the value without having to read each observation.

You can either create an index when you create a data file or create an index for an existing data file. The data file can be either compressed or uncompressed. For each data file, you can create one or multiple indexes. Once an index exists, SAS treats it as part of the data file. That is, if you add or delete observations or modify values, the index is automatically updated.

---

## Benefits of an Index

In general, SAS can use an index to improve performance in the following situations:

- For WHERE processing, an index can provide faster and more efficient access to a subset of data. To process a WHERE expression, SAS by default decides whether to use an index or to read the data file sequentially.
- For BY processing, an index returns observations in the index order, which is in ascending value order, without using the SORT procedure even when the data file is not stored in that order.

**Note:** If you use the SORT procedure, the index is not used.

- 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.

In addition, an index can benefit other areas of SAS. In SCL (SAS Component Language), an index improves the performance of table lookup operations. For the SQL procedure, an index enables the software to process certain classes of queries more efficiently (for example, join queries). For the SAS/IML software, you can explicitly specify that an index be used for read, delete, list, or Append operations.

Even though an index can reduce the time required to locate a set of observations, especially for a large data file, there are costs associated with creating, storing, and maintaining the index. When deciding whether to create an index, you must consider increased resource usage, along with the performance improvement.

**Note:** An index is never used for the subsetting IF statement in a DATA step, or for the FIND and SEARCH commands in the FSEDIT procedure.

---

## The Index File

The index file is a SAS file that has the same name as its associated data file, and that has a member type of INDEX. There is only one index file per data file. That is, all indexes for a data file are stored in a single file.

The index file might be a separate file, or be part of the data file, depending on the operating environment. In any case, the index file is stored in the same SAS library as its data file.

The index file consists of entries that are organized hierarchically and connected by pointers, all of which are maintained by SAS. The lowest level in the index file hierarchy consists of entries that represent each distinct value for an indexed variable, in ascending value order. Each entry contains this information:

- a distinct value
- one or more unique record identifiers (referred to as a RID) that identifies each observation containing the value. (Think of the RID as an internal observation number.)

That is, in an index file, each value is followed by one or more RIDs, which identify the observations in the data file that contains the value. (Multiple RIDs result from

multiple occurrences of the same value.) For example, the following represents index file entries for the variable LastName:

**Table 28.7 Index File Entries**

Value	Record Identifier
Avery	10
Brown	6, 22, 43
Craig	5, 50
Dunn	1

When an index is used to process a request, such as a WHERE expression, SAS performs a binary search on the index file and positions the index to the first entry that contains a qualified value. SAS then uses the value's RID to read the observation that contains the value. If a value has more than one RID (such as in the value for Brown in the previous example), SAS reads the observation that is pointed to by the next RID in the list. The result is that SAS can quickly locate the observations that are associated with a value or range of values.

For example, using an index to process the WHERE expression, SAS positions the index to the index entry for the first value greater than 20 and uses the value's RID or RIDs to read the observation or observations where age > 20 and age < 35;. SAS then moves sequentially through the index entries reading observations until it reaches the index entry for the value that is equal to or greater than 35.

SAS automatically keeps the index file balanced as updates are made, which means that it ensures a uniform cost to access any index entry, and all space that is occupied by deleted values is recovered and reused.

## Types of Indexes

### Simple and Composite Indexes

When you create an index, you designate which variable or variables to index. An indexed variable is called a key variable. You can create two types of indexes:

- a simple index, which consists of the values of one variable
- a composite index, which consists of the values of more than one variable, with the values concatenated to form a single value

In addition to deciding whether you want a simple index or a composite index, you can also limit an index (and its data file) to unique values and exclude from the index missing values.

### Simple Index

The most common index is a simple index, which is an index of values for one key variable. The variable can be numeric or character. When you create a simple index, SAS assigns to the index the name of the key variable.

The following example shows the DATASETS procedure statements that are used to create two simple indexes for variables Class and Major in data file College.Survey:

```
proc datasets library=college;
  modify survey;
    index create class;
    index create major;
run;
```

To process a WHERE expression using an index, SAS uses only one index. When the WHERE expression has multiple conditions using multiple key variables, SAS determines which condition qualifies the smallest subset. For example, suppose that College.Survey contains the following data:

- 42,000 observations contain class=12
- 6,000 observations contain major='Biology'
- 350 observations contain both class=12 and major='Biology'

With simple indexes on Class and Major, SAS would select Major to process the following WHERE expression.

```
where class=12 and major='Biology';
```

## Composite Index

A composite index is an index of two or more key variables with their values concatenated to form a single value. The variables can be numeric, character, or a combination. An example is a composite index for the variables LastName and FirstName. A value for this index consists of the value for LastName immediately followed by the value for FirstName from the same observation. When you create a composite index, you must specify a unique index name.

The following example shows the DATASETS procedure statements that are used to create a composite index for the data file College.MailList, specifying two key variables: ZipCode and Schoolid.

```
proc datasets library=college;
  modify maillist;
    index create zipid=(zipcode schoolid);
run;
```

Often, only the first variable of a composite index is used. For example, for a composite index on ZipCode and Schoolid, the following WHERE expression can use the composite index for the variable ZipCode because it is the first key variable in the composite index:

```
where zipcode = 78753;
```

However, you can take advantage of all key variables in a composite index by how you construct the WHERE expression, which is referred to as compound optimization. Compound optimization is the process of optimizing multiple WHERE expression conditions using a single composite index. If you issue the following WHERE expression, the composite index is used to find all occurrences where the ZIP code is 78753 and the school identification number is 55. In this way, all of the conditions are satisfied with a single search of the index:

```
where zipcode = 78753 and schoolid = 55;
```

When you are deciding whether to create a simple index or a composite index, consider how you will access the data. If you often access data for a single variable, a simple index will do. But if you frequently access data for multiple variables, a composite index could be beneficial.

## Unique Values

Often it is important to require that values for a variable be unique, like Social Security number and employee number. You can declare unique values for a variable by creating an index for the variable and including the UNIQUE option. A unique index guarantees that values for one variable or the combination of a composite group of variables remain unique for every observation in the data file. If an update tries to add a duplicate value to that variable, the update is rejected.

The following example creates a simple index for the variable IdNum and requires that all values for IdNum be unique:

```
proc datasets library=college;
  modify student;
    index create idnum / unique;
run;
```

## Missing Values

If a variable has a large number of missing values, it might be desirable to keep them from using space in the index. Therefore, when you create an index, you can include the NOMISS option to specify that missing values are not maintained by the index.

The following example creates a simple index for the variable Religion and specifies that the index does not maintain missing values for the variable:

```
proc datasets library=college;
  modify student;
    index create religion / nomiss;
run;
```

In contrast to the UNIQUE option, observations with missing values for the key variable can be added to the data file, even though the missing values are not added to the index.

SAS does not use an index that was created with the NOMISS option to process a BY statement or to process a WHERE expression that qualifies observations that contain missing values. If no missing values are present, SAS considers using the index in processing the BY statement or WHERE expression.

In the following example, the index Age was created with the NOMISS option and observations exist that contain missing values for the variable Age. In this case, SAS does not use the index:

```
proc print data=mydata.employee;
  where age < 35;
run;
```

---

## Deciding Whether to Create an Index

### Costs of an Index

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. The following topics provide information about resource usage and give you some guidelines for creating indexes.

#### CPU Cost

Additional CPU time is necessary to create an index as well as to maintain the index when the data file is modified. That is, for an indexed data file, when a value is added, deleted, or modified, it must also be added, deleted, or modified in the appropriate index(es).

When SAS uses an index to read an observation from a data file, there is also increased CPU usage. The increased usage results from SAS using a more complicated process than is used when SAS retrieves data sequentially. Although CPU usage is greater, you benefit from SAS reading only those observations that meet the conditions. Note that increased CPU usage is why using an index is more expensive when there is a larger number of observations that meet the conditions.

**Note:** To compare CPU usage with and without an index, for some operating environments, you can issue the STIMER or FULLSTIMER system options in order to write performance statistics to the SAS log.

#### I/O Cost

Using an index to read observations from a data file can increase the number of I/O (input/output) requests compared to reading the data file sequentially. For example, processing a BY statement with an index might increase I/O count, but you save in not having to issue the SORT procedure. For WHERE processing, SAS considers I/O count when deciding whether to use an index.

- 1 SAS does a binary search on the index file and positions the index to the first entry that contains a qualified value.
- 2 SAS uses the value's RID (identifier) to directly access the observation containing the value. SAS transfers the observation between external storage to a buffer, which is the memory into which data is read or from which data is written. The data is transferred in pages, which is the amount of data (the number of observations) that can be transferred for one I/O request; each data file has a specified page size.
- 3 SAS then continues the process until the WHERE expression is satisfied. Each time SAS accesses an observation, the data file page containing the observation must be read into memory if it is not already there. Therefore, if the observations are on multiple data file pages, an I/O operation is performed for each observation.

The result is that the more random the data, the more I/Os are required to use the index. If the data is ordered more like the index, which is in ascending value order, a smaller number of I/Os are required to access the data.

The number of buffers determines how many pages of data can simultaneously be in memory. Frequently, the larger the number of buffers, the smaller the number of I/Os that are required. For example, if the page size is 4096 bytes and one buffer is allocated, then one I/O transfers 4096 bytes of data (or one page). To reduce I/Os, you can increase the page size but you need a larger buffer. To reduce the buffer size, you can decrease the page size but you use more I/Os.

For information about data file characteristics like the data file page size and the number of data file pages, issue the CONTENTS procedure (or use the CONTENTS statement in the DATASETS procedure). With this information, you can determine the data file page size and experiment with different sizes. Note that the information that is available from PROC CONTENTS depends on the operating environment.

The BUFSIZE= data set option (or system option) sets the permanent page size for a data file when it is created. The page size is the amount of data that can be transferred for an I/O operation to one buffer. The BUFNO= data set option (or system option) specifies how many buffers to allocate for a data file and for the overall system for a given execution of SAS. That is, BUFNO= is not stored as a data set attribute.

## Buffer Requirements

In addition to the resources that are used to create and maintain an index, SAS also requires additional memory for buffers when an index is actually used. Opening the data file opens the index file but none of the indexes. The buffers are not required unless SAS uses the index but they must be allocated in preparation for the index that is being used.

The number of buffers that are allocated depends on the number of levels in the index tree and in the data file open mode. If the data file is open for input, the maximum number of buffers is three; for update, the maximum number is four. (Note that these buffers are available for other uses; they are not dedicated to indexes.)

The IBUFSIZE= system option specifies the page size on disk for an index file when it is created. The default setting causes SAS to use the minimum optimal page size for the operating environment. Typically, you do not need to specify an index page size. However, there are situations that could require a different page size. For more information, see the “[IBUFSIZE= System Option](#)” in *SAS System Options: Reference*.

The IBUFNO= system option specifies an optional number of extra buffers to be allocated when navigating an index file. SAS automatically allocates a minimal number of buffers. Typically, you do not need to specify extra buffers. However, using IBUFNO= to specify extra buffers could improve execution time by limiting the number of input/output operations that are required for a particular index file. The improvement in execution time, however, comes at the expense of increased memory consumption. For more information, see the “[IBUFNO= System Option](#)” in *SAS System Options: Reference*.

## Disk Space Requirements

Additional disk space is required to store the index file. This file might show up as a separate file or appear to be part of the data file, depending on the operating environment.

For information about the index file size, issue the CONTENTS procedure (or the CONTENTS statement in the DATASETS procedure). Note that the available information from PROC CONTENTS depends on the operating environment.

---

## Guidelines for Creating Indexes

### Data File Considerations

- For a small data file, sequential processing is often just as efficient as index processing. Do not create an index if the data file page count is less than three pages. It would be faster to access the data sequentially. To see how many pages are in a data file, use the CONTENTS procedure (or use the CONTENTS statement in the DATASETS procedure). Note that the information that is available from PROC CONTENTS depends on the operating environment.
- Consider the cost of an index for a data file that is frequently changed. If you have a data file that changes often, the overhead associated with updating the index after each change can outweigh the processing advantages you gain from accessing the data with an index.
- Create an index when you intend to retrieve a small subset of observations from a large data file (for example, less than 25% of all observations). When this occurs, the cost of processing data file pages is lower than the overhead of sequentially reading the entire data file. The smaller the subset, the larger the performance gains.
- To reduce the number of I/Os performed when you create an index, first sort the data by the key variable. Then to improve performance, maintain the data file in sorted order by the key variable. This technique reduces the I/Os by grouping like values together. That is, the more ordered the data file is with respect to the key variable, the more efficient the use of the index. If the data file has more than one index, sort the data by the most frequently used key variable.
- An index might not be necessary to optimize a WHERE expression if the data is sorted appropriately in order to satisfy the condition. To process a WHERE expression without an index, SAS first checks for the sort indicator that is stored with the file from a previous SORT procedure. If the sort indicator is appropriate, SAS stops reading the file once there are no more values that satisfy the WHERE expression. For example, consider a file that is sorted by Age, without an index. To process the expression `where age <= 25`, SAS stops reading observations after it finds an observation that is greater than 25. Note that while SAS can determine when to stop reading observations, if there is no index, there is no indication where to begin. Without an index, SAS always begins with the first observation, which can require reading a lot of observations.

### Index Use Considerations

- Keep the number of indexes per data file to a minimum to reduce disk storage and to reduce update costs.
- Consider how often your applications use an index. An index must be used often in order to make up for the resources that are used in creating and maintaining it. That is, do not rely solely on resource savings from processing a WHERE

expression. Take into consideration the resources that it takes to actually create the index and to maintain it every time the data file is changed.

- When you create an index to process a WHERE expression, do not try to create one index that is used to satisfy all queries. If there are several variables that appear in queries, those queries might be best satisfied with simple indexes on the most discriminating of those variables.

## Key Variable Candidates

In most cases, multiple variables are used to query a data file. However, it probably would be a mistake to index all variables in a data file, as certain variables are better candidates than others:

- The variables to be indexed should be variables that are used in queries. That is, your application should require selecting small subsets from a large file, and the most common selection variables should be considered as candidate key variables.
- A variable is a good candidate for indexing when the variable can be used to precisely identify the observations that satisfy a WHERE expression. That is, the variable should be discriminating, which means that the index should select the fewest possible observations. For example, variables such as Age, FirstName, and Gender are not discriminating because it is possible for a large representation of the data to have the same age, first name, and gender. However, a variable such as LastName is a good choice because it is less likely that many employees share the same last name.

For example, consider a data file with variables LastName and Gender.

- If many queries against the data file include LastName, then indexing LastName could prove to be beneficial because the values are usually discriminating. However, the same reasoning would not apply if you issued a large number of queries that included Gender. The Gender variable is not discriminating (because perhaps half the population is male and half is female).
- However, if queries against the data file most often include both LastName and Gender as shown in the following WHERE expression, then creating a composite index on LastName and Gender could improve performance.

```
where lastname='LeVoux' and gender='F';
```

Note that when you create a composite index, the first key variable should be the most discriminating.

## Creating an Index

### Overview of Creating Indexes

You can create one index for a data file, which can be either a simple index or a composite index, and you can create multiple indexes, which can be multiple simple indexes, multiple composite indexes, or a combination of both simple and composite.

- 1 You request to create an index for one or multiple variables using a method such as the INDEX CREATE statement in the DATASETS procedure.
- 2 SAS reads the data file one observation at a time, extracts values and RIDs for each key variable, and places them in the index file.

SAS ensures that the values that are placed in the index are successively the same or increasing. SAS determines whether the data is already sorted by the key variables in ascending order. It determines this by checking the sort indicator in the data file, which is an attribute of the file that indicates how the data is sorted. The sort indicator is stored with the SAS data file descriptor information and is set from a previous SORT procedure or SORTEDBY= data set option.

If the values in the sort indicator are in ascending order, SAS does not sort the values for the index file and avoids the resource. Note that SAS always validates that the data is sorted as indicated. If not, the index is not created. For example, if the sort indicator was set from a SORTEDBY= data set option and the data is not sorted as indicated, an error occurs. A message is written to the SAS log stating that the index was not created because values are not sorted in ascending order.

If the values in the sort indicator are not in ascending order, SAS sorts the data that is included in the index file in ascending value order. To sort the data, SAS follows this procedure:

- 1 SAS first attempts to sort the data using the thread-enabled sort. By dividing the sorting into separately executable processes, the time to sort the data can be reduced. To use the thread-enabled sort, the index must be sufficiently large (which is determined by SAS), the SAS system option CPUCOUNT= must be set to more than one processor, and the THREADS system option must be enabled. Adequate memory must be available for the thread-enabled sort. If not enough memory is available, SAS reduces the number of threads to one and begins the sort process again, which increases the time to create the index.
- 2 If the thread-enabled sort cannot be done, SAS uses the unthreaded sort.

**Note:** To display messages regarding what type of sort is used, memory and resource information, and the status of the index being created, set the SAS system option MSGLEVEL=I; that is:

```
options msglevel=i;
```

## Using the DATASETS Procedure

The DATASETS procedure provides statements that enable you to create and delete indexes. In the following example, the MODIFY statement identifies the data file, the INDEX DELETE statement deletes two indexes, and the two INDEX CREATE statements specify the variables to index, with the first INDEX CREATE statement specifying the options UNIQUE and NOMISS:

```
proc datasets library=mylib;
modify employee;
  index delete salary age;
  index create empnum / unique nomiss;
  index create names=(lastname firstname);
```

**Note:** If you delete and create indexes in the same step, place the INDEX DELETE statement before the INDEX CREATE statement so that space occupied by deleted indexes can be reused during index creation.

## Using the INDEX= Data Set Option

To create indexes in a DATA step when you create the data file, use the INDEX= data set option. The INDEX= data set option also enables you to include the NOMISS and UNIQUE options. The following example creates a simple index on the variable Stock and specifies UNIQUE:

```
data finances(index=(stock /unique));
```

The next example uses the variables SSN, City, and State to create a simple index named SSN and a composite index named CitySt:

```
data employee(index=(ssn cityst=(city state)));
```

## Using the SQL Procedure

The SQL procedure supports index creation and deletion and the UNIQUE option. Note that the variable list requires that variable names be separated by commas (which is an SQL convention) instead of blanks (which is a SAS convention).

The DROP INDEX statement deletes indexes. The CREATE INDEX statement specifies the UNIQUE option, the name of the index, the target data file, and the variable or variables to be indexed. For example:

```
drop index salary from employee;
create unique index empnum on employee (empnum);
create index names on employee (lastname, firstname);
```

## Using Other SAS Products

You can also create and delete indexes using other SAS utilities and products, such as SAS/CONNECT software, SAS/IML software, SAS Component Language, and SAS/Warehouse Administrator.

## Using an Index for WHERE Processing

### Overview of Using an Index for WHERE Processing

WHERE processing conditionally selects observations for processing when you issue a WHERE expression. Using an index to process a WHERE expression improves performance and is referred to as optimizing the WHERE expression.

To process a WHERE expression, by default SAS decides whether to use an index or read all the observations in the data file sequentially. To make this decision, SAS does the following:

- 1 Identifies an available index or indexes.
- 2 Estimates the number of observations that would be qualified. If multiple indexes are available, SAS selects the index that returns the smallest subset of observations.

- 3 Compares resource usage to decide whether it is more efficient to satisfy the WHERE expression by using the index or by reading all the observations sequentially.

**Note:** SAS considers several factors when deciding whether to use an index. Therefore, experimentation is the best way to determine the optimal performance. If you have a WHERE expression that is used repeatedly, compare the results using an index and without an index in order to determine which method provides the best performance. You can control index usage with the IDXWHERE= and IDXNAME= data set options. See “[Controlling WHERE Processing Index Usage with Data Set Options](#)” on page 708.

## Identifying an Available Index or Indexes

The first step for SAS in deciding whether to use an index to process a WHERE expression is to identify if the variable or variables included in the WHERE expression are key variables (that is, have an index). Even though a WHERE expression can consist of multiple conditions that specify different variables, SAS uses only one index to process the WHERE expression. SAS selects the index that satisfies the most conditions and qualifies the fewest observations:

- Usually, SAS selects one condition. The variable specified in the condition has either a simple index or is the first key variable in a composite index.
- However, you can take advantage of multiple key variables in a composite index by constructing an appropriate WHERE expression, referred to as compound optimization. See “[Compound Optimization](#)” on page 705.

SAS attempts to use an index for the following types of conditions:

**Table 28.8** WHERE Conditions That Can Be Optimized

Condition	Valid for Compound Optimization	Examples
comparison operators, which include the EQ operator; directional comparisons like less than or greater than; and the IN operator	yes	where emppnum eq 3374; where emppnum < 2000; where state in ('NC','TX');
comparison operators with NOT	yes	where emppnum ^= 3374; where x not in (5,10);
comparison operators with the colon modifier	yes	where lastname gt: 'Sm';
CONTAINS operator	no	where lastname contains 'Sm';
fully bounded range conditions specifying both an upper and lower limit, which includes the BETWEEN-AND operator	yes	where 1 < x < 10; where emppnum between 500 and 1000;

Condition	Valid for Compound Optimization	Examples
pattern-matching operators LIKE and NOT LIKE	no	where firstname like '%Rob_%'
IS NULL or IS MISSING operator	no	where name is null; where idnum is missing;
TRIM function	no	where trim(state)='Texas';
SUBSTR (left of =) function in the form of:  WHERE SUBSTR ( <i>variable</i> , <i>position</i> <, <i>length</i> >)='string';  when the following conditions are met:  <i>position</i> specifies a numeric constant for the beginning character position that is less than or equal to the <i>variable</i> length.  <i>length</i> specifies a numeric constant for the length of <i>string</i> . The <i>length</i> plus <i>position</i> cannot be larger than the <i>variable</i> length plus 1.	no	where substr (month,4,5)='ember' and (city='Charleston' or city='Atlanta');

**Note:** Conditions are not optimized with an index for arithmetic operators, a variable-to-variable condition, the sounds-like operator, and any function other than the TRIM and SUBSTR function as listed above.

The following examples illustrate optimizing a single condition:

- The following WHERE expressions could use a simple index on the variable Major:
 

```
where major in ('Biology', 'Chemistry', 'Agriculture');
where class=11 and major in ('Biology', 'Agriculture');
```
- With a composite index on variables ZipCode and Schoold, SAS could use the composite index to satisfy the following conditions because ZipCode is the first key variable in the composite index:
 

```
where zipcode = 78753;
```

However, the following condition cannot use the composite index because the variable Schoold is not the first key variable in the composite index:

```
where schoolid gt 1000;
```

## Compound Optimization

Compound optimization is the process of optimizing multiple WHERE expression conditions using a single composite index. Using a single index to optimize the conditions can greatly improve performance.

For example, suppose there is a composite index for LastName and FirstName. If you execute the following WHERE expression, SAS uses the concatenated values for the first two variables, then SAS further evaluates each qualified observation for the EmpId value:

```
where lastname eq 'Smith' and firstname eq 'John' and empid=3374;
```

For compound optimization to occur, all of the following must be true.

- At least the first two key variables in the composite index must be used in valid WHERE expression conditions. For a list of conditions that are valid for compound optimization, see [Table 28.8 on page 703](#).
- At least one condition must use the EQ or IN operator. For example, you cannot have all range conditions.
- The conditions must be connected with the AND or the OR logical operator:
  - When conditions are connected with AND, the conditions can occur in any order. For example:
 

```
where lastname eq 'Smith' and firstname eq 'John';
```
  - When conditions are connected with OR, the conditions must specify the same variable. For example:
 

```
where firstname eq 'John' and
                (lastname eq 'Smith' or lastname eq 'Jones');
```

**Note:** SAS transforms the OR conditions that specify the same variable into a single condition that uses the IN operator. For the above WHERE expression, SAS converts the two OR conditions into `lastname IN ('Smith', 'Jones')`, and then uses the composite index for the variables FirstName and LastName in order to select the observations where FirstName is John and LastName is Smith or Jones.

For the following examples, assume there is a composite index for variables I, J, and CH:

- The following WHERE expression conditions are compound optimized because every condition specifies a variable that is in the composite index, and each condition uses one of the supported operators. SAS positions the composite index to the first entry that meets all three conditions and retrieves only observations that satisfy all three conditions.
 

```
where I = 1 and J not in (3,4) and 'abc' < CH;
```
- For the following WHERE expression, the first two conditions are compound optimized. After retrieving a subset of observations that satisfy the first two conditions, SAS examines the subset and eliminates any observations that fail to match the third condition.
 

```
where I in (1,4) and J = 5 and K like '%c';
```
- This WHERE expression can be compound optimized for variables I and J. After retrieving observations that satisfy the second and third conditions, SAS

examines the subset and eliminates those observations that do not satisfy the first condition.

where X < 5 and I = 1 and J = 2;

- The following WHERE expression can be compound optimized on I and J:

where X < Z and I = 1 and J = 2;

- The following WHERE expression cannot be compound optimized neither J or K is the left-most variable in the composite index:

where J = 1 and K = 2;

- The following WHERE expression cannot be optimized because the comparison condition on the variable I is variable-to-variable, which is not supported for index processing:

where I < K and J in (3,4) and CH = 'abc';

Compound optimization can occur for a NOMISS composite index as long as at least one condition does not qualify missing values. That is, compound optimization cannot occur on a NOMISS index, which is an index that does not maintain missing values, if every condition could result in a missing value. The following examples illustrate compound optimization with a NOMISS composite index for variables I, J, and K.

- The following WHERE expression can be compound optimized, because the condition K = 1 cannot result in a missing value:

where I in (.,5) and J < 4 and K = 1;

- This WHERE expression cannot be compound optimized, because each condition could result in a missing value:

where I in (.,5) and J < 4 and K <= 1;

- The following WHERE expression cannot be compound optimized, because each condition could result in a missing value. The condition J < 4 qualifies observations as J = ., and those observations are not represented in the NOMISS composite index:

where I = . and J < 4 and .A < K < .D;

## Estimating the Number of Qualified Observations

Once SAS identifies the index or indexes that can satisfy the WHERE expression, the software estimates the number of observations that will be qualified by an available index. When multiple indexes exist, SAS selects the one that seems to produce the fewest qualified observations.

SAS estimates the number of observations that will be qualified by using stored statistics called cumulative percentiles (or centiles for short). Centiles information represents the distribution of values in an index so that SAS does not have to assume a uniform distribution. To print centiles information for an indexed data file, include the CENTILES option in PROC CONTENTS (or in the CONTENTS statement in the DATASETS procedure).

Note that, by default, SAS does not update centiles information after every data file change. When you create an index, you can include the UPDATECENTILES option to specify when centiles information is updated. That is, you can specify that centiles information be updated every time the data file is closed, when a certain percentage of values for the key variable have been changed, or never. In addition, you can

also request that centiles information is updated immediately, regardless of the value of UPDATECENTILES, by issuing the INDEX CENTILES statement in PROC DATASETS.

As a general rule, SAS uses an index if it estimates that the WHERE expression will select approximately one-third or less of the total number of observations in the data file.

**Note:** For performance purposes, the following can occur when SAS estimates the number of qualified observations:

- If the number of qualified observations is less than 3% of the data file (or if no observations are qualified), SAS automatically uses the index, and does not bother comparing resource usage.
- If all of the observations are qualified, by default SAS does not use the index unless the IDXNAME= or IDXWHERE= data set option is specified.

## Comparing Resource Usage

Once SAS estimates the number of qualified observations and selects the index that qualifies the fewest observations, SAS must then decide whether it is faster (cheaper) to satisfy the WHERE expression by using the index or by reading all of the observations sequentially. SAS makes this determination as follows:

- If only a few observations are qualified, it is more efficient to use the index than to do a sequential search of the entire data file.
- If most or all of the observations qualify, then it is more efficient to simply sequentially search the data file than to use the index.

This decision is much like a reader deciding whether to use an index at the back of a document. A document's index is designed to enable a reader to locate a topic along with the specific page number. Using the index, the reader would go to a specific page number and read only about a specific topic. If the document covers 42 topics and the reader is interested in only a couple of topics, then the index saves time by preventing the reader from reading other topics. However, if the reader is interested in 39 topics, searching the index for each topic would take more time than simply reading the entire document.

To compare resource usage, SAS does the following:

- 1 SAS predicts the number of I/Os that it takes to satisfy the WHERE expression using the index. To do so, SAS positions the index to the first entry that contains a qualified value. In a buffer management simulation that takes into account the current number of available buffers, the RIDs (identifiers) on that index page are processed, indicating how many I/Os it takes to read the observations in the data file.

If the observations are randomly distributed throughout the data file, the observations are located on multiple data file pages. This means that an I/O is needed for each page. Therefore, the more random the data in the data file, the more I/Os it takes to use the index. If the data in the data file is ordered more like the index, which is in ascending value order, a smaller number of I/Os are needed to use the index.

- 2 SAS calculates the I/O cost of a sequential pass of the entire data file and compares the two resource costs.

Factors that affect the comparison include the size of the subset relative to the size of the data file, data file value order, data file page size, the number of allocated buffers, and the cost to uncompress a compressed data file for a sequential read.

**Note:** If comparing resource costs results in a tie, SAS chooses the index.

## Controlling WHERE Processing Index Usage with Data Set Options

You can control index usage for WHERE processing with the IDXWHERE= and IDXNAME= data set options.

The IDXWHERE= data set option overrides the software's decision regarding whether to use an index to satisfy the conditions of a WHERE expression as follows:

- IDXWHERE=YES tells SAS to decide which index is the best for optimizing a WHERE expression, disregarding the possibility that a sequential search of the data file might be more resource efficient.
- IDXWHERE=NO tells SAS to ignore all indexes and satisfy the conditions of a WHERE expression by sequentially searching the data file.
- Using an index to process a BY statement cannot be overridden with IDXWHERE=.

The following example tells SAS to decide which index is the best for optimizing the WHERE expression. SAS disregards the possibility that a sequential search of the data file might be more resource efficient.

```
data mydata.empnew;
  set mydata.employee (idxwhere=yes);
  where empnum < 2000;
```

For details, see the IDXWHERE data set option in [SAS Data Set Options: Reference](#).

The IDXNAME= data set option directs SAS to use a specific index in order to satisfy the conditions of a WHERE expression.

By specifying IDXNAME=*index-name*, you are specifying the name of a simple or composite index for the data file.

The following example uses the IDXNAME= data set option to direct SAS to use a specific index to optimize the WHERE expression. SAS disregards the possibility that a sequential search of the data file might be more resource efficient. SAS does not attempt to determine whether the specified index is the best one. (Note that the EMPNUM index was not created with the NOMISS option.)

```
data mydata.empnew;
  set mydata.employee (idxname=empnum);
  where empnum < 2000;
```

For details, see the IDXNAME data set option in [SAS Data Set Options: Reference](#).

**Note:** IDXWHERE= and IDXNAME= are mutually exclusive. Using both options results in an error.

## Displaying Index Usage Information in the SAS Log

To display information in the SAS log regarding index usage, change the value of the MSGLEVEL= system option from its default value of N to I. When you issue options msglevel=i;, the following occurs:

- If an index is used, a message displays the name of the index.
- If an index is not used but one exists that could optimize at least one condition in the WHERE expression, messages provide suggestions as to what you can do to influence SAS to use the index. For example, a message could suggest sorting the data file into index order or specifying more buffers.
- A message displays the IDXWHERE= or IDXNAME= data set option value if the setting can affect index processing.

## Using an Index with SAS Views

A SAS view is a type of SAS data set that retrieves data values from other files. There are two types of SAS views: a DATA step view and a PROC view. For more information about SAS views, see “[SAS Views](#)” on page 721.

You cannot create an index for a SAS view; it must be a data file. However, if a SAS view is created from an indexed data file, index usage is available. That is, if the view definition includes a WHERE expression using a key variable, then SAS attempts to use the index. There are other ways to take advantage of a key variable when using a SAS view.

In this example, you create an SQL view named Stat from data file Crime, which has the key variable State. In addition, the view definition includes a WHERE expression:

```
proc sql;
  create view stat as
    select * from crime
    where murder > 7;
quit;
```

If you issue PROC SQL with an SQL WHERE clause that specifies the key variable State, then the SQL view can join the two conditions, which enables SAS to use the index State:

```
proc sql;
select * from stat where state > 42;
quit;
```

---

## Using an Index for BY Processing

BY processing enables you to process observations in a specific order according to the values of one or more variables that are specified in a BY statement. Indexing a data file enables you to use a BY statement without sorting the data file. By creating an index based on one or more variables, you ensure that observations are processed in ascending numeric or character order. Specify in the BY statement the variable or list of variables that are indexed.

For example, if an index exists for LastName, the following BY statement would use the index to order the values by last names:

```
proc print;
  by lastname;
```

When you specify a BY statement, SAS looks for an appropriate index. If one exists, the software automatically retrieves the observations from the data file in indexed order.

A BY statement uses an index in the following situations:

- The BY statement consists of one variable that is the key variable for a simple index or the first key variable in a composite index.
- The BY statement consists of two or more variables and the first variable is the key variable for a simple index or the first key variable in a composite index.

For example, if the variable Major has a simple index, the following BY statements use the index to order the values by Major:

```
by major;
by major state;
```

If a composite index named ZipId exists consisting of the variables ZipCode and Schoolid, the following BY statements use the index:

```
by zipcode;
by zipcode schoolid;
by zipcode schoolid name;
```

However, the composite index ZipId is not used for these BY statements:

```
by schoolid;
by schoolid zipcode;
```

In addition, a BY statement does not use an index in these situations:

- The BY statement includes the DESCENDING or NOTSORTED option.
- The index was created with the NOMISS option.
- The data file is physically stored in sorted order based on the variables specified in the BY statement.

**Note:** Using an index to process a BY statement might not always be more efficient than simply sorting the data file, particularly if the data file has a high blocking factor of observations per page. Therefore, using an index for a BY statement is generally for convenience, not performance.

## Using an Index for Both WHERE and BY Processing

If both a WHERE expression and a BY statement are specified, SAS looks for one index that satisfies requirements for both. If such an index is not found, the BY statement takes precedence.

With a BY statement, SAS cannot use an index to optimize a WHERE expression if the optimization would invalidate the BY order. For example, the following statements could use an index on the variable LastName to optimize the WHERE expression because the order of the observations returned by the index does not conflict with the order required by the BY statement:

```
proc print;
  by lastname;
  where lastname >= 'Smith';
run;
```

However, the following statements cannot use an index on LastName to optimize the WHERE expression because the BY statement requires that the observations be returned in EmpId order:

```
proc print;
  by empid;
  where lastname = 'Smith';
run;
```

## Specifying an Index with the KEY= Option for SET and MODIFY Statements

The SET and MODIFY statements provide the KEY= option, which enables you to specify an index in a DATA step to retrieve particular observations in a data file.

The following MODIFY statement shows how to use the KEY= option to take advantage of the fact that the data file Invty.Stock has an index on the variable Partno. Using the KEY= option tells SAS to use the index to directly access the correct observations to modify.

```
modify invty.stock key=partno;
```

**Note:** A BY statement is not allowed in the same DATA step with the KEY= option, and WHERE processing is not allowed for a data file with the KEY= option.

## Taking Advantage of an Index

Applications that typically do not use indexes can be rewritten to take advantage of an index. For example:

- Consider replacing a subsetting IF statement (which never uses an index) with a WHERE statement.
 

**CAUTION!** However, be careful because IF and WHERE statements are processed differently and might produce different results in DATA steps that use the SET, MERGE, or UPDATE statements. This is because the WHERE statement selects observations before they are brought into the Program Data Vector (PDV), whereas the subsetting IF statement selects observations after they are read into the PDV.
- Consider using the WHERE command in the FSEDIT procedure in place of the SEARCH and FIND commands.

---

## Procedures and SAS Operations That Maintain Indexes

### Displaying Data File Information

The CONTENTS procedure (or the CONTENTS statement in PROC DATASETS) reports the following types of information.

- number and names of indexes for a data file
- the names of key variables
- the options in effect for each key variable
- data file page size
- number of data file pages
- centiles information (using the CENTILES option)
- amount of disk space used by the index file

**Note:** The available information depends on the operating environment.

**Output 28.6 Output of PROC CONTENTS**

The SAS System					
The CONTENTS Procedure					
Data Set Name	MYFILES.STAFF	Observations	148		
Member Type	DATA	Variables	6		
Engine	V9	Indexes	2		
Created	08/17/2012 11:10:03	Observation Length	63		
Last Modified	08/17/2012 11:16:51	Deleted Observations	0		
Protection		Compressed	NO		
Data Set Type		Sorted	NO		
Label					
Data Representation	WINDOWS_32				
Encoding	wlatin1 Western (Windows)				
Engine/Host Dependent Information					
Data Set Page Size	8192				
Number of Data Set Pages	3				
First Data Page	1				
Max Obs per Page	129				
Obs in First Data Page	104				
Index File Page Size	4096				
Number of Index File Pages	5				
Number of Data Set Repairs	0				
Filename	C:\My Documents\staff.sas7bdat				
Release Created	9.0401B0				
Host Created	W32_7PRO				
Alphabetic List of Variables and Attributes					
#	Variable	Type	Len		
4	city	Char	15		
3	fname	Char	15		
6	hphone	Char	12		
1	idnum	Char	4		
2	lname	Char	15		
5	state	Char	2		
Alphabetic List of Indexes and Attributes					
#	Index	Unique Option	NoMiss Option	# of Unique Values	Variables
1	idnum	YES	YES	148	
2	name			148	fname lname

## Copying an Indexed Data File

When you copy an indexed data file with the COPY procedure (or the COPY statement of the DATASETS procedure), you can specify whether the procedure also re-creates the index file for the new data file with the INDEX=YES|NO option; the default is YES, which re-creates the index. However, re-creating the index does increase the processing time for the PROC COPY step.

If you copy from disk to disk, the index is re-created. If you copy from disk to tape, the index is not re-created on tape. However, after copying from disk to tape, if you then copy back from tape to disk, the index can be re-created. Note that if you move a data file with the MOVE option in PROC COPY, the index file is deleted from IN= library and re-created in OUT= library.

The CPRT procedure also has INDEX=YES|NO to specify whether to export indexes with indexed data files. By default, PROC CPRT exports indexes with indexed data files. The CIMPORT procedure, however, does not handle the index file at all, and the index(es) must be re-created.

## Updating an Indexed Data File

Each time that values in an indexed data file are added, modified, or deleted, SAS automatically updates the index. The following activities affect an index as indicated:

*Table 28.9 Maintenance Tasks and Index Results*

Task	Result
delete a data set	index file is deleted
rename a data set	index file is renamed
rename key variable	simple index is renamed
delete key variable	simple index is deleted
add observation	index entries are added
delete observations	index entries are deleted and space is recovered for reuse
update observations	index entries are deleted and new ones are inserted

**Note:** Use SAS to perform additions, modifications, and deletions to your data sets. Using operating environment commands to perform these operations makes your files unusable.

## Sorting an Indexed Data File

You can sort an indexed data file only if you direct the output of the SORT procedure to a new data file so that the original data file remains unchanged. However, the new data file is not automatically indexed.

**Note:** If you sort an indexed data file with the FORCE option, the index file is deleted.

## Adding Observations to an Indexed Data File

Adding observations to an indexed data file requires additional processing. SAS automatically keeps the values in the index consistent with the values in the data file.

## Multiple Occurrences of Values

An index that is created without the UNIQUE option can result in multiple occurrences of the same value, which results in multiple RIDs for one value. For large data files with many multiple occurrences, the list of RIDs for a given value might require several pages in the index file. Because the RIDs are stored in physical order, any new observation added to the data file with the given value is stored at the end of the list of RIDs. Navigating through the index to find the end of the RID list can cause many I/O operations.

SAS remembers the previous position in the index so that when inserting more occurrences of the same value, the end of the RID list is found quickly.

## Appending Data to an Indexed Data File

SAS provides performance improvements when appending a data file to an indexed data file. SAS suspends index updates until all observations are added, and then updates the index with data from the newly added observations. See the APPEND statement in the DATASETS procedure in [Base SAS Procedures Guide](#).

## Recovering a Damaged Index

An index can become damaged for many of the same reasons that a data file or catalog can become damaged. If a data file becomes damaged, use the REPAIR statement in PROC DATASETS to repair the data file or re-create any missing indexes. For example:

```
proc datasets library=mylib;
  repair mydata;
run;
```

## Indexes and Integrity Constraints

Integrity constraints can also use indexes. When an integrity constraint that uses an index is created, if a suitable index already exists, it is used. Otherwise, a new index is created. When an index is created, it is marked as being “owned” by the creator, which can be either the user or an integrity constraint.

If either the user or an integrity constraint requests creation of an index that already exists and is owned by the other, the requestor is also marked as an “owner” of the index. If an index is owned by both, then a request by either to delete the index results in removing only the requestor as owner. The index is deleted only after both the integrity constraint and the user have requested the index's deletion. A note in the log indicates when an index cannot be deleted.

## Indexes and CEDA Processing

When processing a SAS data file with CEDA, indexes are not supported. For example, if you move a SAS data file with a defined index from one operating environment like Windows to a different operating environment like UNIX, CEDA translates the file for you, but the index is not available. Therefore, WHERE optimization for the file is not supported.

For information about CEDA, see [Chapter 34, “Processing Data Using Cross-Environment Data Access \(CEDA\),” on page 765](#).

---

## Extended Attributes

---

### Definition

You can think of extended attributes as customized metadata for your SAS files. Whereas common SAS attributes such as Labels for data sets, or Length and Label for variables are predefined SAS system attributes, extended attributes are attributes that you define yourself. They are organized into name-value pairs and are associated with either a variable within a SAS data set or a SAS data set in general. Extended attributes are organized into (name, value) pairs and for the BASE engine, their data is stored in a separate SAS data file with file extension `sas7bxat`.

---

## Enabling and Manipulating Extended Attributes

---

You can create, add, delete, update, remove, and specify options for extended attributes using various XATTR statements in the DATASETS procedure. You can also use PROC CONTENTS to display data set and variable extended attributes. Only the V9 engine supports extended attributes. Under engines that do not support extended attributes, if you copy a data set with the DATA step, the COPY procedure, or the COPY statement of the DATASETS procedure, extended attributes are dropped and a warning is written to the log. For more information about extended attributes, see [“Extended Attributes” in Base SAS Procedures Guide](#).

The following output shows the results of running PROC CONTENTS on a SAS data set with extended attributes.

**Output 28.7** PROC CONTENTS Output Showing List of Extended Attributes

Alphabetic List of Data Set Extended Attributes			
Extended Attribute	Numeric Value	Character Value	
attrib	.	table	
role	.	train	

Alphabetic List of Extended Attributes on Variables			
Extended Attribute	Attribute Variable	Numeric Value	Character Value
level	income	.	interval
level	purchase	.	nominal
role	age	.	reject
role	income	.	input
role	purchase	.	target

The CONTENTS procedure and the DATASETS procedure produce the following error when the .sas7bxat (extended attributes) file is absent from the directory and extended attributes were defined:

ERROR: File libref.dsname\_EXT.EXTATTR does not exist.

---

## Compressing Data Files

---

### Definition of Compression

Compressing a file is a process that reduces the number of bytes required to represent each observation. In a compressed file, each observation is a variable-length record, while in an uncompressed file, each observation is a fixed-length record.

Advantages of compressing a file include the following:

- reduced storage requirements for the file
- less I/O operations necessary to read from or write to the data during processing

There are disadvantages to compressing a file. For example:

- More CPU resources are required to read a compressed file because of the overhead of uncompressing each observation.
- There are situations when the resulting file size can increase rather than decrease.

## Requesting Compression

By default, a SAS data file is not compressed. To compress, you can use these options:

- COMPRESS= system option to compress all data files that are created during a SAS session
  - COMPRESS= option in the LIBNAME statement to compress all data files for a particular SAS library
  - COMPRESS= data set option to compress an individual data file
- To compress a data file, you can specify one of the following:
- COMPRESS=CHAR to use the RLE (Run Length Encoding) compression algorithm
  - COMPRESS=BINARY to use the RDC (Ross Data Compression) algorithm

When you create a compressed data file, SAS writes a note to the log indicating the percentage of reduction that is obtained by compressing the file. SAS obtains the compression percentage by comparing the size of the compressed file with the size of an uncompressed file of the same page size and record count.

After a file is compressed, the setting is a permanent attribute of the file. This means that you must re-create the file to change the setting. That is, to uncompress a file, specify COMPRESS=NO for a DATA step that copies the compressed data file.

For more information about the COMPRESS= data set option, see [SAS Data Set Options: Reference](#). For more information about the COMPRESS= option in the LIBNAME statement, see [SAS Global Statements: Reference](#). For more information about the COMPRESS= system option, see [SAS System Options: Reference](#).

## Disabling a Compression Request

The V9 engine compresses one observation at a time, and adds a fixed-length block of data to each observation. Because of the additional block of data, some data sets would result in a larger file size if compressed. For example, a data set with an extremely short observation length would not benefit from compression.

When a request is made to compress a data set, SAS attempts to determine whether compression would increase the size of the file. SAS examines the lengths of the variables. If the additional data that is needed for compression would increase the file size, then compression is disabled and a message is written to the SAS log. The size of the additional data depends on the operating environment. For example, on Windows the additional data is 12 bytes per observation for 32-bit and 64-bit SAS. On 64-bit UNIX, the additional data is 24 bytes.

For example, here is a simple data set for which SAS determines that it is not possible for the compressed file to be smaller than an uncompressed one:

```
data one (compress=char) ;  
    length x y $2;  
    input x y;  
    datalines;  
ab cd  
;
```

The following output is written to the SAS log:

**Example Code 28.1 SAS Log Output When Compression Request Is Disabled**

```
NOTE: Compression was disabled for data set WORK.ONE because compression  
      overhead would increase the size of the data set.  
NOTE: The data set WORK.ONE has 1 observations and 2 variables.
```



# SAS Views

---

<b>SAS Views .....</b>	<b>721</b>
Definition of SAS Views .....	721
<b>Benefits of Using SAS Views .....</b>	<b>722</b>
<b>When to Use SAS Views .....</b>	<b>723</b>
<b>DATA Step Views .....</b>	<b>724</b>
Definition of a DATA Step View .....	724
Creating DATA Step Views .....	724
What Can You Do with a DATA Step View? .....	725
Differences between DATA Step Views and Stored Compiled DATA Step Programs .....	725
Restrictions and Requirements .....	725
Performance Considerations .....	726
Example 1: Merging Data to Produce Reports .....	726
Example 2: Producing Additional Output Files .....	727
<b>PROC SQL Views .....</b>	<b>729</b>
<b>Comparing DATA Step and PROC SQL Views .....</b>	<b>729</b>
<b>SAS/ACCESS Views .....</b>	<b>730</b>

---

## SAS Views

---

### Definition of SAS Views

A SAS view is a type of SAS data set that retrieves data values from other files. A SAS view contains only descriptor information such as the data types and lengths of the variables (columns). A SAS view also contains information that is required for retrieving data values from other SAS data sets or from files that are stored in other software vendors' file formats. SAS views are of member type VIEW. In most cases, you can use a SAS view as if it were a SAS data file.

There are two general types of SAS views:

native view

is a SAS view that is created either with a DATA step or with PROC SQL.

#### interface view

is a SAS view that is created with SAS/ACCESS software. An interface view can read data from or write data to a database management system (DBMS) such as DB2 or ORACLE. Interface views are also referred to as SAS/ACCESS views. In order to use SAS/ACCESS views, you must have a license for SAS/ACCESS software.

**Note:** You can create native views that access certain DBMS data by using a SAS/ACCESS dynamic LIBNAME engine. See “[SAS/ACCESS Views](#)” on page [730](#), or the SAS/ACCESS documentation for your DBMS for more information.

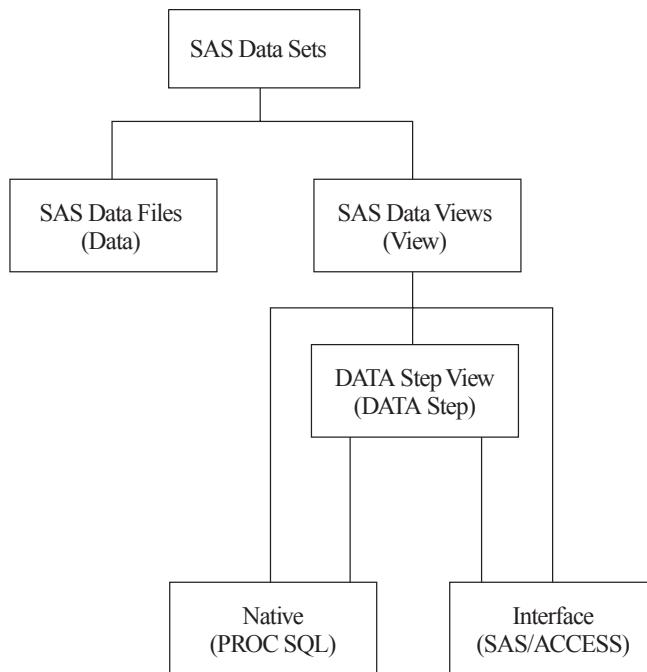
---

## Benefits of Using SAS Views

SAS views provide the following benefits:

- Instead of using multiple DATA steps to merge SAS data sets by common variables, you can construct a SAS view that performs a multi-table join.
- You can save disk space by storing a SAS view definition. The view definition stores only the instructions for where to find the data and how it is formatted, not the actual data.
- SAS views can ensure that the input data sets are always current because data is derived from SAS views at execution time.
- Since SAS views can select data from many sources, once a SAS view is created, it can provide prepackaged information to the information community without the need for additional programming.
- SAS views can reduce the impact of data design changes on users. For example, you can change a query that is stored in a SAS view without changing the characteristics of the view's result.
- With SAS/CONNECT software, a SAS view can join SAS data sets that reside on different host computers, presenting you with an integrated view of distributed company data.

The following figure shows native and interface SAS views and their relationship to SAS data files:

**Figure 29.1** Native and Interface SAS Views

You can use SAS views in the following ways:

- as input to other DATA steps or PROC steps
- to migrate data to SAS data files or to database management systems that are supported by SAS
- in combination with other data sources using PROC SQL
- as pre-assembled sets of data for users of SAS/ASSIST software, enabling them to perform data management, analysis, and reporting tasks regardless of how the data is stored

## When to Use SAS Views

Consider the following in order to determine whether a SAS data file or a SAS view is better for your purposes:

- Data files use additional disk space; SAS views use additional processing time.
- Data file variables can be sorted and indexed before using; SAS views must process data in its existing form during execution.

---

# DATA Step Views

---

## Definition of a DATA Step View

A DATA step view is a native view that has the broadest scope of any SAS view. It contains a stored DATA step program that can read data from a variety of sources, including:

- raw data files
- SAS data files
- PROC SQL views
- SAS/ACCESS views
- DB2, ORACLE, or other DBMS data

---

## Creating DATA Step Views

In order to create a DATA step view, specify the VIEW= option after the final data set name in the DATA statement. The VIEW= option tells SAS to compile, but not to execute, the source program and to store the compiled code in the input DATA step view that is named in the option.

For example, the following statements create a DATA step view named Dept.A:

```
libname dept 'SAS-library';

data dept.a / view=dept.a;
  ... more SAS statements ...
run;
```

If the SAS view exists in a SAS library and if you use the same member name to create a new view definition, then the old SAS view is overwritten.

Beginning with SAS 8, DATA step views retain source statements. You can retrieve these statements using the DESCRIBE statement. The following example uses the DESCRIBE statement in a DATA step view in order to write a copy of the source code to the SAS log:

```
data view=inventory;
  describe;
run;
```

For more information about how to create SAS views and use the DESCRIBE statement, see the DATA statement in [SAS DATA Step Statements: Reference](#).

---

## What Can You Do with a DATA Step View?

Using a DATA step view, you can do the following:

- directly process any file that can be read with an INPUT statement
- read other SAS data sets
- generate data without using any external data sources and without creating an intermediate SAS data file

Because DATA step views are generated by the DATA step, they can manipulate and manage input data from a variety of sources including data from external files and data from existing SAS data sets. The scope of what you can do with a DATA step view, therefore, is much broader than that of other types of SAS views.

---

## Differences between DATA Step Views and Stored Compiled DATA Step Programs

DATA step views and stored compiled DATA step programs differ in the following ways:

- a DATA step view is implicitly executed when it is referenced as an input data set by another DATA or PROC step. Its main purpose is to provide data, one record at a time, to the invoking procedure or DATA step.
- a stored compiled DATA step program is explicitly executed when it is specified by the PGM= option in a DATA statement. Its purpose is usually a more specific task, such as creating SAS data files, or originating a report.

For more information about stored compiled DATA step programs, see [Chapter 30, “Stored Compiled DATA Step Programs,” on page 731](#).

---

## Restrictions and Requirements

Global statements do not apply to a DATA step view. Global statements such as the FILENAME, FOOTNOTE, LIBNAME, OPTIONS, and TITLE statements, even if included in the DATA step that created the SAS view, have no effect on the SAS view. If you do include global statements in your source program statements, SAS stores the DATA step view but not the global statements. When the view is referenced, actual execution can differ from the intended execution.

When a view is created, the labels for the variable that it returns are also created. If a DATA step view reads a data set that contains variable labels and a label is changed after the view is created, any procedure output will show the original labels. The view must be recompiled in order for the procedure output to reflect the new variable labels.

## Performance Considerations

- DATA step code executes each time you use a DATA step view, which might add considerable system overhead. In addition, you run the risk of having your data change between steps. However, this also means that you get the most recent data available—that is, data when the view is executed compared to data when the view was compiled.
  - Depending on how many reads or passes on the data are required, processing overhead increases.
    - When one sequential pass is requested, no data set is created. Compared to traditional methods of processing, making one pass improves performance by decreasing the number of input/output operations and elapsed time.
    - When random access or multiple passes are requested, the SAS view must build a spill file that contains all generated observations so that subsequent passes can read the same data that was read by previous passes. In some instances, the view SPILL= data set option can reduce the size of a spill file.
  - The VBUFSIZE= system option and the OBSBUF= data set option can be used to speed up execution time when processing DATA step views. For information about optimizing performance with SAS views, see “[Setting VBUFSIZE= and OBSBUF= for SAS DATA Step Views](#)” on page 225.
- 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*.

---

## Example 1: Merging Data to Produce Reports

If you want to merge data from multiple files but you do not need to create a file that contains the combined data, you can create a DATA step view of the combination for use in subsequent applications.

For example, the following statements define DATA step view MyV9Lib.QTR1, which merges the sales figures in the data file V9lr.Clothes with the sales figures in the data file V9lr.Equip. The data files are merged by date, and the value of the variable `Total` is computed for each date.

```
libname myv9lib 'SAS-library';
libname v9lr 'SAS-library';

data myv9lib.qtr1 / view=myv9lib.qtr1;
  merge v9lr.clothes v9lr.equip;
    by date;
    total = cl_v9lr + eq_v9lr;
run;
```

The following PRINT procedure executes the view:

```
proc print data=myv9lib.qtr1;
run;
```

## Example 2: Producing Additional Output Files

In this example, the DATA step reads an external file named Student, which contains student data, and then writes observations that contain known problems to data set MyV9Lib.Problems. The DATA step also defines the DATA step view MyV9Lib.Class. The DATA step does not create a SAS data file named MyV9Lib.Class.

The FILENAME and the LIBNAME statements are both global statements and must exist outside of the code that defines the SAS view, because SAS views cannot contain global statements.

Here are the contents of the external file Student:

```
dutterono  MAT   3
lyndenall  MAT
frisbee    MAT   94
              SCI   95
zymeco     ART   96
dimette    94
mesipho    SCI   55
merlbeest   ART   97
scafarnia   91
gilhoolie   ART  303
misqualle   ART   44
xylotone    SCI   96
```

Here is the DATA step that produces the output files:

```
libname myv9lib 'SAS-library';
filename student 'external-file-specification'; 1

data myv9lib.class(keep=name major credits)
      myv9lib.problems(keep=code date) / view=myv9lib.class; 2
  infile student;
  input name $ 1-10 major $ 12-14 credits 16-18; 3

  select;
  when (name=' ' or major=' ' or credits=.)
    do code=01;
       date=datetime();
       output myv9lib.problems;
    end; 4
  when (0<credits<90)
    do code=02;
       date=datetime();
       output myv9lib.problems;
    end; 5
  otherwise
    output myv9lib.class;
  end;
run; 6
```

The following example shows how to print the files created previously. The MyV9Lib.Class contains the observations from Student that were processed without

errors. The data file MyV9Lib.Problems contains the observations that contain errors.

If the data frequently changes in the source data file Student, there would be different effects on the returned values in the SAS view and the SAS data file:

- New records, if error free, that are added to the source data file Student between the time you run the DATA step in the previous example and the time you execute PROC PRINT in the following example, appear in the SAS view MyV9Lib.Class.
- On the other hand, if any new records, failing the error tests, were added to Student, the new records would not show up in the SAS data file MyV9Lib.Problems, until you run the DATA step again.

A SAS view dynamically updates from its source files each time it is used. A SAS data file, each time it is used, remains the same, unless new data is written directly to the file.

```
filename student 'external-file-specification';
libname myv9lib 'SAS-library'; 7

proc print data=myv9lib.class;
run; 8

proc print data=myv9lib.problems;
format date datetime18.;
run; 9
```

- 1 Reference a library called MyV9Lib. Tell SAS where a file that associated with the fileref Student is stored.
- 2 Create a data file called Problems and a SAS view called Class and specify the column names for both data sets.
- 3 Select the file that is referenced by the fileref Student and select the data in character format that resides in the specified positions in the file. Assign column names.
- 4 When data in the column Name, Major, or Credits is blank or missing, assign a code of 01 to the observation where the missing value occurred. Also assign a SAS datetime code to the error and place the information in a file called Problems.
- 5 When the number of credits is greater than zero, but less than ninety, list the observations as code 02 in the file called Problems and assign a SAS datetime code to the observation.
- 6 Place all other observations, which have none of the specified errors, in the SAS view called MyV9Lib.Class.
- 7 The FILENAME statement assigns the fileref Student to an external file. The LIBNAME statement assigns the libref MyV9Lib to a SAS library.
- 8 The first PROC PRINT calls the SAS view MyV9Lib.Class. The SAS view extracts data on the fly from the file referenced as Student.
- 9 This PROC PRINT prints the contents of the data file MyV9Lib.Problems.

---

## PROC SQL Views

A PROC SQL view is a PROC SQL query expression that is given a name and stored for later use. When you use a PROC SQL view in a SAS program, the view derives its data from the data sets (often referred to as tables) or SAS views listed in its FROM clause. The data that is accessed by the view is a subset or superset of the data in its underlying data sets or SAS views.

A PROC SQL view can read or write data from:

- DATA step views
- SAS data files
- other PROC SQL views
- SAS/ACCESS views
- DB2, ORACLE, or other DBMS data

For complete documentation on how to create and use PROC SQL views, see “[SQL Procedure](#)” in [SAS SQL Procedure User’s Guide](#).

For information about using PROC SQL views created in an earlier release, see Chapter 35, “Cross-Release Compatibility and Migration,” on page 779.

---

## Comparing DATA Step and PROC SQL Views

To help you decide between a DATA step view and a PROC SQL view, consider the characteristics of each type of SAS view:

- DATA step views
  - DATA step views are versatile because they use DATA step processing, including DO loops and IF-THEN/ELSE statements.
  - DATA step views do not have Update capability. That is, they cannot directly change the data that they access.
  - There is no way to qualify the data in a DATA step view before using it. Therefore, even if you need only part of the data in your SAS view, you must load into memory the entire DATA step view and discard everything that you do not need.
  - When a WHERE clause is applied to a DATA step view, the WHERE clause is evaluated by the DATA step view engine.
- PROC SQL views
  - PROC SQL views can combine data from many different file formats.
  - PROC SQL views can both read and update the data that they reference.
  - PROC SQL supports more types of WHERE clauses than are available in DATA step processing and has a CONNECT TO component that enables you

to easily send SQL statements and pass data to a DBMS by using the pass-through facility.

- You can also use the SQL language to subset your data before processing it. This capability saves memory when you have a large SAS view, but need to select only a small portion of the data contained in the view.
- PROC SQL views do not use DATA step programming.
- When a WHERE clause is applied to a PROC SQL view, the WHERE clause might be evaluated by the PROC SQL view engine, or the WHERE clause might be evaluated by the underlying library's engine.

---

## SAS/ACCESS Views

A SAS/ACCESS view is an interface view, also called a view descriptor, which accesses DBMS data that is defined in a corresponding access descriptor.

Using SAS/ACCESS software, you can create an access descriptor and one or more view descriptors in order to define and access some or all of the data described by one DBMS table or DBMS view. You can also use view descriptors in order to update DBMS data, with certain restrictions.

In addition, some SAS/ACCESS products provide a dynamic LIBNAME engine interface. If available, it is recommended that you use SAS/ACCESS LIBNAME statement to assign a SAS libref to your DBMS data because it is more efficient and easier to use than access descriptors and view descriptors. The SAS/ACCESS dynamic LIBNAME engine enables you to treat DBMS data as if it were SAS data by assigning a SAS libref to DBMS objects. Using a SAS/ACCESS dynamic LIBNAME engine means that you can use both native DATA step views and native PROC SQL views to access DBMS data instead of view descriptors.

See [Chapter 33, “About SAS/ACCESS Software,” on page 757](#) or the SAS/ACCESS documentation for your database for more information about SAS/ACCESS features.

For information about using SAS/ACCESS view descriptors created in an earlier release, see [Chapter 35, “Cross-Release Compatibility and Migration,” on page 779](#).

**Note:** Starting in SAS 9, PROC SQL views are the preferred way to access relational DBMS data. You can convert existing SAS/ACCESS view descriptors into PROC SQL views by using the CV2VIEW procedure. This enables you to use the LIBNAME statement to access your data. See the CV2VIEW Procedure in *SAS/ACCESS for Relational Databases: Reference*.

# 30

# Stored Compiled DATA Step Programs

---

<i>Definition of a Stored Compiled DATA Step Program</i> .....	731
<i>Uses for Stored Compiled DATA Step Programs</i> .....	732
<i>Restrictions and Requirements for Stored Compiled DATA Step Programs</i> .....	732
<i>How SAS Processes Stored Compiled DATA Step Programs</i> .....	732
<i>Creating a Stored Compiled DATA Step Program</i> .....	733
Syntax for Creating a Stored Compiled DATA Step Program .....	733
Process to Compile and Store a DATA Step Program .....	734
Example: Creating a Stored Compiled DATA Step Program .....	734
<i>Executing a Stored Compiled DATA Step Program</i> .....	735
Syntax for Executing a Stored Compiled DATA Step Program .....	735
Process to Execute a Stored Compiled DATA Step Program .....	736
Using Global Statements .....	736
Redirecting Output .....	737
Printing the Source Code of a Stored Compiled DATA Step Program .....	737
Example: Executing a Stored Compiled DATA Step Program .....	738
<i>Differences between Stored Compiled DATA Step Programs and DATA Step Views</i> .....	739
<i>Example of DATA Step Program</i> .....	739
Quality Control Application .....	739

---

## Definition of a Stored Compiled DATA Step Program

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.

**Note:** Stored compiled programs are available for DATA step applications only. Your stored programs can contain all SAS language elements except global statements. If you do include global statements in your source program, SAS stores

the compiled program. However, SAS does not store the global statements, and it does not display a warning message in the SAS log.

---

## Uses for Stored Compiled DATA Step Programs

The primary use of stored compiled DATA step programs is for executing production jobs. The advantage of using these DATA step programs is that you can execute them as needed without investing the resources required for repeated compilation. The savings are especially significant if the DATA step contains many statements. If you install a new version of SAS, you do not need to recompile your source code.

---

## Restrictions and Requirements for Stored Compiled DATA Step Programs

The following restrictions and requirements apply for using stored compiled DATA step programs:

- Stored compiled DATA step programs are available for DATA step applications only.
- Stored compiled DATA step program cannot contain global statements. If you do include global statements such as FILENAME, FOOTNOTE, LIBNAME, OPTIONS, and TITLE in your source program, SAS stores the compiled program but not the global statements. SAS does not display a warning message in the SAS log.
- SAS does not store raw data in the compiled program.

**Operating Environment Information:** You cannot move a compiled program to an operating environment that has an incompatible computer architecture. You must, instead, recompile your source code and store your new compiled program.

You can, however, move your compiled program to a different host computer that has a compatible architecture.

---

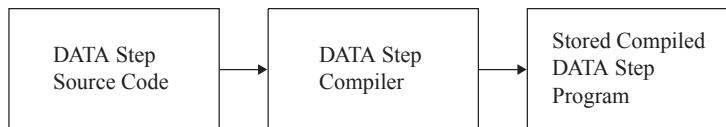
## How SAS Processes Stored Compiled DATA Step Programs

You first compile the SAS source program and store the compiled code. Then, execute the compiled code, redirecting the input and output as necessary.

SAS processes the DATA step through the compilation phase and then stores an intermediate code representation of the program and associated data tables in a SAS file. SAS processes the intermediate code when it executes the stored

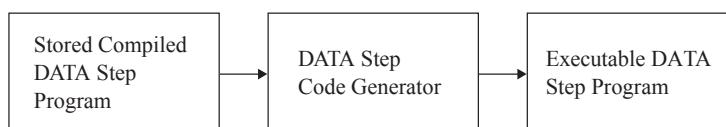
program. The following figure shows the process for creating a stored compiled DATA step program.

**Figure 30.1** Creating a Stored Compiled Program



When SAS executes the stored program, it resolves the intermediate code produced by the compiler and generates the executable machine code for that operating environment. The following figure shows the process for executing a stored compiled DATA step program.

**Figure 30.2** Executing a Stored Compiled Program



To move, copy, rename, or delete stored programs, use the DATASETS procedure or the utility windows in your windowing environment.

## Creating a Stored Compiled DATA Step Program

### Syntax for Creating a Stored Compiled DATA Step Program

The syntax for creating a stored compiled DATA step program is as follows:

**DATA** *data-set-name(s)* / PGM=*stored-program-name*

<(<*password-option*><SOURCE=*source-option*>)>;

where

**data-set-name**

specifies a valid SAS name for the output data set created by the source program. The name can be a one-level name or a two-level name. You can specify more than one data set name in the DATA statement.

**stored-program-name**

specifies a valid SAS name for the SAS file containing the stored program. The name can be a one-level name, but it is usually a two-level name. Stored programs are assigned the member type PROGRAM in the SAS library.

**password-option**

assigns a password to a stored compiled DATA step program.

source-option  
enables you to save or encrypt the source code.

For complete information about the DATA statement, see [SAS DATA Step Statements: Reference](#).

## Process to Compile and Store a DATA Step Program

To compile and store a DATA step program, do the following:

- 1 Write, test, and debug the DATA step program that you want to store.  
If you are reading external raw data files or if you write raw data to an external file, use a fileref rather than the actual filename in your INFILE and FILE statements so that you can redirect your input and output when the stored program executes.
- 2 When the program runs correctly, submit it using the PGM= option in the DATA statement.

The PGM= option tells SAS to compile, but not execute, the program and to store the compiled code in the SAS file named in the option. SAS sends a message to the log when the program is stored.

**Note:** The default SOURCE=SAVE or SOURCE=ENCRYPT options automatically save your source code.

**Note:** If you move your application to another operating environment, you need to recompile your source code and store your new compiled program.

## Example: Creating a Stored Compiled DATA Step Program

The following example uses the information in the input SAS data set In.Sample to assign a plant type based on a plant code. Note that the global LIBNAME statements are necessary to identify the storage location for your files, but are not part of Stored.Sample, the DATA step that SAS stores.

```
libname in 'SAS-library';
libname stored 'SAS-library';

data out.sample / pgm=stored.sample;
  set in.sample;
  if code = 1 then
    do;
      Type='Perennial';
      number+4;
    end;
  else
    if code = 2 then
      do;
        Type='Annual';
        number+10;
      end;
    else
      do;
```

```
Type='ERROR';
Number=0;
end;
run;
```

**Example Code 30.1** Partial SAS Log Identifying the Stored DATA Step Program

```
.
.
.
NOTE: DATA STEP program saved on file Stored.Sample.
NOTE: A stored DATA STEP program cannot run under a different operating
      system.
NOTE: DATA statement used (Total process time):
      real time          0.17 seconds
      cpu time           0.01 seconds
```

---

## Executing a Stored Compiled DATA Step Program

---

### Syntax for Executing a Stored Compiled DATA Step Program

The syntax for executing a stored compiled DATA step program, retrieving source code, and redirecting input or output, is as follows:

*...global SAS statements...*

**DATA PGM=***stored-program-name* <(password-option)>;  
   <DESCRIBE;>  
   <REDIRECT INPUT | OUTPUT *old-name-1 = new-name-1*<... *old-name-n = new-name-n*>;>  
   <EXECUTE;>

where

*global SAS statements*

specifies any global SAS statements that are needed by the program when it executes, such as a FILENAME or a LIBNAME statement that points to input files or routes output.

*stored-program-name*

specifies a valid SAS name for the SAS file containing the stored program. The name can be a one-level name or a two-level name.

*password-option*

specifies a password that you use to access the stored compiled DATA step program.

**DESCRIBE**

is a SAS statement that retrieves source code from a stored compiled DATA step program or a DATA step view.

**Note:** To DESCRIBE a password-protected DATA step program, you must specify its password. If the program has more than one password, you must specify the most restrictive password (with ALTER being the most restrictive and READ the least restrictive). For more information, see “[DESCRIBE Statement](#)” in [SAS DATA Step Statements: Reference](#).

**INPUT | OUTPUT**

specifies whether you are redirecting input or output data sets. When you specify INPUT, the REDIRECT statement associates the name of the input data set in the source program with the name of another SAS data set. When you specify OUTPUT, the REDIRECT statement associates the name of the output data set with the name of another SAS data set.

*old-name*

specifies the name of the input or output data set in the source program.

*new-name*

specifies the name of the input or output data set that you want SAS to process for the current execution.

**EXECUTE**

is a SAS statement that executes a stored compiled DATA step program.

For complete information about the DATA statement, see “[DATA Statement](#)” in [SAS DATA Step Statements: Reference](#).

## Process to Execute a Stored Compiled DATA Step Program

To execute a stored compiled DATA step program, follow these steps:

- 1 Write a DATA step for each execution of the stored program. In this DATA step, specify the name of the stored program in the PGM= option of the DATA statement and include an optional password. You can do any of the following tasks:
  - Submit this DATA step as a separate program.
  - Include it as part of a larger SAS program that can include other DATA and procedure (PROC) steps.
  - Point to different input and output SAS data sets each time you execute the stored program by using the REDIRECT statement.
- 2 Submit the DATA steps. Be sure to end each one with a RUN statement or other step boundary.

## Using Global Statements

You can use global SAS statements such as FILENAME or LIBNAME when you store or execute a stored compiled DATA step program. However, the global statements that you use to compile and store a DATA step program are not stored with the DATA step code.

---

## Redirecting Output

You can redirect external files using filerefs. You can use the REDIRECT statement for renaming input and output SAS data sets.

You can use the REDIRECT statement to redirect input and output data to data sets you specify. Note that the REDIRECT statement is available only for use with stored compiled DATA step programs.

**Note:** To redirect input and output stored in external files, include a FILENAME statement at execution time to associate the fileref in the source program with different external files.

**CAUTION! Use caution when you redirect input data sets.** The number and attributes of variables in the input SAS data sets that you read with the REDIRECT statement should match those of the input data sets in the SET, MERGE, MODIFY, or UPDATE statements of the source code. If they do not match, the following occurs:

- If the variable length attributes differ, the length of the variable in the source code data set determines the length of the variable in the redirected data set.
- If extra variables are present in the redirected data sets, the stored program stops processing, and an error message is sent to the SAS log.
- If the variable type attributes are different, the stored program stops processing, and an error message is sent to the SAS log.

---

## Printing the Source Code of a Stored Compiled DATA Step Program

If you use both the DESCRIBE and the EXECUTE statements when you execute a stored compiled DATA step program, SAS writes the source code to the log. The following example executes a stored compiled DATA step program. The DESCRIBE statement in the program writes the source code to the SAS log.

```
data pgm=stored.sample;
  describe;
  execute;
run;
```

**Example Code 30.2 Partial SAS Log Showing the Source Code Generated by the DESCRIBE Statement**

```

190  data pgm=stored.sample;
191      describe;
192      execute;
193  run;
NOTE: DATA step stored program Stored.Sample is defined as:

data out.sample / pgm=stored.sample;
    set in.sample;
    if code = 1 then
        do;
            Type='Perennial';
            number+4;
        end;
    else
        if code = 2 then
            do;
                Type='Annual';
                number+10;
            end;
        else
            do;
                Type='ERROR';
                Number=0;
            end;
    end;
run;

NOTE: DATA STEP program loaded from file Stored.Sample.
NOTE: There were 7 observations read from the data set In.Sample.
NOTE: The data set OUT.SAMPLE has 7 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time          0.03 seconds
      cpu time           0.00 seconds

```

For more information about the DESCRIBE statement, see [SAS DATA Step Statements: Reference](#).

---

## Example: Executing a Stored Compiled DATA Step Program

The following DATA step executes the stored program Stored.Sample created in “[Example: Creating a Stored Compiled DATA Step Program](#)” on page 734. The REDIRECT statement specifies the source of the input data as BASE.SAMPLE. The output from this execution of the program is redirected and stored in a data set named Totals.Sample. “[Partial SAS Log Identifying the Redirected Output File](#)” on page 739 shows part of the SAS log.

```

libname in 'SAS-library';
libname base 'SAS-library';
libname totals 'SAS-library';
libname stored 'SAS-library';

data pgm=stored.sample;
    redirect input in.sample=base.sample;
    redirect output out.sample=totals.sample;

```

```
run;
```

**Example Code 30.3 Partial SAS Log Identifying the Redirected Output File**

```
224  data pgm=stored.sample;
225      redirect input in.sample=base.sample;
226      redirect output out.sample=totals.sample;
227  run;
NOTE: DATA STEP program loaded from file Stored.Sample.
NOTE: There were 7 observations read from the data set BASE.SAMPLE.
NOTE: The data set Totals.Sample has 7 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time          0.12 seconds
      cpu time           0.01 seconds
228  proc printto; run;
```

## Differences between Stored Compiled DATA Step Programs and DATA Step Views

Stored compiled DATA step programs and DATA step views are similar in function. They both store DATA step programs that can retrieve and process data stored in other files. Both have the same restrictions and requirements (see “[Restrictions and Requirements for Stored Compiled DATA Step Programs](#)” on page 732). For information about DATA step views, see “[DATA Step Views](#)” on page 724.

Stored compiled DATA step programs and DATA step views differ in the following ways:

- A stored compiled DATA step program is explicitly executed when it is specified by the PGM= option in a DATA statement. The stored compiled DATA step is used primarily in production jobs.
- A DATA step view is implicitly executed when the view is referenced as an input data set by another DATA or procedure (PROC) step. Its main purpose is to provide data one record at a time to the invoking procedure or DATA step.
- You can use the REDIRECT statement when you execute a stored compiled DATA step. You cannot use this statement with DATA step views.

## Example of DATA Step Program

### Quality Control Application

This example illustrates how to use a stored compiled DATA step program for a simple quality control application. This application processes several raw data files. The source program uses the fileref Daily in the INFILE statement. Each DATA step

that is used to execute the stored program can include a FILENAME statement to associate the fileref Daily with a different external file.

The following statements compile and store the program:

```
libname stored 'SAS-library-1';

data flaws / pgm=stored.flaws;
  length Station $ 15;
  infile daily;
  input Station $ Shift $ Employee $ Flaws;
  Total + Flaws;
run;
```

The following statements execute the stored compiled program, redirect the output, and print the results:

```
libname stored 'SAS-library-1';
libname testlib 'SAS-library-2';

data pgm=stored.flaws;
  redirect output flaws=testlib.daily;
run;

proc print data=testlib.daily;
  title 'Quality Control Report';
run;
```

#### *Output 30.1 Quality Control Application Output*

#### Quality Control Report

Obs	Station	Shift	Employee	Flaws	Total
1	cambridge	1	Lin	3	3
2	Northampton	1	Kay	0	3
3	Springfield	2	Sam	9	12

Note that you can use the TITLE statement when you execute a stored compiled DATA step program or when you print the results.

## 31

# DICTIONARY Tables

---

<i>Definition of a DICTIONARY Table</i> .....	741
<i>How to View DICTIONARY Tables</i> .....	742
About Dictionary Tables .....	742
How to View a DICTIONARY Table .....	742
How to View a Summary of a DICTIONARY Table .....	742
How to View a Subset of a DICTIONARY Table .....	743
DICTIONARY Tables and Performance .....	744

---

## Definition of a DICTIONARY Table

A DICTIONARY table is a read-only SAS view that contains information about SAS libraries, SAS data sets, SAS macros, and external files that are in use or available in the current SAS session. A DICTIONARY table also contains the settings for SAS system options that are currently in effect.

When you access a DICTIONARY table, SAS determines the current state of the SAS session and returns the desired information accordingly. This process is performed each time a DICTIONARY table is accessed, so that you always have current information.

DICTIONARY tables can be accessed by a SAS program by using either of these methods:

- run a PROC SQL query against the table, using the DICTIONARY libref
- use any SAS procedure or the DATA step, referring to the PROC SQL view of the table in the Sashelp library

For more information about DICTIONARY tables, including a list of available DICTIONARY tables and their associated Sashelp views, see [SAS SQL Procedure User's Guide](#).

# How to View DICTIONARY Tables

## About Dictionary Tables

You might want to view the contents of DICTIONARY tables in order to see information about your current SAS session, before actually using the table in a DATA step or a SAS procedure.

Some DICTIONARY tables can become quite large. In this case, you might want to view a part of a DICTIONARY table that contains only the data that you are interested in. The best way to view part of a DICTIONARY table is to subset the table using a PROC SQL WHERE clause.

## How to View a DICTIONARY Table

Each DICTIONARY table has an associated PROC SQL view in the Sashelp library. You can see the entire contents of a DICTIONARY table by opening its Sashelp view with the VIEWTABLE or FSVIEW utilities. This method provides more detail than you receive in the output of the DESCRIBE TABLE statement, as shown in ["How to View a Summary of a DICTIONARY Table" on page 742](#).

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**  $\Rightarrow$  **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',
  idxxname 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 31.1 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.



## 32

# SAS Catalogs

---

<i>Definition of a SAS Catalog</i> .....	<b>747</b>
<i>SAS Catalog Names</i> .....	<b>748</b>
Parts of a Catalog Name .....	748
Accessing Information in Catalogs .....	748
<i>Tools for Managing SAS Catalogs</i> .....	<b>748</b>
<i>Profile Catalog</i> .....	<b>749</b>
Definition of a Profile Catalog .....	749
How the Information Is Used .....	749
How Sasuser.Profile Is Created .....	749
Default Settings .....	750
How to Recover Locked or Corrupt Profile Catalogs .....	750
<i>Catalog Concatenation</i> .....	<b>751</b>
Definitions .....	751
Example 1: LIBNAME Catalog Concatenation .....	751
Example 2: CATNAME Catalog Concatenation .....	753
Rules for Catalog Concatenation .....	754

---

## 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.

---

## SAS Catalog Names

---

### Parts of a Catalog Name

SAS catalog entries are fully identified by a four-level name in the following form:

*libref.catalog.entry-name.entry-type*

You commonly specify the two-level name for an entire catalog, as follows:

*libref.catalog*

*libref*

is the logical name of the SAS library to which the catalog belongs.

*catalog*

is a valid SAS name for the file.

The entry name and entry type are required by some SAS procedures. If the entry type has been specified elsewhere or if it can be determined from context, you can use the entry name alone. To specify entry names and entry types, use this form:

*entry-name.entry-type*

*entry-name*

is a valid SAS name for the catalog entry.

*entry-type*

is assigned by SAS when the entry is created.

---

### Accessing Information in Catalogs

In Base SAS software, SAS catalog entries are generally accessed automatically by SAS when the information stored in them is required for processing. In other SAS software products, you must specify the catalog entry in various procedures. Because the requirements differ with the SAS procedure or software product, see the appropriate procedure or product documentation for details.

---

## Tools for Managing SAS Catalogs

There are several SAS features to help you manage the entries in catalogs. The CATALOG procedure and the CEXIST function are two features of Base SAS software. Another tool is SAS Explorer, which enables you to view the contents of SAS catalogs. Many interactive windowing procedures contain a catalog directory window for managing entries. The following list summarizes the tools that are available for managing catalogs:

**CATALOG** procedure

is similar to the DATASETS procedure. Use the CATALOG procedure to copy, delete, list, and rename entries in catalogs.

**CEXIST** function

enables you to verify the existence of a SAS catalog or catalog entry. See the CEXIST function in *SAS Functions and CALL Routines: Reference* for more information.

**CATALOG** window

is a window that you can access at any time in an interactive windowing environment. It displays the name, type, description, and date of last update for each entry in the specified catalog. CATALOG window commands enable you to edit catalog entries. You can also view and edit catalog entries after double-clicking on a catalog file in SAS Explorer.

**catalog directory windows**

are available in some procedures in SAS/AF, SAS/FSP, and SAS/GRAFH software. A catalog directory window lists the same type of information that the CATALOG window provides: entry name, type, description, and date of last update. See the description of each interactive windowing procedure for details about the catalog directory window for that procedure.

---

## Profile Catalog

---

### Definition of a Profile Catalog

The Profile catalog (Sasuser.Profile) is a catalog that is available for customizing how you work with SAS. SAS uses this catalog to store function key definitions, fonts for graphics applications, window attributes, and other information from interactive windowing procedures.

---

### How the Information Is Used

The information in the Sasuser.Profile catalog is accessed automatically by SAS when you need it for processing. For example, each time you enter the KEYS window and change the settings, SAS stores the new settings with the KEYS entry type. Similarly, if you change and save the attributes for interactive window procedures, the changes are stored under the appropriate entry name and type. When you use the window or procedure, SAS then looks for information in the Profile catalog.

---

### How Sasuser.Profile Is Created

SAS creates the Profile catalog the first time it tries to refer to it and discovers that it does not exist. If you are using an interactive windowing environment, this occurs

during system initialization in your first SAS session. If you use one of the other modes of execution, the Profile catalog is created the first time you execute a SAS procedure that requires it.

At SAS start-up, SAS checks for an existing uncorrupted Sasuser.Profile catalog. If this catalog is found, then SAS copies the Sasuser.Profile catalog to Sasuser.Profbak. The backup is used if the Sasuser.Profile catalog becomes corrupted. For more information, see “[How to Recover Locked or Corrupt Profile Catalogs](#)” on page 750.

**Operating Environment Information:** The Sasuser library is implemented differently in various operating environments. See the SAS documentation for your host system for more information about how the Sasuser library is created.

## Default Settings

The default settings for your SAS session are stored in several catalogs in the Sashelp installation library. If you do not make any changes to key settings or other options, SAS uses the default settings. If you make changes, the new information is stored in your Sasuser.Profile catalog. To restore the original default settings, use the CATALOG procedure or the CATALOG window to delete the appropriate entries from your Profile catalog. By default, SAS then uses the corresponding entry from the Sashelp library.

During SAS sessions, you can make customizations, such as window resizing and positioning, and save them to Sasuser.Profile.

## How to Recover Locked or Corrupt Profile Catalogs

Occasionally, a Sasuser.Profile catalog becomes locked or corrupted. SAS uses Sashelp.Profile and Sasuser.Profbak to replace the locked or corrupted catalog.

If your Sasuser.Profile catalog is locked, SAS checks for Sashelp.Profile. If Sashelp.Profile exists, SAS copies it to Work.Profile and then saves the customizations in Work.Profile instead of in Sasuser.Profile. The following notes appear in the SAS log:

```
ERROR: Expecting page 1, got page -1 instead.
ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.
NOTE: Unable to open SASUSER.PROFILE. WORK.PROFILE will be opened instead.
NOTE: SASHELP.PROFILE has been copied to WORK.PROFILE.
NOTE: All profile changes will be lost at the end of the session.
```

If your Sasuser.Profile catalog is corrupted, SAS copies the corrupted catalog to Sasuser.Badpro. SAS then checks for Sasuser.Profbak. If Sasuser.Profbak exists, SAS copies it to Sasuser.Profile. Any changes made to the Sasuser.Profile catalog during the previous session is lost. The following notes appear in the SAS log:

```
ERROR: Expecting page 1, got page -1 instead.
ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.
NOTE: A corrupt SASUSER.PROFILE has been detected. A PROFILE catalog can
      become corrupt when a SAS session is prematurely terminated.
NOTE: SASUSER.PROFILE.CATALOG has been renamed to SASUSER.BADPRO.CATALOG.
NOTE: SASUSER.PROFILE.CATALOG has been restored from SASUSER.PROFBAK.CATALOG.
```

NOTE: Changes made to SASUSER.PROFILE.CATALOG during the previous SAS session have been lost. The type of data stored in the PROFILE catalog is typically related to SAS session customizations such as key definitions, fonts for graphics, and window attributes.

If your Sasuser.Profile catalog is corrupted and there is no Sasuser.Profbak, SAS checks for Sashelp.Profile. If Sashelp.Profile exists, SAS copies it to Work.Profile and then saves the customizations in Work.Profile instead of in Sasuser.Profile. The following notes appear in the SAS log:

ERROR: Expecting page 1, got page -1 instead.  
 ERROR: Page validation error while reading SASUSER.PROFILE.CATALOG.  
 NOTE: Unable to open SASUSER.PROFILE. WORK.PROFILE will be opened instead.  
 NOTE: SASHHELP.PROFILE has been copied to WORK.PROFILE.  
 NOTE: All profile changes will be lost at the end of the session.

## Catalog Concatenation

### Definitions

You can logically combine two or more SAS catalogs by concatenating them. This enables you to access the contents of several catalogs, using one catalog name. There are two ways to concatenate catalogs, using the LIBNAME statement and CATNAME statement.

#### LIBNAME catalog concatenation

results from a concatenation of libraries through a LIBNAME statement. When two or more libraries are logically combined through concatenation, any catalogs with the same name in each library become logically combined as well.

#### CATNAME catalog concatenation

is a concatenation that is specified by the global CATNAME statement in which the catalogs to be concatenated are specifically named. During CATNAME catalog concatenation, a logical catalog is set up in memory.

### Example 1: LIBNAME Catalog Concatenation

This LIBNAME statement concatenates the two SAS libraries:

```
libname both ('SAS-library-1' 'SAS-library-2');
```

Members of Library 1	Members of Library 2
MyCat.CATALOG	MyCat.CATALOG
Table1.DATA	MyCat2.CATALOG
Table3.DATA	Table1.DATA

Members of Library 1	Members of Library 2
	Table1.INDEX
	Table2.DATA
	Table2.INDEX

The concatenated libref Both would have the following:

Concatenated Libref Both
MyCat.CATALOG (from library 1 and 2)
MyCat2.CATALOG (from library 2)
Table1.DATA (from library 1)
Table2.DATA (from library 2)
Table2.INDEX (from library 2)
Table3.DATA (from library 1)

Notice that Table1.INDEX does not appear in the concatenation but Table2.INDEX does appear. SAS suppresses listing the index when its associated data file is not part of the concatenation.

So what happened to the catalogs when the libraries were concatenated? A resulting catalog now exists logically in memory, with the full name Both.MyCat.CATALOG. This catalog combines each of the two physical catalogs residing in *SAS-library-1* and *SAS-library-2*, called MyCat.CATALOG.

To understand the contents of the concatenation Both.MyCat, first look at the contents of both parts of the concatenation. Assume that the two original MyCat.CATALOG files contain the following:

Contents of MyCat.CATALOG in Library 1	Contents of MyCat.CATALOG in Library 2
A.FRAME	A.GRSEG
C.FRAME	B.FRAME
	C.FRAME

Then the combined catalog Both.MyCat contains the following:

Both.MyCat
A.GRSEG (from library 2)
A.FRAME (from library 1)

**Both.MyCat**

B.FRAME (from library 2)

C.FRAME (from library 1)

## Example 2: CATNAME Catalog Concatenation

The syntax of the CATNAME statement is:

```
CATNAME libref.catref
  (libref-1.catalog-1 (ACCESS=READONLY)
   libref-n.catalog-n (ACCESS=READONLY)) ;
```

To disassociate a concatenated catalog the syntax is:

```
CATNAME libref.catref | _ALL_ CLEAR;
```

In the following example, there must be a libref that is defined and named CatDog. The libref CatDog establishes the scope for the CATNAME concatenation definition.

**Note:** If a file in CatDog named Combined.CATALOG already exists, it becomes inaccessible until the CATNAME concatenation CatDog.Combined is cleared.

Members of Library 1	Members of Library 2
MyCat.CATALOG	MyDog.CATALOG
Table1.DATA	MyCat2.CATALOG
Table3.DATA	Table1.DATA
	Table1.INDEX
	Table2.DATA
	Table2.INDEX

If you issue the following statement,

```
catname catdog.combined
  (library1.mycat (access=readonly)
   library2.mydog (access=readonly));
```

then the concatenated catalog CatDog.Combined combines the following catalogs:

**Concatenated catalog CatDog.Combined**

MyCat.CATALOG (from library1)

MyDog.CATALOG (from library2)

**Note:** In CATNAME concatenation only the named catalogs are combined. In LIBNAME concatenation, any catalogs that have the same name in their respective libraries are concatenated when those libraries are concatenated.

The previous CATNAME statement creates a catalog that exists logically in memory. This catalog, named CatDog.Combined.CATALOG, combines the two physical catalogs residing in library1 and library2, called MyCat.CATALOG and MyDog.CATALOG respectively.

To understand the contents of the concatenation Combined.CATALOG, first look at the contents of both parts of the concatenation. The two original catalog files contain the following entries:

MyCat.CATALOG	MyDog.CATALOG
Library 1	Library 2
A.FRAME	A.GRSEG
C.FRAME	B.FRAME
	C.FRAME

The concatenated catalog Combined contains:

#### Combined.CATALOG contents

- A.GRSEG (from MyDog)
- A.FRAME (from MyCat)
- B.FRAME (from MyDog)
- C.FRAME (from MyCat)

---

## Rules for Catalog Concatenation

The rules for catalog concatenation are the same, whether the catalogs are concatenated using the LIBNAME statement or the CATNAME statement.

- When a catalog entry is open for input or update, the parts are searched and the first occurrence of the specified entry is used.
- When an item is open for output, it is created in the catalog that is listed first in the concatenation.

**Note:** A new catalog entry is created in the first catalog even if there is an item with the same name in another part of the concatenation.

**Note:** If the first catalog in a concatenation that is opened for update does not exist, the item is written to the next catalog that exists in the concatenation.

- When you want to delete or rename a catalog entry, only the first occurrence of the entry is affected.

- Any time a list of catalog entries is displayed, only one occurrence of the catalog entry is shown.

**Note:** Even if a catalog entry occurs multiple times in the concatenation, only the first occurrence is shown.



# 33

## About SAS/ACCESS Software

---

<i>Definition of SAS/ACCESS Software</i> .....	757
<i>Dynamic LIBNAME Engine</i> .....	758
SAS/ACCESS LIBNAME Statement .....	758
Using Data Set Options with SAS/ACCESS Librefs .....	758
Embedding a SAS/ACCESS LIBNAME Statement in a PROC SQL View .....	759
<i>SQL Procedure Pass-Through Facility</i> .....	759
<i>ACCESS Procedure and Interface View Engine</i> .....	760
<i>DBLOAD Procedure</i> .....	761
<i>Interface DATA Step Engine</i> .....	762

---

## 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 37, “SAS Engines,” on page 803](#) 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 deassigned.

**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 a PROC SQL, 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 “[Embedding a SAS/ACCESS LIBNAME Statement in a PROC SQL View](#)” on page 759 and “[SQL Procedure Pass-Through Facility](#)” on page 759.

**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 [Chapter 29, “SAS Views,” on page 721](#) 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 dblload 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 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 accuptd dli call=func dbname=db ssa=ssa1;
  file accupd 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.



# Processing Data Using Cross-Environment Data Access (CEDA)

<i>Definition of Cross-Environment Data Access (CEDA) .....</i>	<b>765</b>
<i>Advantages of CEDA .....</i>	<b>766</b>
<i>SAS File Processing with CEDA .....</i>	<b>767</b>
What Types of Processing Does CEDA Support? .....	767
Behavioral Differences for Output Processing .....	767
Restrictions for CEDA .....	768
Understanding When CEDA Is Used to Process a File .....	769
Determining Whether Update Processing Is Allowed .....	771
<i>Alternatives to Using CEDA .....</i>	<b>772</b>
<i>Creating Files in a Different Data Representation .....</i>	<b>773</b>
<i>Examples of Using CEDA .....</i>	<b>773</b>
Example 1: Automatically Processing a File .....	773
Example 2: Creating a New File in a Different Data Representation .....	774
Example 3: Changing the Data Representation of an Existing File .....	774
Example 4: Specifying UTF-8 Encoding When You Change Data Representation .....	775

---

## Definition of Cross-Environment Data Access (CEDA)

Cross-environment data access (CEDA) is a Base SAS feature. 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. This documentation explains the restrictions, benefits, and behavior of CEDA processing.

Here are a few concepts to help you understand CEDA:

**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 in memory. (See [Table 34.2 on page 770](#).)

- 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 in memory 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 character 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 transport file to 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

---

## What Types of Processing Does CEDA Support?

CEDA supports SAS 7 and later SAS files that are created in directory-based operating environments like UNIX, Windows, and OpenVMS. CEDA provides the following SAS file processing for these SAS engines:

**BASE**

default Base SAS engine for SAS 9 (V9), SAS 8 (V8), and SAS 7 (V7).

**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 for SAS 9 (V9TAPE), SAS 8 (V8TAPE), and SAS 7 (V7TAPE).

**Table 34.1** SAS File Processing Provided by CEDA

SAS File Type	Engine	Supported Processing
SAS data file	BASE, SASESOCK, SPDE, TAPE	input and output*
PROC SQL view	BASE	input
SAS/ACCESS view for Oracle or SAP	BASE	input
MDDB file**	BASE	input

\* For output processing that replaces an existing SAS data file, there are behavioral differences. For more information, see “[Behavioral Differences for Output Processing](#)” on page 767.

\*\* CEDA supports SAS 8 and later MDDB files.

---

## Behavioral Differences for Output Processing

For output processing that replaces an existing SAS data file, the engines behave differently regarding the following attributes:

encoding

- The BASE 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 BASE 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 SASSHARE or SASCONNECT, the default behavior is to use the encoding of the current SAS session.
- The SPD Engine uses the current SAS session encoding. The CLONE option of PROC COPY is not supported.

#### data representation

- The BASE and TAPE engines use the data representation of the current SAS session, except with PROC COPY.
- For both the BASE 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 SASSHARE or SASCONNECT, the default behavior is to use the data representation of the current SAS session.
- The SPD Engine uses the data representation of the current SAS session. The CLONE option of PROC COPY is not supported.

## Restrictions for CEDA

CEDA has the following restrictions:

- SAS catalogs are not supported. Catalog entries could include formats, stored compiled macros, SAS/AF applications, SASGRAPH 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, SASACCESS views that are not for SASACCESS 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 character 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 BASE and sequential engines on z/OS, see [SAS Companion for z/OS](#).

- Because the BASE engine translates the data as the data is read, multiple procedures require SAS to read and translate the data multiple times. In this way, the translation could 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 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. For information about encoding and transcoding, see the [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 “[Numerical Accuracy in SAS Software](#)” on page 72.
- 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 CPRT 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.

## Understanding When CEDA Is Used to Process a File

Because CEDA translation is transparent, you might not be aware when CEDA is being used. Knowing when CEDA is used could be helpful (for example, CEDA translation might require additional resources).

By default, SAS writes a message to the log when CEDA is used. Here is an example:

**Example Code 34.1 Log Output from Processing a File from a Different Operating Environment**

NOTE: Data file HEALTH.OXYGEN.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.

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.

In the following table, each row contains a group of operating environments that are compatible with each other. CEDA is used only when you create a file with a data representation in one row and process the file under a data representation of another row. (The current release of SAS does not run on some of these environments, but they are included here for completeness.)

**Table 34.2 Compatibility across Environments**

Data Representation Value	Environment
ALPHA_TRU64	Tru64 UNIX *
LINUX_IA64	Linux for Itanium-based systems*
LINUX_X86_64	Linux for x64 *
SOLARIS_X86_64	Solaris for x64 *
LINUX_POWER_64	Linux on the Power Architecture*
<hr/>	
ALPHA_VMS_32	OpenVMS Alpha **
<hr/>	
ALPHA_VMS_64	OpenVMS Alpha **
VMS_IA64	OpenVMS on HP Integrity**
<hr/>	
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
<hr/>	
HP_UX_32	HP-UX for PA-RISC
MIPS_ABI	MIPS ABI
RS_6000_AIX_32	AIX
SOLARIS_32	Solaris for SPARC
<hr/>	
LINUX_32	Linux for Intel architecture
INTEL_ABI	ABI for Intel architecture
<hr/>	
MVS_32	31-bit SAS on z/OS
<hr/>	

<b>Data Representation Value</b>	<b>Environment</b>
MVS_64_BFP	64-bit SAS on z/OS
OS2	OS/2 for Intel
VAX_VMS	OpenVMS VAX
WINDOWS_32	32-bit SAS on Microsoft Windows***
WINDOWS_64	64-bit SAS on Microsoft Windows (for both Itanium-based systems and 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. Linux on the Power Architecture is added in SAS Viya 3.5 and is not supported in SAS 9.
- \*\* Although these OpenVMS environments have different data representations for some compiler types, SAS data sets that are created by the BASE 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](#).
- \*\*\* Although these Windows environments are compatible, catalogs are an exception. Catalogs are not compatible between 32-bit and 64-bit SAS for Windows.

## Determining Whether Update Processing Is Allowed

If a file's data representation is the same as the data representation of the processing environment, and if the encoding is compatible with the currently executing SAS session encoding, then you can manually update the file, because CEDA is not needed in order to translate the file. For example, if a file was created in a 64-bit Solaris environment or if the OUTREP= option was used to designate the file with that data representation, then you can update the file in a 64-bit SAS session on Solaris for SPARC, HP-UX, or AIX.

Otherwise, if CEDA is used to translate the file, you cannot update the file. If you attempt to update the file, then you receive an error message stating that updating is not allowed. For example:

```
ERROR: File HEALTH.OXYGEN 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 HP_UX_64, RS_6000_AIX_64, SOLARIS_64, HP_IA64.
```

To determine the data representation and the encoding of a file, you can use the CONTENTS procedure (or the CONTENTS statement in PROC DATASETS). For example, the data set HEALTH.OXYGEN was created in a UNIX environment in SAS 9. The file was moved to a SAS 9 Windows environment, in which the following CONTENTS output was requested:

**Output 34.1** CONTENTS Output Showing Data Representation

The CONTENTS Procedure			
<b>Data Set Name</b>	HEALTH.OXYGEN	<b>Observations</b>	40
<b>Member Type</b>	DATA	<b>Variables</b>	7
<b>Engine</b>	V9	<b>Indexes</b>	0
<b>Created</b>	04/10/2014 15:54:03	<b>Observation Length</b>	56
<b>Last Modified</b>	04/10/2014 15:54:03	<b>Deleted Observations</b>	0
<b>Protection</b>		<b>Compressed</b>	NO
<b>Data Set Type</b>		<b>Sorted</b>	NO
<b>Label</b>			
<b>Data Representation</b>	HP_UX_64, RS_6000_AIX_64, SOLARIS_64, HP_IA64		
<b>Encoding</b>	latin1 Western (ISO)		

---

## 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:

### 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 long file and variable names.

### 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.

### 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.

#### Data transfer services in SAS/CONNECT software

Data transfer services is a bulk data transfer mechanism that transfers a disk 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 SASSHARE software

Remote library services gives you transparent access to remote data through the use of the LIBNAME statement.

## Creating Files in a Different Data Representation

By default, when you create a new file, SAS uses the data representation of the CPU that is running SAS. You can specify the OUTREP= option to override this default.

The OUTREP= option is both a SAS data set option and a LIBNAME statement option. The data set option applies to an individual file. The LIBNAME statement option applies to an entire library. This option uses CEDA to create a file that has a different data representation.

For example, in a UNIX environment, you can create a data set with the data representation of WINDOWS\_32. If you move that data set to a Windows environment and process the data set with 32-bit SAS for Windows, you do not invoke CEDA.

See “[Example 2: Creating a New File in a Different Data Representation](#)” on page 774 and “[Example 3: Changing the Data Representation of an Existing File](#)” on page 774.

See the OUTREP= option for the LIBNAME statement in [SAS DATA Step Statements: Reference](#) or see the OUTREP= data set option in [SAS Data Set Options: Reference](#).

## Examples of Using CEDA

### Example 1: Automatically Processing a File

This example shows how simple it is to process a SAS data set in a different operating environment without any conversion steps.

The data set was originally created on a UNIX environment and later moved to a Windows PC with a tool like FTP or operating system commands.

Using CEDA, SAS automatically recognizes the file's UNIX data representation and translates it to the data representation for the Windows environment. The log output displays a message that the file is being processed using CEDA.

```
libname Health 'c:\MyFiles';

proc print data=Health.Oxygen;
run;
```

**Example Code 34.2 Log Output from Processing a File from a Different Operating Environment**

NOTE: Data file HEALTH.OXYGEN.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.

---

## Example 2: Creating a New File in a Different Data Representation

In this example, an administrator who works in a z/OS operating environment wants to create a SAS file in a UNIX file system directory that can be processed in a 64-bit Linux environment. Specifying OUTREP=LINUX\_X86\_64 as a data set option forces the data representation to match the data representation of the Linux operating environment that later processes the file. This method of creating the file can enhance system performance because the file does not require data conversion later when it is processed on the Linux computer.

```
libname MyLib v9 'HFS-file-spec';

data MyLib.a (outrep=linux_x86_64);
  infile file-specifications;
  input student $ test1 test2 test3 final;
  total = test1+test2+test3+final;
  grade = total/4.0;
run;
```

---

## Example 3: Changing the Data Representation of an Existing File

To change an existing file's data representation, you can use PROC COPY with the NOCLONE option and specify OUTREP= in the LIBNAME statement. The following example copies a source library of Windows data sets to a target library of Solaris data sets. Note that if you move a data set to another platform (for example, with FTP), you must move it as a binary file.

```
libname Target 'target-pathname' outrep=solaris_x86_64;
libname Source 'source-pathname';
proc copy in=Source out=Target noclone memtype=data;
run;
```

For more information, see the OUTREP= option for the LIBNAME statement in [SAS DATA Step Statements: Reference](#) or see the OUTREP= data set option in [SAS Data Set Options: Reference](#).

---

## Example 4: Specifying UTF-8 Encoding When You Change Data Representation

SAS has a default session encoding that is based on two parameters of the current session: the operating environment and the locale. For a list of default encodings by locale and operating environment, see “[Default Values for DFLANG, DATESTYLE, and PAPERSIZE System Options Based on the LOCALE= System Option](#)” in [SAS National Language Support \(NLS\): Reference Guide](#).

If you do not want the default encoding, you can use language elements to specify an encoding for the session, or a library, or a data set. For example, many users choose UTF-8 encoding.

When you want to use a nondefault encoding such as UTF-8, be aware of an interaction with the OUTREP= data set option or LIBNAME statement option. 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 operating environment that is represented by the OUTREP= option and the locale of the current session, not the encoding of the current session. This example demonstrates the interaction.

In this example, the user is running SAS for Windows. The user’s locale is EN\_US. Normally, those two parameters would result in a default session encoding of wlatin1. However, the user’s encoding is set to UTF-8 in their SAS configuration file.

The following code checks the encoding and the locale:

```
proc options option=(encoding locale) define value;
run;
```

The code returns the following information in the SAS log:

```
Option Value Information For SAS Option ENCODING
      Value: UTF-8
      .
      .
      .

Option Value Information For SAS Option LOCALE
      Value: EN_US
```

The user wants to share data with another office. The other office is running SAS on 64-bit Linux. Their locale is EN\_US. Normally, that combination would result in a default session encoding of latin1. However, the user has been told that the other office has set their session encoding to UTF-8.

The user creates a data set. Because the data set will later be processed in the other office’s Linux environment, the user assigns the appropriate data representation. The user wants to avoid CEDA processing in the target environment, so the encoding and data representation must match the target. The user assumes that because their own environment and the Linux environment are both set to UTF-8 session encoding, an encoding specification is not required. The user sets the OUTREP= value only. This is not the correct syntax for this situation.

```
libname mylib 'C:\test';
```

```
data mylibtestdata (outrep=linux_x86_64);
x=1;
run;
```

The code generates a CEDA message in the log. The user expects this message, because the data representation LINUX\_X86\_64 does not match the current Windows environment.

NOTE: Data file MYLIB.TESTDATA.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.

NOTE: The data set MYLIB.TESTDATA has 1 observations and 1 variables.

The user checks their assumptions against the attributes of the data set:

```
proc contents data=mylibtestdata;
run;
```

The user is surprised. The data set does not have the session encoding of UTF-8. Instead, the encoding is latin1, which is the default for OUTREP=LINUX\_X86\_64 together with the LOCALE value of the current session, EN\_US.

<b>Data Set Name</b>	MYLIB.TESTDATA	<b>Observations</b>	1
<b>Member Type</b>	DATA	<b>Variables</b>	1
<b>Engine</b>	V9	<b>Indexes</b>	0
<b>Created</b>	05/10/2018 09:33:39	<b>Observation Length</b>	8
<b>Last Modified</b>	05/10/2018 09:33:39	<b>Deleted Observations</b>	0
<b>Protection</b>		<b>Compressed</b>	NO
<b>Data Set Type</b>		<b>Sorted</b>	NO
<b>Label</b>			
<b>Data Representation</b>	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
<b>Encoding</b>	latin1 Western (ISO)		

The correct way to prevent this issue is as follows. If you want a nondefault encoding value, then when you specify the OUTREP= option, you must also specify the ENCODING= option:

```
data mylib.test2 (outrep=linux_x86_64 encoding=utf8);
x=1;
run;
proc contents data=mylib.test2;
run;
```

Now the encoding and data representation both match the target environment:

<b>Data Set Name</b>	MYLIB.TEST2	<b>Observations</b>	1
<b>Member Type</b>	DATA	<b>Variables</b>	1
<b>Engine</b>	V9	<b>Indexes</b>	0
<b>Created</b>	05/10/2018 09:34:15	<b>Observation Length</b>	8
<b>Last Modified</b>	05/10/2018 09:34:15	<b>Deleted Observations</b>	0
<b>Protection</b>		<b>Compressed</b>	NO
<b>Data Set Type</b>		<b>Sorted</b>	NO
<b>Label</b>			
<b>Data Representation</b>	SOLARIS_X86_64, LINUX_X86_64, ALPHA_TRU64, LINUX_IA64		
<b>Encoding</b>	utf-8 Unicode (UTF-8)		

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*)).



# Cross-Release Compatibility and Migration

---

<i>Introduction to Cross-Release Compatibility and Migration</i> .....	779
<i>Accessing a File That Was Created in a Previous Release</i> .....	779
<i>Using SAS Files in a Previous Release</i> .....	780
Circumstances Where File Features Are Not Supported .....	780
File Features Not Supported in Previous Releases .....	780
Regressing a File to a Previous Release .....	782
<i>SAS Library Engines and the SAS File Format</i> .....	783

---

## Introduction to Cross-Release Compatibility and Migration

In many cases, you can process SAS files that were created in a different release without first converting the files. Compatibility is generally handled automatically by SAS. However, there are some limitations.

Compatibility between versions varies depending on the type of SAS file, the SAS release that you are running, the operating environment in which the file was created, and the type of processing you need to do.

---

## Accessing a File That Was Created in a Previous Release

SAS files that are created with SAS 7 and SAS 8 are generally compatible with SAS 9. However, if you change to a different operating environment or a different character encoding, you might encounter some processing restrictions. See [Chapter 34, “Processing Data Using Cross-Environment Data Access \(CEDA\),” on page 765](#).

For specific processing information and guidelines for migration issues, see the Migration Focus Area at [support.sas.com/migration](http://support.sas.com/migration). The Migration Focus Area is

your guide to migrating files from previous versions of SAS. Refer to this focus area for planning and cost analysis information, known compatibility issues and their resolutions, and step-by-step instructions. The MIGRATE procedure, which provides a simple way to migrate a library of SAS files from previous releases of SAS, is documented in [Base SAS Procedures Guide](#).

## Using SAS Files in a Previous Release

### Circumstances Where File Features Are Not Supported

SAS files that are created with SAS 9 are generally compatible with earlier releases.

However, new file features can be an important issue, such as in the following examples:

- You upgrade to a new release of SAS, but you cannot modify legacy data sets. You want to be aware of any SAS features that are not supported by the earlier file format.
- You exchange data with a SAS user who has not upgraded. You cannot use file features that are not supported by that version of SAS.

A few SAS file features can result in the following error message when the data is used under an earlier SAS release. These features are explained in the next topic.

ERROR: File MYLIB.TABLENAME.DATA not compatible with this SAS version.

## File Features Not Supported in Previous Releases

PROC SQL view that contains a USING clause

In SAS 9.4M6 and later releases, if you use the V9 engine to create a PROC SQL view that contains a USING clause, the view is not accessible in SAS 9.4M5 or prior releases. A USING clause stores DBMS connection information in the view by embedding the SAS/ACCESS LIBNAME statement inside the view.

AES and AES2 encryption

If you specify ENCRYPT=AES2 to encrypt a data set, the data set cannot be accessed by any release prior to SAS 9.4M5.

If you specify ENCRYPT=AES to encrypt a data set, the data set cannot be accessed by any release prior to SAS 9.4.

See “[AES Encryption](#)” on page 796.

Extended attributes

If you add extended attributes to a data set or to a variable in a data set, the data set cannot be accessed by any release prior to SAS 9.4. See “[Extended Attributes](#)” on page 716.

### Metadata-bound libraries

Metadata-bound data sets cannot be accessed by any release prior to SAS 9.3M2. See “[Metadata-Bound Libraries](#)” on page 801.

### Data sets created with EXTENDOBS COUNTER=YES

In SAS 9.4, extended observation count is the default. In SAS 9.3 it is optional. If a data set has EXTENDOBS COUNTER=YES, then it is not accessible by any release prior to SAS 9.3. The behavior depends on your operating environment. See “[Backward Compatibility of the Extended Observation Count Attribute](#)” on page 658.

**TIP** Set the system option EXTENDOBS COUNTER=NO for compatibility with releases prior to SAS 9.3.

### Greater than 2 billion observations

In SAS 9.2 and earlier releases, a data set with more than 2 billion observations is unusable in 32-bit SAS. See the information about [EXTENDOBS COUNTER=YES](#) on page 781.

### Extended naming for data sets, views, and item stores

SAS 9.3 and later releases support extended naming rules for data sets, views, and item stores. See “[Rules for SAS Data Set Names, View Names, and Item Store Names](#)” on page 28.

**TIP** Set the system option VALIDMEMNAME=COMPATIBLE to enforce naming that is compatible with releases prior to SAS 9.3. Messages are printed in the log when names cannot be transcoded.

### CHECK integrity constraints

If you add the constraint to an existing SAS data set or create a SAS data set that includes the constraint, the data set cannot be used in any release prior to SAS 9.2.

### Encoded passwords from the PWENCODE procedure

If you use PROC PWENCODE to encode a password, and you protect a data set with that password, then the data set might not be accessible by prior releases. For example, the SAS005 method is added in SAS 9.4M5. A data set that is protected with a SAS005-encoded password cannot be opened in any release prior to SAS 9.4M5. Protecting a data set with an encoded password is not supported prior to SAS 9.2.

### Linguistic collation with the SORT procedure

When you use SORTSEQ=LINGUISTIC with PROC SORT in SAS 9.2 and later, metadata is stored in the data set header to indicate that the data set was sorted. Previous releases do not support linguistic collation. If you use the sorted data set in previous releases, the data set is not shown to be sorted, and the COMPARE procedure might show differences due to the sort indicator. The data set can be accessed.

### Greater than 32,767 variables

A data set with greater than 32,767 variables is unusable in SAS 9.0 and earlier releases. See “[SAS Variable Attributes](#)” on page 38.

### Suppressed transcoding of a specified variable

Setting TRANSCODE=NO causes a data set to be unusable in SAS 9.0 and earlier releases. See *SAS National Language Support (NLS): Reference Guide*.

### Format or informat names longer than 8 bytes

A data set that uses format or informat names longer than 8 bytes is unusable in SAS 8 and earlier releases. For more compatibility information, see the VALIDFMTNAME system option in *SAS System Options: Reference*.

**TIP** Set VALIDFMTNAME=FAIL to get an error message for names that exceed the limit. If you specify the V7 or V8 engine, such as in a LIBNAME statement, SAS automatically uses the VALIDFMTNAME=FAIL behavior. Under SAS 6, use of long format names automatically causes a failure.

### Encoding attribute

The encoding attribute is not supported in SAS 6. SAS 7 and 8 data sets must be updated or output in a SAS 9 session to be stamped with an encoding attribute. (The encoding attribute was supported prior to SAS 9 in China, Korea, and Japan.)

**TIP** If you replace or update a data set that does not have an encoding attribute, then be aware that the session encoding is stamped on the data set by default. If that behavior is not desired, you can override the data set's encoding by using the DATA step option ENCODING= or the LIBNAME options INENCODING= or OUTENCODING=. If a session encoding is stamped on a data set incorrectly, and you are certain of the correct encoding, then you can set it with the CORRECTENCODING= option in the MODIFY statement of the DATASETS procedure. For the correct use of encoding language elements, see *SAS National Language Support (NLS): Reference Guide*.

### Variable names longer than 32 bytes

Longer variable names are not supported in SAS 6. See “Rules for SAS Variable Names” on page 26.

**TIP** Set VALIDVARNAME=V6 to get an error message.

## Regressing a File to a Previous Release

Regression is not necessary among SAS 9 releases. If you use a SAS 9 file in a SAS 8 session, regression is probably not necessary, due to cross-release compatibility.

If you need to regress a SAS 9 file to SAS 8 file format, you have some choices. You can use the COPY procedure with the NOCLONE option or the DATA step. The code must be run in a SAS 8 session. For details about regression, see the Migration Focus Area at [support.sas.com/migration](http://support.sas.com/migration).

# SAS Library Engines and the SAS File Format

The SAS 9 file format is very similar to that of SAS 8 and SAS 7. Therefore, the file extensions have not changed. For example, in SAS 9, SAS 8, and SAS 7, sas7bdat is the file extension for a data set in most operating environments.

The file format changed significantly from the SAS 6 file format, so SAS 6 file extensions are different from SAS 9, and they vary depending on the operating environment.

**Operating Environment Information:** For a complete list of SAS member types and extensions, see the SAS documentation for your operating environment.

You might not be in the habit of specifying the engine name when you assign a SAS library. SAS can usually assign one automatically, based on the default or on the SAS files that exist in the library location. (For details about engine assignment, see [Chapter 37, “SAS Engines,” on page 803](#).)

SAS can differentiate between a SAS 6 library and a SAS 9 library. However, because the file format for SAS 9, SAS 8, and SAS 7 files are very similar, SAS does not differentiate between them. For example, in a SAS 9 session, if you issue the following LIBNAME statement to assign a libref to a SAS library containing SAS 8 files, SAS automatically uses the V9 engine. If that library contains only SAS 6 files, then SAS automatically uses the V6 compatibility engine.

```
libname mylib 'SAS-library';
```

*Table 35.1 Default Library Engine Assignment in SAS 9*

SAS Library Contents	Default Engine Assignment
No SAS files; the library is empty	V9
Only SAS 9 SAS files	V9
Only SAS 8 SAS files	V9
Only SAS 7 SAS files	V9
Only SAS 6 SAS files	V6
Both SAS 9 SAS files and SAS files from earlier releases	V9



# File Protection

---

<b>Definition of a Password .....</b>	<b>786</b>
<b>Assigning Passwords .....</b>	<b>787</b>
Syntax .....	787
Assigning a Password with a DATA Step .....	787
Assigning a Password to an Existing Data Set .....	788
Assigning a Password with a Procedure .....	788
Assigning a Password with the SAS Windowing Environment .....	789
Assigning a Password outside of SAS .....	789
<b>Removing or Changing Passwords .....</b>	<b>789</b>
<b>Using Password-Protected SAS Files in DATA and PROC Steps .....</b>	<b>789</b>
<b>How SAS Handles Incorrect Passwords .....</b>	<b>790</b>
<b>Assigning Complete Protection with the PW= Data Set Option .....</b>	<b>790</b>
<b>Encoded Passwords .....</b>	<b>791</b>
<b>Using Passwords with Views .....</b>	<b>792</b>
Levels of Protection .....	792
PROC SQL Views .....	793
SAS/ACCESS Views .....	793
DATA Step Views .....	794
<b>SAS Data File Encryption .....</b>	<b>794</b>
About Encryption on SAS Data Files .....	794
SAS Proprietary Encryption .....	795
AES Encryption .....	796
AES Encryption and Referential Integrity Constraints .....	798
Passwords and Encryption with Generation Data Sets, Audit Trails, Indexes, and Copies .....	798
<b>Blotting Passwords and Encryption Key Values .....</b>	<b>798</b>
Check the SAS Log .....	798
Examples of Passwords and Encryption Keys That Are Not Blotted .....	799
Using Macros .....	800
Length of Passwords .....	800
<b>Metadata-Bound Libraries .....</b>	<b>801</b>

---

## 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 790](#).

**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 792](#).

---

# 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 36.1** Password-protected View

```

NOTE: DATA step view WORK.EXAM is defined as:

data exam / view=exam(read=XXX alter=XXXXXX);
   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 795](#) is implemented with the ENCRYPT=YES data set option.
- [AES \(Advanced Encryption Standard\) encryption on page 796](#) 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 36.1** 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
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 ENCRYPTIONKEY= data set option when the data set is created. You cannot change the ENCRYPTIONKEY= 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 CPRT 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
;
```

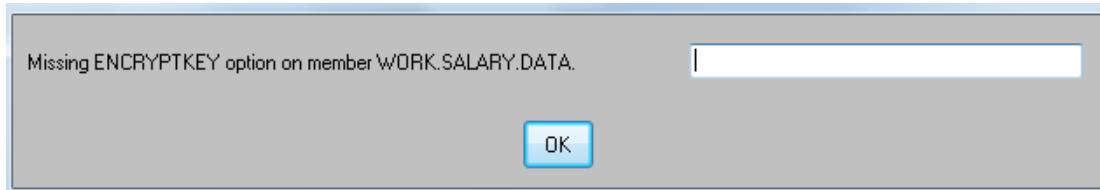
To print this data set, specify the ENCRYPTKEY= key value:

```
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 36.1** 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 795](#) and ["AES Encryption" on page 796](#).

## 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=XXXXXXXX) data=more(READ=secret2);
run;
```

- Typing errors cause the following passwords to show in the SAS log:

```
proc print data=library.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=XXXXXXXX XXXXXXXX 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*.



# SAS Engines

---

<i>Definition of a SAS Engine</i> .....	803
<i>Specifying an Engine</i> .....	803
<i>How Engines Work with SAS Files</i> .....	804
<i>Engine Characteristics</i> .....	805
About Engine Characteristics .....	805
Read/Write Activity .....	806
Access Patterns .....	806
Levels of Locking .....	807
Indexing .....	807
<i>About Library Engines</i> .....	808
Definition of a Library Engine .....	808
Native Library Engines .....	808
Interface Library Engines .....	810
<i>Special-Purpose Engines</i> .....	811
Character Variable Padding (CVP) Engine .....	811
SAS Information Maps LIBNAME Engine .....	811
SAS JMP LIBNAME Engine .....	811
SAS Metadata LIBNAME Engine .....	812
SAS XML LIBNAME Engine .....	812

---

## Definition of a SAS Engine

An engine is a component of SAS software that reads from or writes to a file. Each engine enables SAS to access files that are in a particular format. There are several types of engines.

---

## Specifying an Engine

Usually, you do not have to specify an engine. If you do not specify an engine, SAS automatically assigns one based on the contents of the SAS library.

Even though SAS automatically assigns an engine based on the library contents, it is more efficient for you to specify the engine. In some operating environments, in

order to determine the contents of a library, SAS must perform extra processing steps by looking at all of the files in the directory until it has enough information to determine which engine to use.

For example, if you explicitly specify the engine name as in the following LIBNAME statement, SAS does not need to determine which engine to use:

```
libname mylib v9 'SAS-library';
```

In order to use some engines, you must specify the engine name. For example, in order to use engines like the XML engine or the metadata engine, specify the engine name and specify specific arguments and options for that engine. For example, the following LIBNAME statement specifies the XML engine to import or export an XML document:

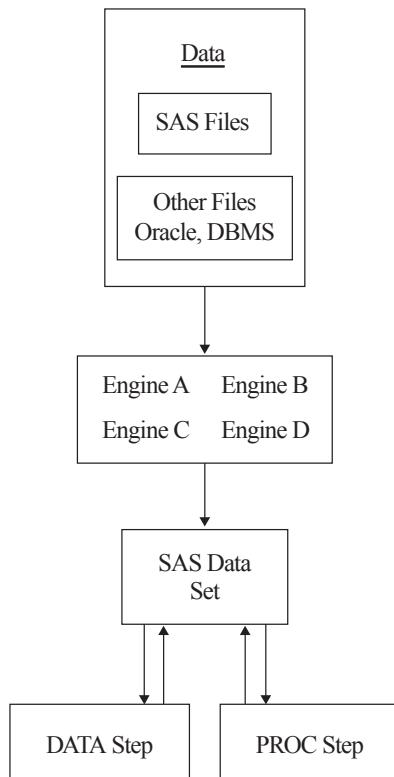
```
libname myxml xml 'C:\MyFiles\XML\MyXmlFile.xml' xmltype=generic;
```

You can specify an engine name in the LIBNAME statement, the ENGINE= system option, and in the New Library window.

## How Engines Work with SAS Files

The following figure shows how SAS data sets are accessed through an engine.

**Figure 37.1** How SAS Data Sets Are Accessed

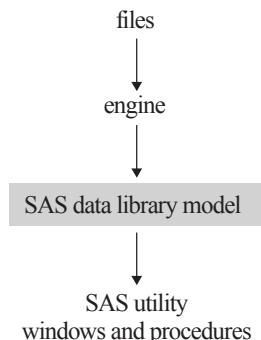


- Your data is stored in files for which SAS provides an engine. When you specify a SAS data set name, the engine locates the appropriate file or files.

- The engine opens the file and obtains the descriptive information that is required by SAS (for example, which variables are available and what attributes they have, whether the file has special processing characteristics such as indexes or compressed observations, and whether other engines are required for processing). The engine uses this information to organize the data in the standard logical form for SAS processing.
- This standard form is called the SAS data file, which consists of the descriptor information and the data values organized into columns (variables) and rows (observations).
- SAS procedures and DATA step statements access and process the data only in its logical form. During processing, the engine executes whatever instructions are necessary to open and close physical files and to read and write data in appropriate formats.

Data that is accessed by an engine is organized into the SAS data set model, and in the same way, groups of files that are accessed by an engine are organized in the correct logical form for SAS processing. Once files are accessed as a SAS library, you can use SAS utility windows and procedures to list their contents and to manage them. See [Chapter 26, “SAS Libraries,” on page 623](#) for more information about SAS libraries. The following figure shows the relationship of engines to SAS libraries.

**Figure 37.2 Relationship of Engines to SAS Libraries**




---

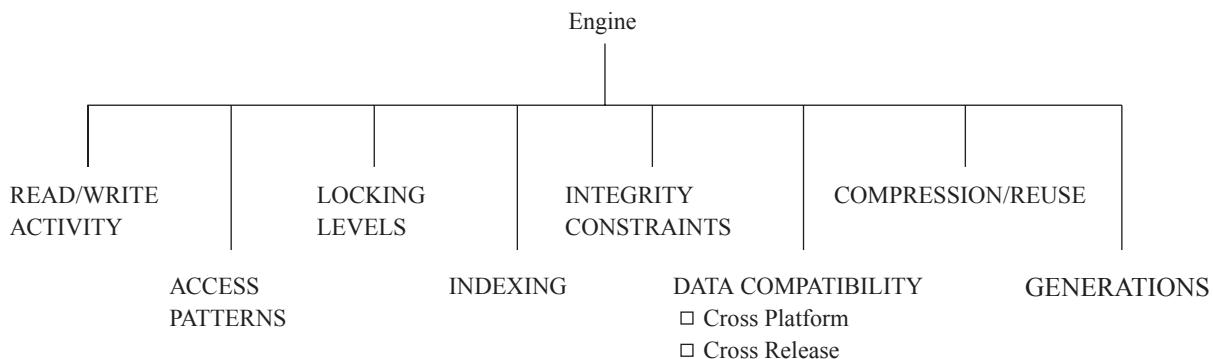
## Engine Characteristics

---

### About Engine Characteristics

The engine that is used to access a SAS data set determines its processing characteristics. Different statements and procedures require different processing characteristics. For example, the FSEDIT procedure requires the ability to update selected data values. And, 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.

The following figure describes the types of activities that engines regulate.

**Figure 37.3** Activities That Engines Regulate

## Read/Write Activity

An engine can perform one or more of the following tasks:

- limit read/write activity for a SAS data set to read-only
- fully support updating, deleting, renaming, or redefining the attributes of the data set and its variables
- support only some of these functions

For example, the engines that process BMDP, OSIRIS, or SPSS files support read-only processing. Some engines that process SAS views permit SAS procedures to modify existing observations while others do not.

---

## 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.

**random access**

processes observations according to the value of some indicator variable without processing previous observations.

**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 SAS statement or procedure tries to access a SAS data set whose engine does not support the required access pattern, SAS prints an appropriate error message in the SAS log.

---

## Levels of Locking

Some features of SAS require that data sets support different levels at which Update access is used. 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 determines how many sessions, procedures, or statements can read and write to the file at the same time. For example, with the FSEDIT procedure, you can request two windows on the same SAS data set in one session. Some engines support this capability; others do not.

The levels that are supported are record level and member (data set) level. Member-level 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 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. Not all engines support both levels.

By default, SAS provides the greatest possible level of concurrent access, while guaranteeing the integrity of the data. In some cases, you might want to guarantee the integrity of your data by controlling the levels of Update access yourself. Use the CNTLLEV= data set option to control levels of locking. CNTLLEV= enables locking at three levels:

- library
- data set
- observation

Here are situations in which you should consider using the CNTLLEV= data set option:

- Your application controls access to the data, such as in SAS Component Language (SCL), SAS/IML software, or DATA step programming.
- You access data through an interface engine that does not provide member-level control of the data.

For more information about the CNTLLEV= data set option, see [SAS Data Set Options: Reference](#).

You can also acquire an exclusive lock on an existing SAS file by issuing the LOCK global statement. After an exclusive lock is obtained, no other SAS session can read or write to the file until the lock is released. For more information about the LOCK statement, see [SAS DATA Step Statements: Reference](#).

**Note:** SAS products, such as SAS/ACCESS and SAS/SHARE, contain engines that support enhanced session management services and file locking capabilities.

---

## Indexing

A major processing feature of SAS is the ability to access observations by the values of key variables with indexes. See “[Understanding SAS Indexes](#)” on page

[692](#) for more information about using indexes for SAS data files. Note that not all engines support indexing.

---

## About Library Engines

---

### Definition of a Library Engine

A library engine is an engine that accesses groups of files and puts them into a logical form for processing by SAS utility procedures and windows. A library engine also determines the fundamental processing characteristics of the library and presents lists of files for the library directory. Library engines can be classified as native or interface.

---

## Native Library Engines

---

### Definition of Native Library Engine

A native library engine is an engine that accesses forms of SAS files that are created and processed only by SAS.

**Operating Environment Information:** Engine availability is host dependent. See the SAS documentation for your operating environment. Also, specific products provide additional engines.

### Default Base SAS Engine

The default Base SAS engine writes SAS libraries in disk format. The engine processes SAS 7, SAS 8, and SAS 9 files. If you do not specify an engine name when you are creating a new SAS library, the Base SAS engine, which for SAS 9 is named V9, is automatically selected.

When accessing existing SAS data sets on disk, SAS assigns an engine based on the contents of the library. The Base SAS engine has the following characteristics:

- It is the only engine that supports the full functionality of the SAS data set and the SAS library.
- It supports view engines.
- It meets all the processing characteristics required by SAS statements and procedures.
- It creates, maintains, and uses indexes.
- It reads and writes compressed (variable-length) observations. SAS data sets created by other engines have fixed-length observations.
- It assigns a permanent page size to data sets and temporarily assigns the number of buffers to be used when processing them.

- It repairs damaged SAS data sets, indexes, and catalogs.
- It enforces integrity constraints, creates backup files, and creates audit trails.

**Note:** SAS files created in SAS 7, 8, and 9 have the same file format.

## Remote Engine

The REMOTE engine is a SAS library engine for SAS/SERVEr software. Using it enables a SAS session to access shared data by communicating with a SAS server. For more information, see the [SAS/SERVEr User's Guide](#).

## SASESOCK Engine

The SASESOCK engine processes input to and output from TCP/IP ports instead of physical disk devices. The SASESOCK engine is required for SAS/CONNECT applications that implement MP CONNECT processing with the piping mechanisms. For more information, see the [SAS/CONNECT User's Guide](#).

## SAS Scalable Performance Data (SPD) Engine

The SAS Scalable Performance Data Engine (SPD Engine) provides parallel I/O, using multiple CPUs to read SAS data and deliver it rapidly to applications. The SPD Engine can process very large data sets because the data can span volumes but can be referenced as a single data set. The data in these data sets is also partitioned, enabling the data to be read in multiple threads per CPU. The SPD Engine is not intended to replace the default Base SAS engine for processing data sets that do not span volumes.

See [SAS Scalable Performance Data Engine: Reference](#) for details about this engine's capabilities.

## Sequential Engines

A sequential engine processes SAS files on storage media that do not provide random access methods (for example, tape or sequential format on disk). A sequential engine requires less overhead than the default Base SAS engine because sequential access is simpler than random access. However, a sequential engine does not support some Base SAS features such as audit trails, generation data sets, integrity constraints, and indexing.

The sequential engine supports some file types for backup and restore purposes only, such as CATALOG, VIEW, and MDDB. ITEMSTOR is the only file type that the sequential engine does not support. DATA is the only file type that is useful for purposes other than backup and restore.

The following sequential engines are available:

### V9TAPE (TAPE)

processes SAS 7, SAS 8, and SAS 9 files.

### V6TAPE

processes SAS 6 files without requiring you to convert the file to the SAS 9 format.

For more information, see [“Sequential Data Libraries” on page 635](#).

## Transport Engine

The XPORT engine processes transport files. The engine transforms a SAS file from its operating environment-specific internal representation to a transport file. A transport file is a machine-independent format that can be used for all hosts. In order to create a transport file, explicitly specify the XPORT engine in the LIBNAME statement, and then use the DATA step or COPY procedure.

For information about using the XPORT engine, see [Moving and Accessing SAS Files](#).

## 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 [Chapter 35, “Cross-Release Compatibility and Migration,” on page 779](#), or see the Migration Focus Area at [support.sas.com](http://support.sas.com).

## Interface Library Engines

An interface library engine is a SAS engine that accesses files formatted by other software. Interface library engines are not transparent to the user and must be explicitly specified (for example, in the LIBNAME statement).

The following are interface library engines:

### SPSS

reads SPSS portable file format. This file format is analogous to the transport format for SAS data sets. The SPSS portable files (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.

### OSIRIS

reads OSIRIS data and dictionary files in EBCDIC format.

### BMDP

reads BMDP save files.

In addition, a view engine is an interface library engine that is used by SAS/ACCESS software in order to retrieve data from files formatted by another vendor's software. These engines enable you to read and write data directly to and from files formatted by a database management system (DBMS), such as DB2 and ORACLE.

View engines enable you to use SAS procedures and statements in order to process data values stored in these files without the cost of converting and storing them in files formatted by SAS. Contact your on-site SAS support personnel for a list of the SAS/ACCESS interfaces available at your site. For more information about SAS/ACCESS features, see [Chapter 33, “About SAS/ACCESS Software,” on page 757](#) and the SAS/ACCESS documentation for your DBMS.

**Operating Environment Information:** The capabilities and support of these engines vary depending on your operating environment. See the SAS documentation for your operating environment for more complete information.

---

# Special-Purpose Engines

---

## Character Variable Padding (CVP) Engine

The character variable padding (CVP) engine expands character variable lengths, using a specified expansion amount, so that character data truncation does not occur when a file requires transcoding. Character data truncation can occur when the number of bytes for a character in one encoding is different from the number of bytes for the same character in another encoding, such as when a single-byte character set (SBCS) is transcoded to a double-byte character set (DBCS) or a multi-byte character set (MBCS).

The CVP engine is a read-only engine for SAS data files only. You can request character variable expansion by either of the following methods:

- You can explicitly specify the CVP engine (for example, with the LIBNAME statement, and using the default expansion of 1.5 times the variable lengths).
- You can implicitly specify the CVP engine with the LIBNAME statement options CVPBYTES= or CVPMULTIPLIER=. The options specify the expansion amount. In addition, you can use the CVPENGINE= option to specify the primary engine to use for processing the SAS file; the default is the default Base SAS engine.

For more information about using the CVP engine to avoid character data truncation and for details about the CVP engine options in the LIBNAME statement, see [SAS National Language Support \(NLS\): Reference Guide](#).

---

## SAS Information Maps LIBNAME Engine

The new SAS Information Maps LIBNAME engine provides a read-only way to access data generated from a SAS Information Map and to bring it into a SAS session. Once you retrieve the data, you can run almost any SAS procedure against it.

To use the Information Maps engine, specify INFOMAPS as the engine name, along with specific arguments and options in the LIBNAME statement.

For information about how to use the Information Maps engine, see [Base SAS Guide to Information Maps](#).

---

## SAS JMP LIBNAME Engine

The SAS JMP LIBNAME engine enables you to read and write JMP files in a Base SAS session. A JMP file is a file format that the JMP software program creates. JMP is an interactive statistics package that is available for Microsoft Windows and Macintosh. For more information about a JMP concept or term, see the JMP documentation that is packaged with your system.

To use the JMP engine, specify JMP as the engine name, along with the location of a SAS library in the LIBNAME statement. For example, the following code reads and prints five observations from the JMP file Baseball.jmp:

```
libname b jmp 'C:\JMP\SampleData';

proc print data=b.baseball (obs=5);
run;
```

For information about how to use the JMP engine, “[LIBNAME Statement: JMP Engine](#)” in *SAS Global Statements: Reference*

## SAS Metadata LIBNAME Engine

The metadata engine accesses metadata that is stored on the SAS Metadata Server within a specific SAS Metadata Repository. The metadata is information about the structure and content of data, and about the applications that process and manipulate that data. The metadata contains details such as the location of the data and the SAS engine that is used to process the data.

The metadata engine works in a similar way to other SAS engines. That is, you execute a LIBNAME statement to assign a libref and specify an engine. You then use that libref throughout the SAS session where a libref is valid. However, instead of the libref being associated with the physical location of a SAS library, the metadata libref is associated with specific metadata objects that are stored in a specific repository on the metadata server. The metadata objects define the SAS engine and options that are necessary to process a SAS library and its members.

When you execute the LIBNAME statement for the metadata engine, the metadata engine retrieves information about the target SAS library from the metadata. The metadata engine uses this information in order to construct a LIBNAME statement for the underlying engine and assigns it with the appropriate options. Then, when the metadata engine needs to access your data, the metadata engine uses the underlying engine to process the data.

You invoke the metadata engine by explicitly specifying the engine name META, along with specific arguments and options for the metadata engine (for example, in the LIBNAME statement or in the New Library window).

For information about how to use the metadata engine, see [SAS Language Interfaces to Metadata](#).

## SAS XML LIBNAME Engine

The SAS XML LIBNAME engine imports an XML document as one or more SAS data sets and exports a SAS data set as an XML document.

- The engine imports (reads from an input file) an external XML document by translating the XML markup into SAS proprietary format.
- The engine exports (writes to an output file) an XML document from a SAS data set by translating SAS proprietary format to XML markup.

To use the XML engine, specify either the `XML` or `XMLV2` engine nickname, along with specific arguments and options (for example, in the LIBNAME statement or in the New Library window).

For information about how to use the XML engine, see the [\*SAS XMLV2 and XML LIBNAME Engines: User's Guide\*](#).



# SAS File Management

<i>Improving Performance of SAS Applications</i> .....	815
<i>Moving SAS Files between Operating Environments</i> .....	815
<i>Repairing Damaged SAS Files</i> .....	816
Detecting Damage to SAS Files .....	816
Recovering SAS Data Files .....	816
Recovering Indexes .....	819
Recovering Disabled Indexes and Integrity Constraints .....	819
Recovering Catalogs .....	819

## Improving Performance of SAS Applications

SAS offers tools to control the use of memory and other computer resources. Most SAS applications run efficiently in your operating environment without using these features. However, if you develop applications under the following circumstances, you might want to experiment with tuning performance:

- You work with large data sets.
- You create production jobs that run repeatedly.
- You are responsible for establishing performance guidelines for a data center.
- You do interactive queries on large SAS data sets using SAS/FSP software.

For information about improving performance, see [Chapter 12, “Optimizing System Performance,” on page 217](#).

## Moving SAS 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](#).

---

## Repairing Damaged SAS Files

---

### Detecting Damage to SAS Files

The Base SAS engine detects possible damage to SAS data files (including indexes, integrity constraints, and the audit file) and SAS catalogs and provides a means for repairing some of the damage. If one of the following events occurs while you are updating a SAS file, SAS can recover the file and repair some of the damage:

- A system failure occurs while the data file or catalog is being updated.
- The disk where the data file (including the index file and audit 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 file, index file, audit file, or catalog.

When the failure occurs, the observations or records that were not written to the data file or catalog are lost and some of the information about where values are stored is inconsistent. The next time SAS reads the file, it recognizes that the file's contents are damaged and repairs it to the extent possible in accordance with the setting for the DLDMGACTION= data set option or system option, unless the data set is truncated. In this case, use the REPAIR statement to restore the data set.

If damage occurs to the storage device where a data file resides, you can restore the damaged data file, the index, and the audit file from a backup device.

**Note:** SAS is unable to repair or recover a SAS view (a DATA step view, an SQL view, or a SAS/ACCESS view) or a stored compiled DATA step program. If a SAS file of type VIEW or PROGRAM is damaged, you must re-create it.

**Note:** If the audit file for a SAS data file becomes damaged, you cannot process the data file until you terminate the audit trail. Then, you can initiate a new audit file or process the data file without one.

---

### Recovering SAS Data Files

To determine the type of action SAS takes when it tries to open a SAS data file that is damaged, set the DLDMGACTION= data set option or system option. That is, when a data file is detected as damaged, SAS automatically responds based on your specification as follows:

#### DLDMGACTION=FAIL

tells SAS to stop the step without a prompt and issue an error message to the SAS log indicating that the requested file is damaged. This specification gives the application control over the repair decision and provides awareness that a problem occurred.

To recover the damaged data file, you can issue the REPAIR statement in PROC DATASETS, which is documented in [Base SAS Procedures Guide](#).

**DLDMGACTION=ABORT**

tells SAS to terminate the step, issue an error message to the SAS log indicating that the request file is damaged, and end the SAS session.

**DLDMGACTION=REPAIR**

tells SAS to automatically repair the file and rebuild indexes, integrity constraints, and the audit file as well. If the repair is successful, a message is issued to the SAS log indicating that the open and repair steps were successful. If the repair is unsuccessful, processing stops without a prompt and an error message is issued to the SAS log indicating the requested file is damaged.

**Note:** If the data file is large, the time needed to repair it can be long.

**DLDMGACTION=NOINDEX**

tells SAS to automatically repair the data file, disable the indexes and integrity constraints, delete the index file, update the data file to reflect the disabled indexes and integrity constraints, and limit the data file to be opened only in INPUT mode. A warning is written to the SAS log instructing you to execute the PROC DATASETS REBUILD statement to correct the disabled indexes and integrity constraints and rebuild the index file. For more information, see ["Recovering Disabled Indexes and Integrity Constraints" on page 819](#).

**DLDMGACTION=PROMPT**

tells SAS to provide the same behavior that exists in Version 6 for both interactive mode and batch mode. For interactive mode, SAS displays a dialog box that asks you to select the FAIL, ABORT, or REPAIR action. For batch mode, the files fail to open.

For a data file, the date and time of the last repair and a count of the total number of repairs is automatically maintained. To display the damage log, use PROC CONTENTS as shown below:

```
proc contents data="c:\temp\testuser\large";
run;
```

## Output 38.1 CONTENTS Procedure Output

The SAS System			
The CONTENTS Procedure			
Data Set Name	TESTDATA.LARGE	Observations	10000
Member Type	DATA	Variables	2
Engine	V9	Indexes	0
Created	Wed, Dec 22, 2010 04:34:42 PM	Observation Length	112
Last Modified	Wed, Dec 22, 2010 04:35:24 PM	Deleted Observations	0
Protection		Compressed	NO
Data Set Type		Sorted	NO
Label			
Data Representation	WINDOWS_32		
Encoding	wlatin1 Western (Windows)		
Engine/Host Dependent Information			
Data Set Page Size	12288		
Number of Data Set Pages	92		
First Data Page	1		
Max Obs per Page	109		
Obs in First Data Page	98		
Number of Data Set Repairs	5		
Last Repair	16:35 Wednesday, December 22, 2010		
Filename	c:\temp\TestUser\large.sas7bdat		
Release Created	9.0301B0		
Host Created	NET_ASRV		
Alphabetic List of Variables and Attributes			
#	Variable	Type	Len
1	filler	Char	100
2	i	Num	8

---

## Recovering Indexes

In addition to the failures listed earlier, you can damage the indexes for SAS data files by using an operating environment command to delete, copy, or rename a SAS data file, but not its associated index file. The index is repaired similarly to the DLDMGACTION= option as described for SAS data files, or you can use the REPAIR statement in PROC DATASETS to rebuild composite and simple indexes that were damaged.

You cannot use the REPAIR statement to recover indexes that were deleted by one of the following actions:

- copying a SAS data file by some means other than PROC COPY or PROC DATASETS, for example, using a DATA step
- using the FORCE option in the SORT procedure to write over the original data file

In the above cases, the index must be rebuilt using the PROC DATASETS INDEX CREATE statement.

---

## Recovering Disabled Indexes and Integrity Constraints

When the DLDMGACTION=NOINDEX data set or system option is used and SAS encounters a damaged data file, SAS does the following:

- automatically repairs the data file without the indexes and integrity constraints
- disables the indexes and integrity constraints
- deletes the index file
- updates the data file to reflect the disabled indexes and integrity constraints
- limits the data file to be opened only in INPUT mode
- writes the following warning to the SAS log:

WARNING: SAS data file MYLIB.MYFILE.DATA was damaged and has been partially repaired. To complete the repair, execute the DATASETS procedure REBUILD statement.

The data file stays in INPUT mode until the PROC DATASETS REBUILD statement is executed. You use this statement to specify whether you want to restore the indexes and integrity constraints and rebuild the index file or delete the disabled integrity constraints and indexes. For more information, see the REBUILD statement in PROC DATASETS, which is documented in the *Base SAS Procedures Guide*.

---

## Recovering Catalogs

To determine the type of action that SAS takes when it tries to open a SAS catalog that is damaged, set the DLDMGACTION= data set option or system option. Then

when a catalog is detected as damaged, SAS automatically responds based on your specification.

**Note:** There are two types of catalog damage:

- localized damage is caused by a disk condition. This damage results in some data in memory not being flushed to disk. The catalog entries that are currently open for update are marked as damaged. Each damaged entry is checked to determine whether all the records can be read without error.
- severe damage is caused by a severe I/O error. The entire catalog is marked as damaged.

#### DLDMGACTION=FAIL

tells SAS to stop the step without a prompt and issue an error message to the log indicating that the requested file is damaged. This specification gives the application control over the repair decision and provides awareness that a problem occurred.

To recover the damaged catalog, you can issue the REPAIR statement in PROC DATASETS, which is documented in the *Base SAS Procedures Guide*. Note that when you use the REPAIR statement to restore a catalog, you receive a warning for entries that have possible damage. Entries that have been restored might not include updates that were not written to disk before the damage occurred.

#### DLDMGACTION=ABORT

tells SAS to terminate the step, issue an error message to the log indicating that the requested file is damaged, and end the SAS session.

#### DLDMGACTION=REPAIR

for localized damage, tells SAS to automatically check 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.

#### DLDMGACTION=PROMPT

for localized damage, tells SAS to provide the same behavior that exists in SAS 6 for both interactive mode and batch mode. For interactive mode, SAS displays a dialog box that asks you to select the FAIL, ABORT, or REPAIR action. For batch mode, the files fail to open. For severe damage, the entire catalog is copied to a new catalog.

Unlike data files, a damage log is not maintained for a catalog.

# External Files

---

<i>Definition of External Files</i> .....	821
<i>Referencing External Files Directly</i> .....	822
<i>Referencing External Files Indirectly</i> .....	822
<i>Referencing Many External Files Efficiently</i> .....	823
<i>Referencing External Files with Other Access Methods</i> .....	824
<i>Working with External Files</i> .....	826
Reading External Files .....	826
Writing to External Files .....	826
Processing External Files .....	827

---

## Definition of External Files

### external files

are files that are managed and maintained by your operating system, not by SAS. They contain data or text or are files in which you want to store data or text. They can also be SAS catalogs or output devices. Every SAS job creates at least one external file, the SAS log. Most SAS jobs create external files in the form of procedure output or output created by a DATA step.

External files used in a SAS session can store input for your SAS job as:

- records of raw data that you want to use as input to a DATA step
- SAS programming statements that you want to submit to the system for execution

External files can also store output from your SAS job as:

- a SAS log (a record of your SAS job).
- a report written by a DATA step.
- procedure output created by SAS procedures, including regular list output, and, beginning in Version 7, HTML and PostScript output from the Output Delivery System (ODS).

The PRINTTO procedure also enables you to direct procedure output to an external file. For more information, see “[PRINTTO Procedure](#)” in *Base SAS Procedures Guide*. See Chapter 9, “SAS Output,” on page 175 for more information about ODS.

**Note:** Database management system (DBMS) files are a special category of files that can be read with SAS/ACCESS software. For more information about DBMS files, see [Chapter 33, “About SAS/ACCESS Software,” on page 757](#) and the SAS/ACCESS documentation for your DBMS.

**Operating Environment Information:** Using external files with your SAS jobs entails significant operating-environment-specific information. For more information, see the SAS documentation for your operating environment.

## Referencing 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 39.1 Referencing External Files Directly*

External File Task	Tool	Example
Specify the file that contains input data.	INFILE	<pre>data weight;   infile 'input-file';   input idno \$ week1 week16;   loss=week1-week16;</pre>
Identify the file that the PUT statement writes to.	FILE	<pre>file 'output-file'; 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;</pre>
Bring statements or raw data from another file into your SAS job and execute them.	%INCLUDE	<pre>%include 'source-file';</pre>

## Referencing External Files Indirectly

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.<sup>1</sup> Note that you can assign

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

a fileref to a SAS catalog that is an external file, or to an output device, as shown in the following table.

**Table 39.2 Referencing External Files Indirectly**

External File Task	Tool	Example
Assign a fileref to a file that contains input data.	FILENAME	filename mydata 'input-file';
Assign a fileref to a file for output data.	FILENAME	filename myreport 'output-file';
Assign a fileref to a file that contains program statements.	FILENAME	filename mypgm 'source-file';
Assign a fileref to an output device.	FILENAME	filename myprinter <device-type> <host-options>;
Specify the file that contains input data.	INFILE	data weight; infile mydata; input idno \$ week1 week16; loss=week1-week16;
Specify the file that the PUT statement writes to.	FILE	file myreport; 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;
Bring statements or raw data from another file into your SAS job and execute them.	%INCLUDE	%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 39.3 Referencing Many Files Efficiently**

External File Task	Tool	Example
Assign a fileref to aggregate storage location.	FILENAME	filename mydir 'directory-or-PDS-name';
Specify the file that contains input data.	INFILE	<pre>data weight;   infile mydir(qrt1.data);   input idno \$ week1 week16;   loss=week1-week16;</pre>
Specify the file that the PUT statement writes to.*	FILE	<pre>file mydir(awards); if loss ge 5 then put idno loss   'AWARD STATUS=3'; else if loss ge 10   then put idno loss 'AWARD STATUS=2'; else if loss ge 15   then put idno loss 'AWARD STATUS=1'; run;</pre>
Bring statements or raw data from another file into your SAS job and execute them.	%INCLUDE	%include mydir(whole.program);

\* SAS creates a file that is named with the appropriate extension for your operating environment.

## Referencing External Files with Other Access Methods

You can assign filerefs to external files that you access with the following FILENAME access methods:

- CATALOG
- DATAURL
- FTP
- Hadoop
- SFTP
- TCP/IP SOCKET
- URL
- WebDAV

■ ZIP

Examples of how to use each method are shown in the following table:

**Table 39.4 Referencing External Files with Other Access Methods**

External File Task	Tool	Example
Assign a fileref to a SAS catalog that is an aggregate storage location.	FILENAME with CATALOG specifier	<code>filename mycat catalog 'catalog'                           &lt;catalog-options&gt;;</code>
Assign a fileref to an external file accessed by a data URL.	FILENAME with DATAURL specifier	<code>filename myfile dataurl 'external-file'                           &lt;dataurl-options&gt;;</code>
Assign a fileref to an external file accessed with FTP.	FILENAME with FTP specifier	<code>filename myfile FTP 'external-file'                           &lt;ftp-options&gt;;</code>
Assign a fileref to an external file accessed on a Hadoop Distributed File System.	FILENAME with Hadoop specifier	<code>filename myfile hadoop 'external-file'                           &lt;hadoop-options&gt;;</code>
Assign a fileref to an external file accessed with SFTP.	FILENAME with SFTP specifier	<code>filename myfile SFTP 'external-file'                           &lt;sftp-options&gt;;</code>
Assign a fileref to an external file accessed by TCP/IP SOCKET in either client or server mode.	FILENAME with SOCKET specifier	<code>filename myfile SOCKET 'hostname: portno'                           &lt;tcpip-options&gt;;</code> or <code>filename myfile SOCKET ':portno' SERVER                           &lt;tcpip-options&gt;;</code>
Assign a fileref to an external file accessed by URL.	FILENAME with URL specifier	<code>filename myfile URL 'external-file'                           &lt;url-options&gt;;</code>
Assign a fileref to an external file accessed on a WebDAV server.	FILENAME with WEBDAV specifier	<code>filename myfile WEBDAV 'external-file'                           &lt;webdav-options&gt;;</code>
Assign a fileref to a ZIP file accessed by using Zlib services.	FILENAME with ZIP specifier	<code>filename myfile ZIP 'external-file'                           &lt;zip-options&gt;;</code>

See [SAS DATA Step Statements: Reference](#) for detailed information about each of these statements.

---

## Working with External Files

---

### Reading External Files

The primary reason for reading an external file in a SAS job is to create a SAS data set from raw data. This topic is covered in [Chapter 21, “Reading Raw Data,” on page 471](#).

---

### Writing to External Files

You can write to an external file by using:

- a SAS DATA step
- the External File Interface (EFI)
- the Export Wizard.

When you use a DATA step to write a customized report, you write it to an external file. In its simplest form, a DATA step that writes a report looks like this:

```
data _null_;  
  set budget;  
  file 'your-file-name';  
  put variables-and-text;  
run;
```

For examples of writing reports with a DATA step, see [Chapter 20, “DATA Step Processing,” on page 441](#).

If your operating environment supports a graphical user interface, you can use the EFI or the Export Wizard to write to an external file. The EFI is a point-and-click graphical interface that you can use to read and write data that is not in SAS internal format. By using the EFI, you can read data from a SAS data set and write it to an external file, and you can read data from an external file and write it to a SAS data set. See [SAS/ACCESS Interface to PC Files: Reference](#) for more information about the EFI.

**Note:** If the data file you are passing to EFI is password protected, you are prompted multiple times for your login ID and password.

The Export Wizard guides you through the steps to read data from a SAS data set and write it to an external file. As a wizard, it is a series of windows that present simple choices to guide you through the process. See [SAS/ACCESS Interface to PC Files: Reference](#) for more information about the wizard.

---

## Processing External Files

When reading data from or to a file, you can also use a DATA step to:

- copy only parts of each record to another file
- copy a file and add fields to each record
- process multiple files in the same way in a single DATA step
- create a subset of a file
- update an external file in place
- write data to a file that can be read in different computer environments
- correct errors in a file at the bit level.

For examples of using a DATA step to process external files, see [Chapter 21, "Reading Raw Data," on page 471](#).



**PART 6****Industry Protocols Used in SAS**

<i>Chapter 40</i>		
<i>The SMTP E-Mail Interface</i>	.....	831
<i>Chapter 41</i>		
<i>Universal Unique Identifiers</i>	.....	835
<i>Chapter 42</i>		
<i>Internet Protocol Version 6 (IPv6)</i>	.....	839



# The SMTP E-Mail Interface

---

<i>Sending E-Mail through SMTP</i> .....	831
<i>System Options That Control SMTP E-Mail</i> .....	832
<i>Statements That Control SMTP E-mail</i> .....	833
FILENAME Statement .....	833
FILE and PUT Statements .....	833

---

## Sending E-Mail through SMTP

You can send electronic mail programmatically from SAS using the SMTP (Simple Mail Transfer Protocol) e-mail interface. SMTP is available for all operating environments in which SAS runs. To send SMTP e-mail with SAS e-mail support, you must have an intranet or Internet connection that supports SMTP.

Some SMTP servers require just the user identification as the login ID while others require the full e-mail address. The SAS SMTP e-mail interface authenticates the user identification in the following order.

- 1 If the user ID is specified by the `USERID=` option in the `EMAILHOST=` system option, the SAS SMTP e-mail interface attempts to authenticate by using this user ID.
- 2 If the user ID is not specified by the `USERID=` option, the SAS SMTP e-mail interface attempts to authenticate by using the user ID specified by the `FROM=` option of the `FILENAME=` statement.
- 3 If the user ID is not specified in the `FROM=` option in the `FILENAME=` statement, the SAS SMTP e-mail interface attempts to authenticate by using the user ID specified by the `EMAILID=` system option.
- 4 If the user ID is not specified by the `EMAILID=` system option, the SAS SMTP e-mail interface looks up the user ID from the operating system and attempts to authenticate that user ID.

For more information about sending e-mail from SAS, see the SAS documentation for your operating environment.

---

## System Options That Control SMTP E-Mail

Several SAS system options control SMTP e-mail. Depending on your operating environment and whether the SMTP e-mail interface is supported at your site, you might need to specify these options at start-up or in your SAS configuration file.

**Operating Environment Information:** To determine the default e-mail interface for your operating environment and to determine the correct syntax for setting system options, see the SAS documentation for your operating environment.

The EMAILSYS system option specifies which e-mail system to use for sending electronic mail from within SAS. For more information about the EMAILSYS system option, see the SAS documentation for your operating environment.

The following system options are specified only when the SMTP e-mail interface is supported at your site:

**EMAILACKWAIT=**

specifies the number of seconds that SAS will wait to receive an acknowledgment from an SMTP server. For more information, see the “[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.



# Universal Unique Identifiers

<b><i>Universally Unique Identifiers and the Object Spawner</i></b> .....	<b>835</b>
What Is a Universally Unique Identifier? .....	835
What Is the Object Spawner? .....	835
Defining the UUID Generator Daemon .....	836
Installing the UUID Generator Daemon .....	837
<b><i>Using SAS Language Elements to Assign UUIDs</i></b> .....	<b>838</b>
Overview of Using SAS Language Elements to Assign UUIDs .....	838
UUIDGEN Function .....	838
UUIDCOUNT= System Option .....	838
UUIDGENDHOST System Option .....	838

---

## 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 (UUIDGEN), which creates UUIDs for the requesting SAS session.

The **UUIDGEN** 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 **UUIDGEN** 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 **UUIDGEN** in an operating environment other than Windows, contact SAS Technical Support [http://support.sas.com/techsup/contact/\(\)](http://support.sas.com/techsup/contact/) to obtain a UUID node. The UUID node must be unique for each **UUIDGEN** installation in order for **UUIDGEN** to guarantee truly unique UUIDs.

Here is an example of a **UUIDGEN** setup configuration file for an operating environment other than Windows:

```
#  
## Define our UUID Generator Daemon. Since this UUIDGEN is  
## executing on a UNIX host, we contacted SAS Technical  
## Support to get the specified sasUUIDNode.  
#  
dn: sasSpawnercn=UUIDGEN,sascomponent=sasServer,cn=SAS,o=ABC Inc,c=US  
objectClass: sasSpawner  
sasSpawnercn: UUIDGEN  
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 **UUIDGEN** setup configuration file for Windows:

```
#  
## Define our UUID Generator Daemon. Since this UUIDGEN is  
## executing in a Windows operating environment, we do not need to specify  
## the sasUUIDNode.  
#  
dn: sasSpawnercn=UUIDGEN,sascomponent=sasServer,cn=SAS,o=ABC Inc,  
c=US  
objectClass: sasSpawner  
sasSpawnercn: UUIDGEN  
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 UUUIDGEN 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 UUUIDGEN 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 UUUIDGEN, you can use the following to assign UUIDs:

- UUUIDGEN function
- UUUIDCOUNT= system option
- UUUIDGENDHOST systems option

---

### UUIDGEN Function

The UUUIDGEN function returns a UUID for each cell. For more information, see “[UUIDGEN Function](#)” in *SAS Functions and CALL Routines: Reference*.

---

### UUIDCOUNT= System Option

The UUUIDCOUNT= 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 UUUIDGENDHOST 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*.

# Internet Protocol Version 6 (IPv6)

<i>Overview of IPv6</i> .....	839
<i>IPv6 Address Format</i> .....	840
<i>Examples of IPv6 Addresses</i> .....	840
Example of Full and Collapsed IPv6 Address .....	840
Example of an IPv6 Address That Includes a Port Number .....	841
Example of an IPv6 Address That Includes a URL .....	841
<i>Fully Qualified Domain Names (FQDN)</i> .....	841

---

## 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 42.1 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:

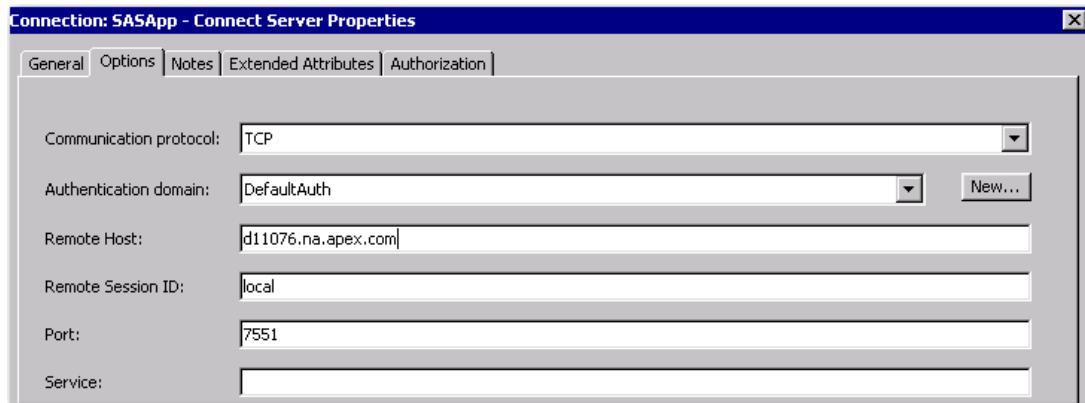
```
PROC METAOPERATE  
  SERVER="d6292.us.company.com"  
  PORT=2222  
  USERID="myuserid"  
  PASSWORD="mypassword"  
  PROTOCOL=BRIDGE  
  
  ACTION=RESUME  
  OPTIONS=""  
  NOAUTOPAUSE;
```

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 42.1** 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.